CS460, Fall 2022, Boston, MA

SHADER EFFECT FOR BRAIN SURFACE

Sairam Bandarupalli and Sruthi Chirumamilla
{s.bandarupalli001@umb.edu},{s.chirumamilla001@umb.edu}
University of Massachusetts Boston

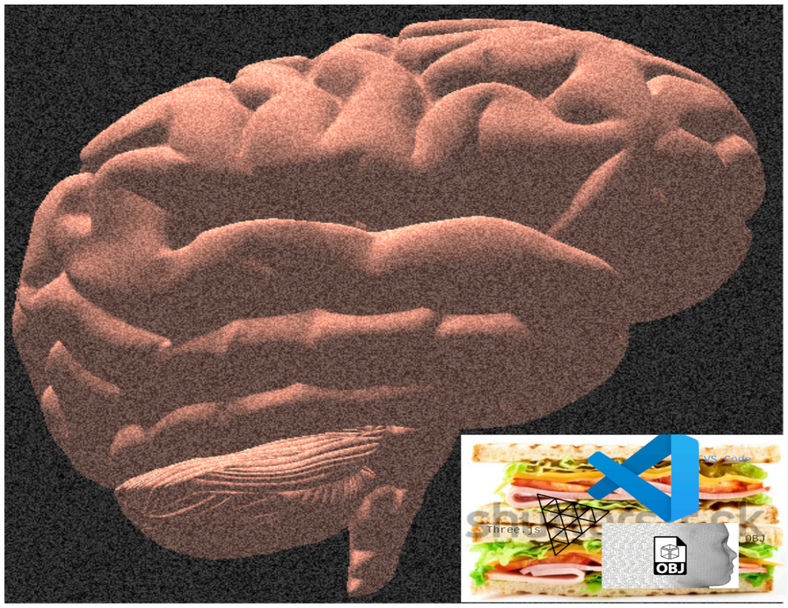


Figure 1: Final Output and Tech Stack

ABSTRACT

This project uses Three.js infrastructure to import a 3D brain model, apply film grain effects and draw the final output to HTML canvas. The model is an OBJ which is applied a **THREE.MeshStandardMaterial** and applied hex colour 0xF48E72 for a natural brain-like colour.

KEYWORDS

WebGL, Three.js, Post processing, Shader, film grain

1 INTRODUCTION

Post processing is adding effect to an existing to improve certain aesthetics such as lighting and presentation. This project uses a film grain post effect to be overlaid on an image of a brain to create a noisy effect.

Film grain is the natural process that causes particles to form on old school films due to components on the film such as silver halide not receiving enough photons. It is a technique used in movies to create the effect of amateur shots and in horror films as a source of cheap scares.

Using computer graphics, it is possible to emulate such an effect can be simulated using a random number generating function (or noise function). First, a scene is rendered to a texture using framebuffers or render targets depending on the graphics API. Next, sample the texture using the screen coordinates as UV's. If the effect was rendered, it would be as if there was no noise effect as shown here:

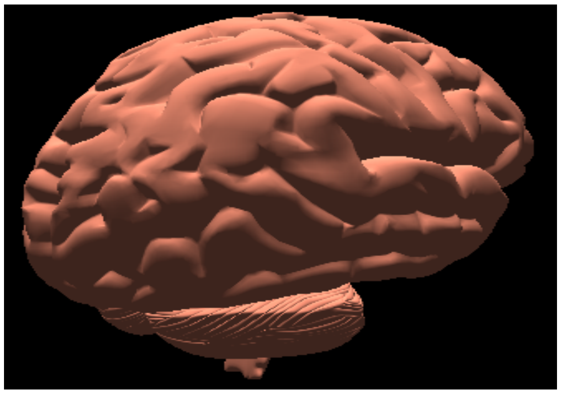


Figure 2: Plain boring no noise ☹

In order to get noise effects, we need to layer on top some noise onto the fragment. First, we need to self-multiply the UV's used to sample the fragment by a certain amount. This amount should be varied so as to get a unique sample every time we do so for the fragment. This can be achieved by using uniforms. The amount, will be the seed for our random number generator (or noise function)

Next, using the new resampled UV's, we resample once again to get the final noise value. This value might be too large hence we need to vary the intensity. Finally, the noise is going to be added to the r-g-b output for the fragment.

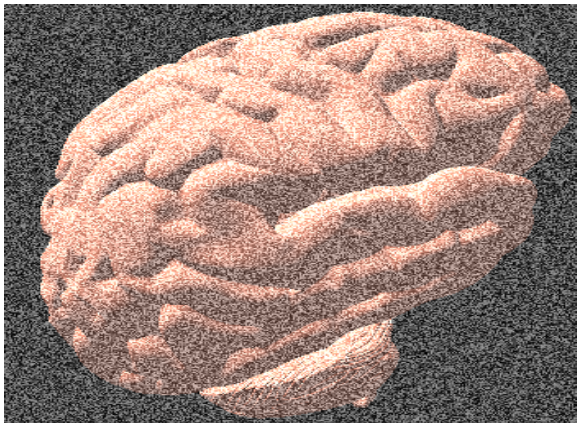
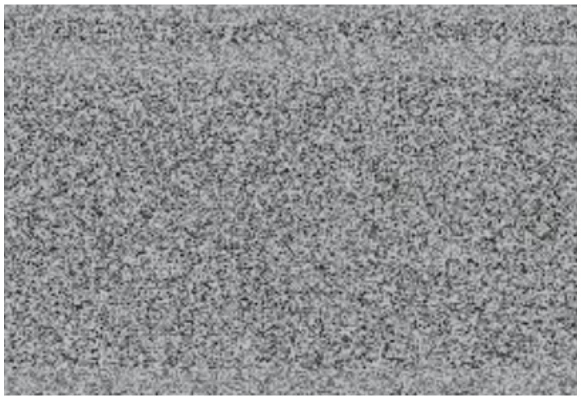


Figure 3: A lot of noise!

The final output once all fragments are rendered, resemble a noisy effect that was common with analogue television sets which produced white noise once the signal was lost.



2 RELATED WORKS

1. Three.js Github repository
2. Three.js Film Grain, Simon Harris
3. Three.js Online documentation

3 METHOD

Create Three.js scene and add lights, cameras and TrackballControls

```
    scene = new THREE.Scene()
    loader = new THREE.OBJLoader()
    composer = new
THREE.EffectComposer(renderer)
    scene.add(new THREE.AmbientLight(0x404040))
    light = new THREE.DirectionalLight( 0xffffff, 1.0 )
    light.castShadow = false
    camera = new THREE.PerspectiveCamera(60,
aspect, 0.01, 1000)
    controls = new
THREE.TrackballControls( camera,
renderer.domElement )
```

Add THREE.RenderPass to THREE.EffectComposer, add THREE.ShaderPass to THREE.EffectComposer

```
        renderPass = new
THREE.RenderPass(scene, camera)
        composer.addPass(renderPass)

        shaderPass = new
THREE.ShaderPass(effect)
        shaderPass.renderToScreen = true
        composer.addPass(shaderPass)
```

Render the scene with THREE.EffectComposer and Vary the THREE.ShaderPass noise uniform

```
        controls.update();
        counter += 0.01
        shaderPass.uniforms["amount"].value =
        counter;
        shaderPass.uniforms["intensity"].value =
        controller.intensity;

        requestAnimationFrame(animate)
        /* uncomment to disable effect */
        // renderer.render( scene, camera );
        composer.render()
```

3.1 Implementation

The code in the method section can be directly translated to a HTML webpage. There are several scripts that have to be included at the head of the HTML page. These scripts can be found in the Three.js Github repository [1]. Ensure that you have switched to the master branch then search for the following scripts in the example directory as of commit 0783e2b:

- controls/TrackballControls.js
- loader/OBJLoader.s
- postprocessing/EffectComposer.js
- postprocessing/Pass.js
- postprocessing/RenderPass.js
- postprocessing/ShaderPass.js
- shaders/CopyShader.js

Also ensure that you obtain build/three.js from the root directory of the repository to ensure the versions are not mixed.

Place the scripts in a deps directory or somewhere accessible then include them as external JavaScript code in your HTML file.

```
…
<script src="deps/three.js"></script>
…
```

Now, it becomes apparent there is need to separate the shader code, models and assets from the code to have a bit of the organization of the project. A good approach we used it to have the shaders and OBJ brain model to be separate files on the file system. This allows flexibility as shaders can be separately edited and the model can be easily swapped out but it does present a security challenge. How will the models be safely loaded by the browser? Remember a browser should **NOT** directly access the file system for the user as that would be breaching user privacy

A common approach, which we used, was to use client-server architecture. This model allows the assets to be stored on a server then they are retrieved by the browser. An XMLHttpRequest should do the trick but an even better approach is to use ES6 standard fetch [5] method. This requires the window.onload function to be an asynchronous function in that it will not block the other web components to prevent the rest of the page from loading. It's a really revolutionary feature of ES6 which allows creating responsive websites.

```
window.onload = async function(){}
```

Now this makes it possible to asynchronously fetch content over a network in a secure and simple manner.

```
await fetch("file.txt").then((res)=>
res.text()).then((data)=>{})
```

This new interface uses Promises, another ES6 feature which aims to make webpages faster and responsive using Service workers. The first promise (res.text()) converts the result body to text. The second promise is the used to access the converted data.
Now all that is left is to load the brain model using fetch, which is pretty straightforward.

```
loader.load('brain.obj', function(obj)
{
obj.children[0].material = new
THREE.MeshStandardMaterial({color:
0xF48E72})
scene.add(obj.children[0])
})
```

Also we access the OBJ children as manually set the material to a reddish color of 0xF48E72 to create a brain-like surface.

The following are some more snippets of code that are also included in the HTML file:

Code to add shader pass:

```
effect =
{
uniforms:{
"tDiffuse": {value: null},
"amount": {value: counter}
},
vertexShader: vshader,
fragmentShader: fshader
}
shaderPass = new
THREE.ShaderPass(effect)
shaderPass.renderToScreen = true
composer.addPass(shaderPass)
```

This is the full fragment shader used:

```
precision mediump float;
varying vec2 vUv;
uniform sampler2D tDiffuse;
uniform float amount;
float noise(vec2 p)
{
    vec2 k = vec2(23.147, 2.665); /* e*pi,
2^sqrt(2) */
    return fract( cos( dot(p, k) ) * 12345.678 );
}
void main(){
    vec4 color = texture2D(tDiffuse, vUv);
    vec2 uv_random = vUv;
    uv_random.y *= noise(vec2(uv_random.y,
amount));
    color.rgb += noise(uv_random) * 0.3;
    gl_FragColor = vec4(color);
}
```

3.2 Milestones

Development was structured in stages.

Milestone 3.2.1
Obtain a brain model from done3d.com/brain.

- Done3d.com allows free downloads of their models for non-commercial use

Milestone 3.2.2
Get a basic post effect working with Three.js EffectComposer

- Using a very basic shader to copy the contents of previous pass to the next (CopyShader.js)

Milestone 3.2.3
Getting documentation of film grain post effect

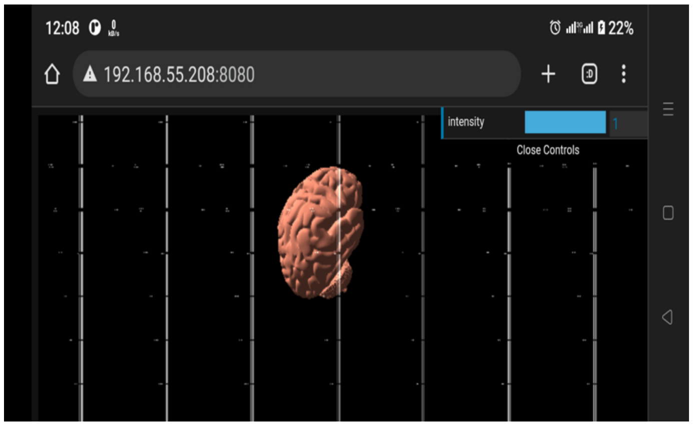
- We searched precariously the web for a tutorial. Luckily we found the Simon Harris blog [2] which had detailed instructions on how to render film grain on Three.js

Milestone 3.2.4
Finally getting EffectComposer to render the scene with RenderPass and ShaderPass passes

- This involved replacing the usual THREE.WebGLRenderer with THREE.EffectComposer as the renderer

3.3 Challenges

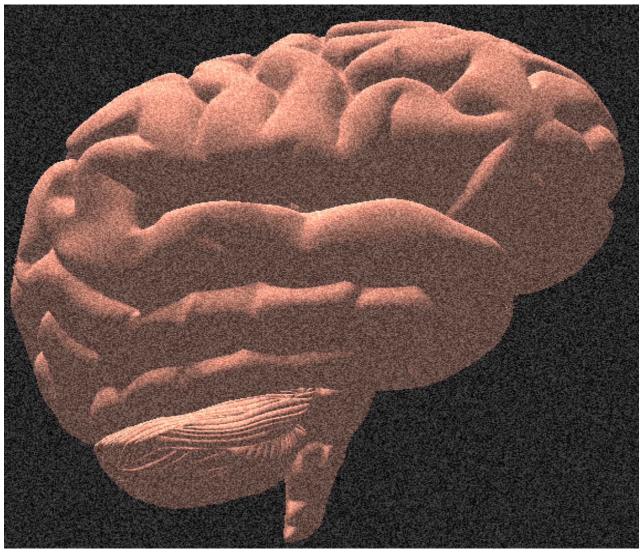
- It's quite the task to get a highly detailed free 3d brain model [4]. Most models either lacked fidelity or were accessible only through a premium subscription
- Finding documentation about post processing. We used a lot of the examples in the Three.js Github repository.
- Low end devices would have a hard time running the app. This was as a result of the very massive 3D brain model which is 28.7 MB. This requires a lot of video memory to load the vertices to the GPU. It is possible to reduce the vertices using Blender but that causes a huge loss in detail for the model
- Rendering bugs on certain devices (some mobile chips). This might be an issue with Chrome OpenGL ES renderer since the noise only rendered on the y-axis producing artefacts as shown here:



- Long load times for the brain model

4 RESULTS

This was the final rendered output with a noise intensity of 0.4; we get a very subtle noise effect which is very neat!



The table here shows a few devices and their respective graphics cards on which the application was tested on. Frame rate was measured using MSI Afterburner [6] and tabulated below:

Device – Graphics Card	Performance
Intel HD 2000	35 FPS
Power VR Rogue GE 8322	45 FPS
NVIDIA GTX 1080	100 FPS

We can see that an average desktop GPU can run the app at a decent frame rate and so can a mobile GPU. A gaming desktop clearly blows the whole competition and churns out a whopping 120 frames every second!

5 CONCLUSIONS

Three.js provides a very nice and simple abstraction of drawing 3D graphics in the web browser. This is a stark contrast to WebGL which requires a lot of code just to get a simple triangle to draw on the screen. Three.js takes care of setting up all the buffers, context creation, lighting, shading and even goes ahead and provides an interface to make post processing easier for the developer.

This goes to show that the web browser is a very versatile application that can not only allow us to read our emails and search the web, but also let us play games and run complex 3D simulations, even on low power devices such as mobile phones. Such is the power of modern computers.

6 REFERENCES

1. Ricardo Cabello, Three.js
   URL: https://github.com/mrdoob/three.js

2. Simon Harris et al. 2019
   URL: https://simonharris.co/making-a-noise-film-grain-post-processing-effect-from-scratch-in-threejs

3. Three.js Docs
   URL: https://threejs.org/docs/

4. Done3D, cgcreation Brain
   URL: https://done3d.com/brain

5. Fetch API MSDN
   URL: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

6. MSI Afterburner
   URL: https://www.msi.com/Landing/afterburner

GITHUB LINK:

https://sairam-bandarupalli.github.io/cs460student/CS460_Final_Project_Code/index.html