Gabe Testa

CS 680

10/3/11

Pong using Three.js

(WebGL Library)

# Goal:

For our first assignment, we were asked to create a pong game using openGL and shaders.

# Overview:

My pong project used a webgl library called Three.js (<https://github.com/mrdoob/three.js/>) written by a guy named Mr. Doob. Three.js encapsulates most of the webgl api’s into a nice little javascript package. I was able to go from having never used Three.js, to using shaders in about 3 hours. Using Three.js, I was able to create this pong game and add some cool effects to it. For my ball, I decided to make it the sun and have all the planets orbiting it while the user plays. All the textures for the planets are rendered using shaders, as well as the background texture. The sun dynamically lights all the planets as well, thanks to the fragment shader.

I spend a total of about 20 hours writing this program. It took me a while to understand the shaders and how to communicate with them using javascript. Once I figured out how to render the ball, everything else just fell into place. I found the shader code for the background and once I got that working, I decided to make the theme of the game somewhat of a solar system eating one.

# Manual:

Figure 1 shows the dropdown control board. Here you can choose how many players you want to have and change the camera mode. You can also mess with the background texture color from this menu.

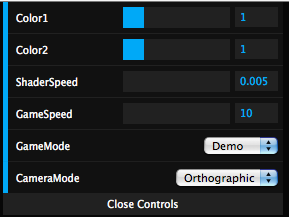


Figure Shader/Control Menu (courtesy of dat.gui.js)

Once you have selected the number of players, toggling the escape button will begin pause and unpause the game. The first one to 10 wins.

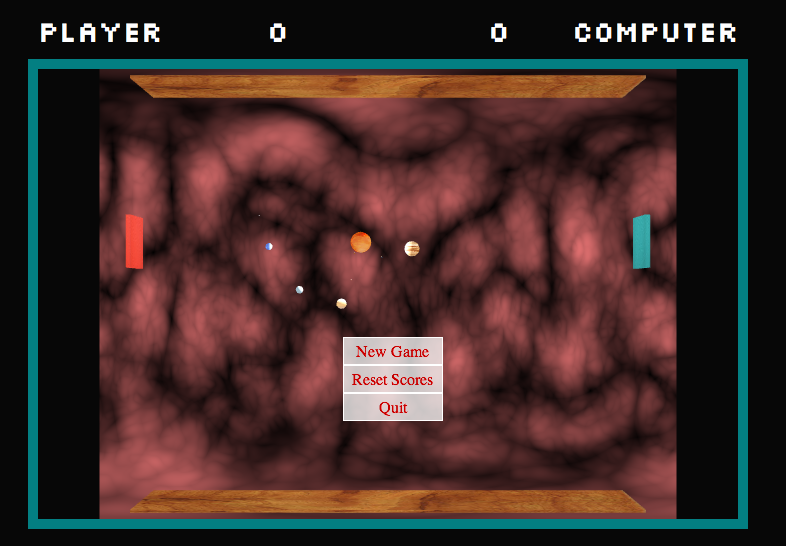


Figure Pong Game in Perspective View

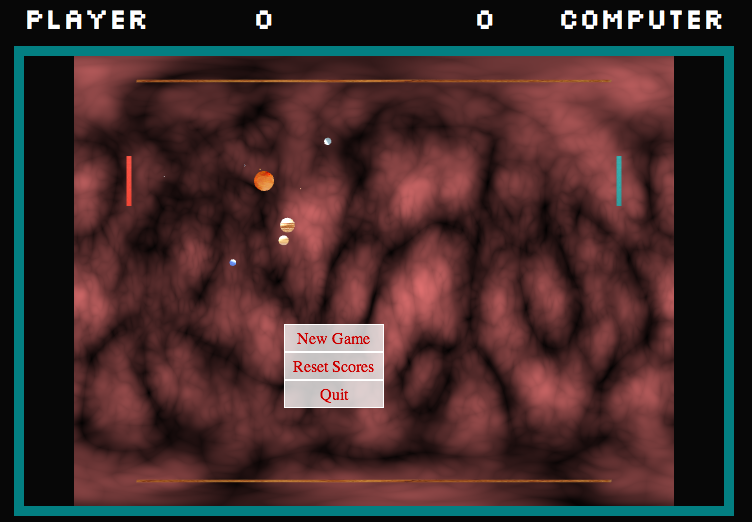
1

Figure Pong Game in Orthographic View

# Technical Overview:

I wrote a lot of code for this project. I probably wrote about 6 different shaders that didn’t work, so there may be some remnants in the code from those failed attempts that may be confusing.

Design wise, I knew I wanted to use Three.js for this project. Webgl is just going to get bigger, so I figured that if I had to write shader code, I might as well do it in webgl.

Deficiencies:

1. I did not use shaders to render every object. The pads and the borders are rendered in a webgl render, but solely on software.
2. I didn’t model the solar system completely. I didn’t offset the plane of rotation for a couple planets.
3. I didn’t make the AI that awesome.
4. I may have spent too much time of the look and not enough time of the mechanics.

Next time, I would incorporate a physics engine into the game and make the AI better.

# APPENDIX A (SHADERS):

Vertex and Fragment Shaders for planets:

Vertex:

// create a shared variable for the

// VS and FS containing the normal

varying vec3 vNormal;

varying vec2 vUv;

attribute float displacement;

uniform float amplitude;

void main() {

// set the vNormal value with

// the attribute value passed

// in by Three.js

vNormal = normal;

// push the displacement into the three

// slots of a 3D vector so it can be

// used in operations with other 3D

// vectors like positions and normals

vec3 newPosition = position +

normal \*

vec3(displacement \* amplitude);

// get position for texture

vUv = uv;

gl\_Position = projectionMatrix \*

modelViewMatrix \*

vec4(position,1.0);

}

Fragment:

// same name and type as VS

varying vec3 vNormal;

varying vec2 vUv;

//give it a changing light source

uniform vec3 lightsource;

uniform sampler2D image;

void main() {

//texture the object using the vUv points passed from the vertex shader

vec4 texture = texture2D(image, vUv);

// calc the dot product and clamp

// 0 -> 1 rather than -1 -> 1

vec3 light = lightsource;

// ensure it's normalized

light = normalize(light);

// calculate the dot product of

// the light to the vertex normal

float dProd = max(0.0, dot(vNormal, light));

// feed into our frag colour with our texture

gl\_FragColor = vec4(dProd, dProd, dProd, 1.0 ) + texture;

}

Vertex and Fragment Shaders for Background:

Vertex:

varying vec2 vUv;

uniform vec2 scale;

void main( void ) {

vUv = uv \* scale;

gl\_Position = projectionMatrix \* modelViewMatrix \* ve

c4( position, 1.0 );

}

Fragment:

//

// Description : Array and textureless GLSL 3D simplex noise function.

// Author : Ian McEwan, Ashima Arts.

// Maintainer : ijm

// Lastmod : 20110409 (stegu)

// License : Copyright (C) 2011 Ashima Arts. All rights reserved.

// Distributed under the MIT License. See LICENSE file.

//

uniform float time;

uniform float uSpeed;

uniform float color1;

uniform float color2;

varying vec2 vUv;

vec4 permute( vec4 x ) {

return mod( ( ( x \* 34.0 ) + 1.0 ) \* x, 289.0 );

}

vec4 taylorInvSqrt( vec4 r ) {

return 1.79284291400159 - 0.85373472095314 \* r;

}

float PI = 3.14159265;

float TWOPI = 6.28318531;

float BaseRadius = 1.0;

vec3 sphere( float u, float v) {

u \*= PI;

v \*= TWOPI;

vec3 pSphere;

pSphere.x = BaseRadius \* cos(v) \* sin(u);

pSphere.y = BaseRadius \* sin(v) \* sin(u);

pSphere.z = BaseRadius \* cos(u);

return pSphere;

}

float snoise( vec3 v ) {

const vec2 C = vec2( 1.0 / 6.0, 1.0 / 3.0 );

const vec4 D = vec4( 0.0, 0.5, 1.0, 2.0 );

// First corner

vec3 i = floor( v + dot( v, C.yyy ) );

vec3 x0 = v - i + dot( i, C.xxx );

// Other corners

vec3 g = step( x0.yzx, x0.xyz );

vec3 l = 1.0 - g;

vec3 i1 = min( g.xyz, l.zxy );

vec3 i2 = max( g.xyz, l.zxy );

vec3 x1 = x0 - i1 + 1.0 \* C.xxx;

vec3 x2 = x0 - i2 + 2.0 \* C.xxx;

vec3 x3 = x0 - 1. + 3.0 \* C.xxx;

// Permutations

i = mod( i, 289.0 );

vec4 p = permute( permute( permute(

i.z + vec4( 0.0, i1.z, i2.z, 1.0 ) )

+ i.y + vec4( 0.0, i1.y, i2.y, 1.0 ) )

+ i.x + vec4( 0.0, i1.x, i2.x, 1.0 ) );

// Gradients

// ( N\*N points uniformly over a square, mapped onto an octahedron.)

float n\_ = 1.0 / 7.0; // N=7

vec3 ns = n\_ \* D.wyz - D.xzx;

vec4 j = p - 49.0 \* floor( p \* ns.z \*ns.z ); // mod(p,N\*N)

vec4 x\_ = floor( j \* ns.z );

vec4 y\_ = floor( j - 7.0 \* x\_ ); // mod(j,N)

vec4 x = x\_ \*ns.x + ns.yyyy;

vec4 y = y\_ \*ns.x + ns.yyyy;

vec4 h = 1.0 - abs( x ) - abs( y );

vec4 b0 = vec4( x.xy, y.xy );

vec4 b1 = vec4( x.zw, y.zw );

vec4 s0 = floor( b0 ) \* 2.0 + 1.0;

vec4 s1 = floor( b1 ) \* 2.0 + 1.0;

vec4 sh = -step( h, vec4( 0.0 ) );

vec4 a0 = b0.xzyw + s0.xzyw \* sh.xxyy;

vec4 a1 = b1.xzyw + s1.xzyw \* sh.zzww;

vec3 p0 = vec3( a0.xy, h.x );

vec3 p1 = vec3( a0.zw, h.y );

vec3 p2 = vec3( a1.xy, h.z );

vec3 p3 = vec3( a1.zw, h.w );

// Normalise gradients

vec4 norm = taylorInvSqrt( vec4( dot( p0, p0 ), dot( p1, p1 ), dot( p2, p2 ), dot( p3, p3 ) ) );

p0 \*= norm.x;

p1 \*= norm.y;

p2 \*= norm.z;

p3 \*= norm.w;

// Mix final noise value

vec4 m = max( 0.6 - vec4( dot( x0, x0 ), dot( x1, x1 ), dot( x2, x2 ), dot( x3, x3 ) ), 0.0 );

m = m \* m;

return 42.0 \* dot( m\*m, vec4( dot( p0, x0 ), dot( p1, x1 ), dot( p2, x2 ), dot( p3, x3 ) ) );

}

float surface( vec3 coord ) {

float n = 0.0;

n += 0.7 \* abs( snoise( coord ) );

n += 0.25 \* abs( snoise( coord \* 2.0 ) );

n += 0.125 \* abs( snoise( coord \* 4.0 ) );

n += 0.0625 \* abs( snoise( coord \* 8.0 ) );

return n;

}

void main( void ) {

vec3 coord = sphere(vUv.y,vUv.x);

coord.x += -time;

coord.y += -time;

coord.z -= -uSpeed;

float n = surface( coord );

gl\_FragColor = vec4( vec3( n, n/color1, n/color2 ), 1.0 );

}