Space Scene

Shuli He, Zixuan Zhao, Jingcheng Shi, Eric Ota

ABSTRACT

Four graphical effects were created using Three.js and GLSL and combined into one scene to create a more elaborate space scene. Alpha blending, texturing techniques, particle systems, and perlin noise, are a few of the many techniques used to create these effects.

CCS CONCEPTS

Computing methodologies → Graphics systems and interfaces;

KEYWORDS

shading languages, alpha blending, perlin noise, particle systems, bump map

1) INTRODUCTION

The purpose of this project is to take a variety of different graphical effects using various techniques and combine them into one scene. Each team member was responsible for creating one effect. Using alpha blending, we incorporate a hidden image effect, allowing a single image to be seen two different ways depending on the background color. Two different starfields are incorporated into the background, one using a particle system and perlin noise, the other using a shader technique. Finally, we added some planets to the scene made using bump maps and other texturing techniques.

2) EXPERIMENTAL AND COMPUTATIONAL DETAILS

2.1)HIDDEN IMAGE

2.1.1) UNDERSTANDING THE EFFECT

Fig 2.1.1 same image with white background (left) and black background (right)

A single image can show two different pictures by changing the background color(black/white). The left one with white background shows nothing and the right one with a black background has a character. This kind of image is occasionally seen at mobile applications which display the image with white background and when the user click on and open the image it will be shown at fullscreen with black background.

2.1.2) ALPHA BLENDING

After reading some articles about this kind of image, it seems that alpha blending is the key to realize this technic. So, it start with the basic alpha blending. For alpha blending, to calculate the color there is a formula:

$$Color = Color_{front} * Alpha + Color_{back} * (1 - Alpha)$$

For image 1, we have: $Color_1 = (r_1, b_1, g_1, 1)$

For image 2, we have: $Color_2 = (r_2, b_2, g_2, 1)$

For two images mixed assume that the final result of the combined image color is $Color_{mix}$. Then have: $Color_{mix} = (r_{mix}, b_{mix}, g_{mix}, 1)$

Then we get the equation by putting the color into the alpha blending formula:

$$\begin{cases} r_1 = r_{mix} * a_{mix} + (1 - a_{mix}) \\ g_1 = g_{mix} * a_{mix} + (1 - a_{mix}) \\ b_1 = b_{mix} * a_{mix} + (1 - a_{mix}) \\ \\ r_2 = r_{mix} * a_{mix} \\ g_2 = g_{mix} * a_{mix} \\ b_2 = b_{mix} * a_{mix} \end{cases}$$

The solution of the equation should be:

$$\begin{cases} a_{mix} = 1 - r_1 + r_2 \\ a_{mix} = 1 - g_1 + g_2 \\ a_{mix} = 1 - b_1 + b_2 \\ \\ r_{mix} = r_2/(1 - r_1 + r_2) \\ g_{mix} = g_2/(1 - g_1 + g_2) \\ b_{mix} = b_2/(1 - b_1 + b_2) \end{cases}$$

Obviously, the solution would be simple if it's a grayscale image which means the rgb have the same value(r = g = b).

Thus, the solution for the mix image is:

$$\begin{cases} a_{mix} = 1 - r_1 + r_2 \\ r_{mix} = r_2/a_{mix} \end{cases}$$

Moreover, in consider of the range of alpha ([0,1]) we must fix the value of r_1 and r_2. The appropriate way is to scale down the value of r_2.
Therefore, hiding the image in the black background need combine a black image with the original image and the reverse can hid in the white.

2.2) STARFIELD (JUMP TRANSITION)

2.2.1) DESCRIBING THE EFFECT

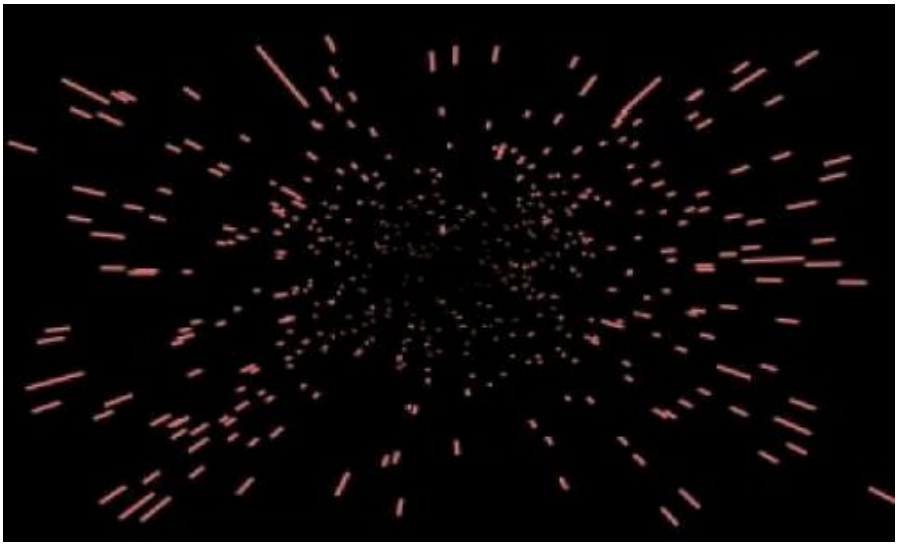


Fig 2.2.1 starfield screensaver

My favorite screensaver back in the days of Windows 98 was the starfield simulation that shipped with Windows itself. It was a very simple screensaver that represented stars as a collection of white pixels on a black screen, with each updated frame the 'stars' appear to move closer to the viewer by moving in a direction away from the center of the screen.

2.2.2) CREATING THE “JUMP” EFFECT

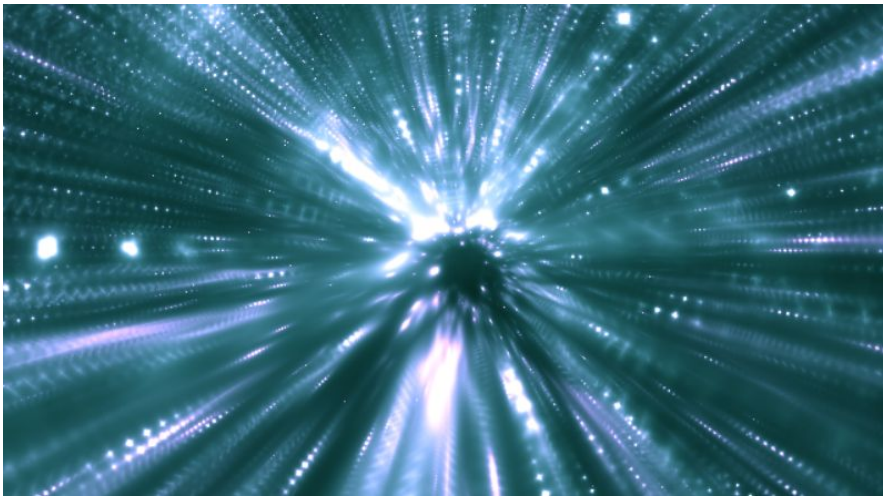


Fig 2.2.2 Jump Transition

Shadertoy.com provides amazing shaders for me, such like Interstellar and Warp Speed. I initially chose to implement Interstellar in three.js but it did not work well. The noise effect from the shader is not compatible with the WebGL version I used in my project and I could not find out why. Then I tried the Warp Speed and it looks great. As we know, these shadertoy are fragment shaders. So I wrote a basic vertex shader. Then I use the gl_Fragcolor and gl_Fragcoord to replace the fragCoord and fragcolor in original shader codes, which is only

provided in ShaderToy. Three.js part is pretty straightforward. I created a plane for shaders to render and update the shader global time to have such effect.

2.3) STARFIELD (PARTICLE SYSTEM)

2.3.1) PARTICLE SYSTEM

We use particle system for star effect. It uses particles that fade out quickly and are then re-emitted from the effect's source. A particle system is a technique in game physics, motion graphics, and computer graphics that uses a large number of very small sprites, 3D models, or other graphic objects to simulate certain kinds of "fuzzy" phenomena, which are otherwise very hard to reproduce with conventional rendering techniques - usually highly chaotic systems, natural phenomena, or processes caused by chemical reactions.

Typically a particle system's position and motion in 3D space are controlled by what is referred to as an emitter. The emitter acts as the source of the particles, and its location in 3D space determines where they are generated and where they move to. A regular 3D mesh object, such as a cube or a plane, can be used as an emitter. The emitter has attached to it a set of particle behavior parameters. These parameters can include the spawning rate (how many particles are generated per unit of time), the particle's' initial velocity vector (the direction they are emitted upon creation), particle lifetime (the length of time each individual particle exists before disappearing), particle color, and many more. It is common for all or most of these parameters to be "fuzzy" — instead of a precise numeric value, the artist specifies a central value and the degree of randomness allowable on either side of the center (i.e. the average particle's lifetime might be 50 frames ±20%). When using a mesh object as an emitter, the initial velocity vector is often set to be normal to the individual face(s) of the object, making the particles appear to "spray" directly from each face but optional.

A typical particle system's update loop (which is performed for each frame of animation) can be separated into two distinct stages, the parameter update/simulation stage and the rendering stage.

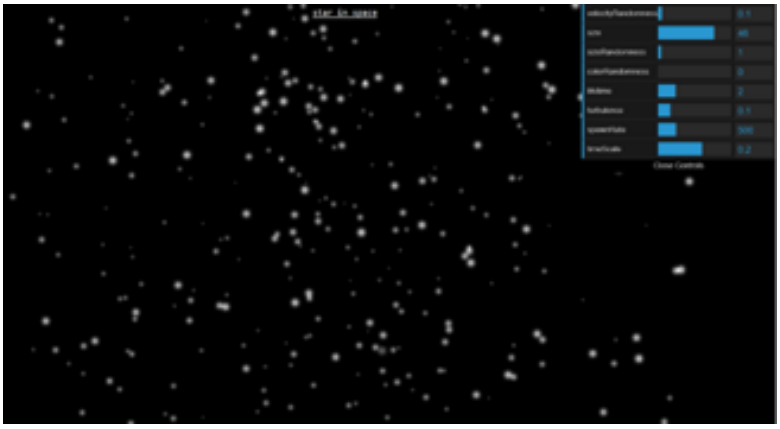


Fig 4.1.1 star effect using a particle system

2.3.2) PERLIN NOISE

We also employ one noise function for star effect. Perlin Noise, a technique used to produce natural appearing textures on computer generated surfaces for motion picture visual effects. The development of Perlin Noise has allowed computer graphics artists to better represent the complexity of natural phenomena in visual effects for the motion picture industry.

Perlin noise is a procedural texture primitive, a type of gradient noise used by visual effects artists to increase the appearance of realism in computer graphics. The function has a pseudo-random appearance, yet all of its visual details are the same size. This property allows it to be readily controllable; multiple scaled copies of Perlin noise can be inserted into mathematical expressions to create a great variety of procedural textures. Synthetic textures using Perlin noise are often used in CGI to make computer-generated visual elements – such as object surfaces, fire, smoke, or clouds – appear more natural, by imitating the controlled random appearance of textures in nature.

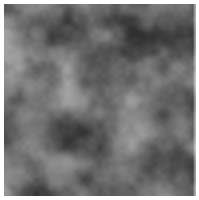
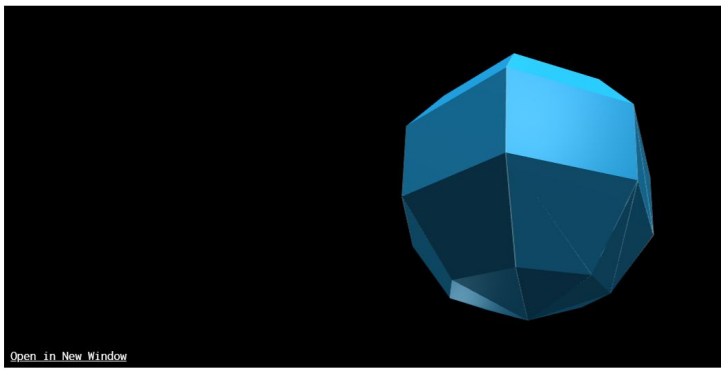


Fig 4.2.1 Perlin noise rescaled and added into itself to create fractal noise.

2.4) PLANETS

2.4.1) CREATING PLANETS AND MOONS WITH GEOMETRY



Example

```
var geometry = new THREE.SphereGeometry( 5, 32, 32 );
var material = new THREE.MeshBasicMaterial( {color: 0xffff00} );
var sphere = new THREE.Mesh( geometry, material );
scene.add( sphere );
```

Fig 2.4.1 SphereGeometry API documentation

The base for all of our planet and moon effects is sphere geometry, created using:
var planetGeometry = new THREE.SphereGeometry(float radius, integer width, integer height);

2.4.2) APPLYING TEXTURES

Fig 2.4.2 Earth bump map

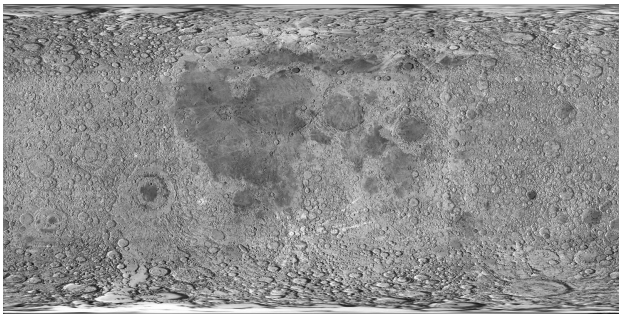


Fig 2.4.2 Moon texture

If you have good textures, it's really easy to make very nice looking planets. Simply load in a texture with THREE.TextureLoader() and add it to the material you want to mesh with your geometry.

2.4.3) USING A BUMP MAP

```
var earthGeometry = new THREE.SphereGeometry(10, 50, 50);
var earthMaterial = new THREE.MeshPhongMaterial({
  map: new THREE.TextureLoader().load('img/earth_texture.jpg'),
  bumpMap : new THREE.TextureLoader().load('img/earthbump1k.jpg'),
  bumpScale : 0.7,
  color: 0xf1f1f1
});
var earth = new THREE.Mesh(earthGeometry, earthMaterial);
scene.add(earth);
```

Fig 2.4.3 Example code for implementing a bump map in a MeshPhongMaterial()

Using the bump map texture from Fig 5.2.1, we can give our map texture some depth using the bumpMap and bumpScale properties.

2.4.4) CREATING ATMOSPHERE (CLOUDS, ETC)

```
//Clouds
var cloudGeometry = new THREE.SphereGeometry(10.3,  50, 50);
var cloudMaterial = new THREE.MeshPhongMaterial({
  map: new THREE.TextureLoader().load('img/clouds.jpg'),
  transparent: true,
  opacity: 0.3
});
var clouds = new THREE.Mesh(cloudGeometry, cloudMaterial);
scene.add(clouds);
```

Fig 2.4.4 Example cloud code

Using the same method of creating geometry and applying a texture, we can create an atmosphere by using a slightly larger sphere and making it transparent.

3)RESULTS AND DISCUSSION

3.1) HIDDEN IMAGE

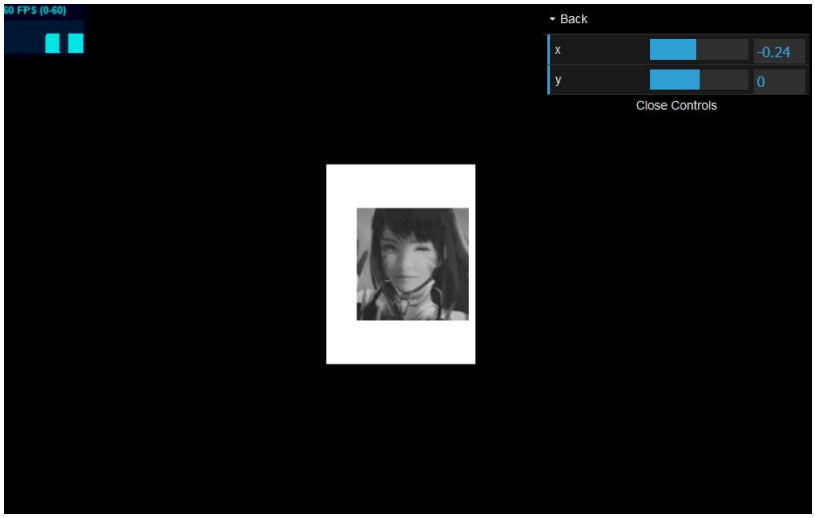
In the shader, first, we need to transform the image into gray_scale as the code below:

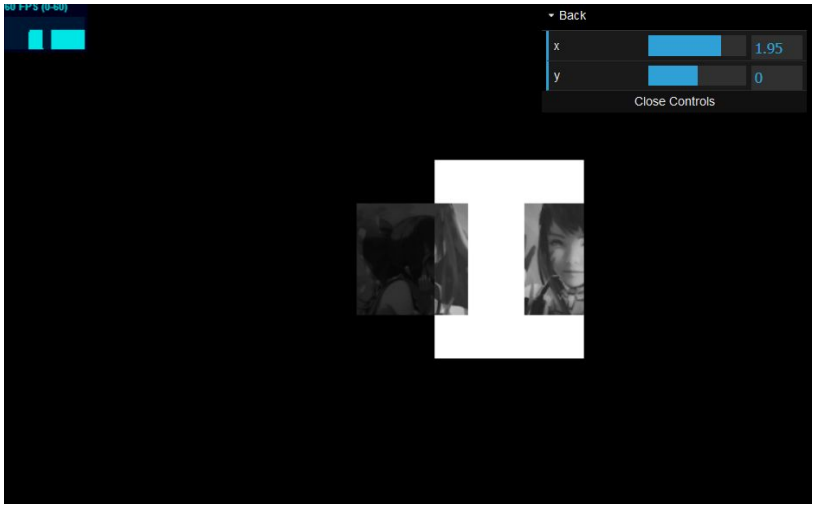
```
//turn to gray_scale
float grey_scale = dot(color.rgb, vec3(0.222, 0.707, 0.071));

color.r = grey_scale;
color.g = grey_scale;
color.b = grey_scale;
```

The implement of alpha blending is much simpler:

```
float a = 1.0 - color1.r + color2.r;
```

Then we combined two images and we can see the effect through three.js web.





3.2) COMBINED PLANET EFFECTS

Fig 5.5.1 final result using one earth and one moon

Using different textures and different sized spheres, you can easily repeat these steps to create and entire solar system of planets. Fig 5.5.1 gives an example with the earth and moon.

4) CONCLUSIONS

Bringing all the effects together into one scene was the biggest challenge of this project. Each person's implementation was slightly different so eliminating useless code and identifying the core pieces was a process. When combining effects we started one at a time, mixing the first two, then adding another, and finally bringing the fourth effect into the scene.

We decided to show the jump transition at the beginning of the scene and when the "jump is complete" you will surrounded by stars and planets and will look around at the hidden image illusion. Using the mouse and orbit controls, you can look around and see how the image changes depending on its background. Some parameters

A HEADINGS IN APPENDICES

Here is an outline of the body of this document in Appendix-appropriate form:

A.1 Introduction
A.2 Hidden Image
  A.2.1 Understanding the Effect
  A.2.2 Alpha Blending
A.3 Starfield (Jump Transition)
  A.3.1 Describing the Effect
A.4 Starfield (Particle System)
  A.4.1 Particle System
  A.4.2 Perlin Noise
A.5 Planets
  A.5.1 Creating Planets and Moons with Geometry
  A.5.2 Applying Textures
  A.5.3 Using a Bump Map
  A.5.4 Combined Results

ACKNOWLEDGMENTS

This work is created for CMPM 163 (Game Graphics), Professor Angus Forbes, UCSC.

REFERENCES