TREBALL FINAL DE GRAU

ANNEXOS

TÍTOL: IMPLEMENTACIÓ D'UN GENERADOR PROCEDURAL DE PLANETES

AUTORS: FERNÁNDEZ, BRIONES, ORIOL

DATA DE PRESENTACIÓ: JULIOL, 2022

SUMARI

Comentaris

- El codi està disponible a GitHub: https://github.com/orifer/TFG-procedural-generator
- Pes facilitar la lectura, els noms dels directoris estan indicats amb [D]

ESTRUCTURA DE FITXERS

```
Generador procedural:.
    .gitignore
    app.js
    index.html
    LICENSE
    README.md

    src
        css
            style.css

        js
            Atmosphere.js
            BaseApp.js
            Interface.js
            Planet.js
            Stars.js
            Sun.js

            maps
                TextureMap.js

        shaders
            BufferManager.js
            BufferShader.js
            vertexShader.js

            atmosphere
                atmosFrag.js

            heightMap
                heightMapFrag.js

            normalMap
                normalMapFrag.js

            starsShaders
                starsFrag.js
                starsVertex.js

            sunShaders
                sunFrag.js
                sunVertex.js

            textureMap
                circulationFrag.js
                commonFrag.js
                geologyFrag.js
                mainFrag.js
                soilFrag.js
                windFrag.js
```

[D] ARREL

APP.JS

```
import * as THREE from 'three';
import BaseApp from './src/js/BaseApp.js'
import Planet from './src/js/Planet.js'
import Interface from './src/js/Interface.js'
import Atmosphere from './src/js/Atmosphere.js'
import Sun from './src/js/Sun.js';
import Stars from './src/js/Stars.js';


export default class app extends BaseApp {

    constructor() {
        super();

        // Interface (GUI)
        this.interface = new Interface(this);

        this.time = 0;
        this.playing = false;

        this.stars = new Stars(this);

        super.postProcessing();
        this.render();
    }

    loadScene(props) {
        this.sceneId = props.sceneId;

        // Seed
        this.seed = props.seed;
        props.seed = (THREE.MathUtils.seededRandom(this.seed)  +
0.5); // Normalized seed between 0.5 and 1.5

        switch (this.sceneId) {
            case "0": this.loadScene0(props); break;
            case "1": this.loadScene1(props); break;
            case "2": this.loadScene2(props); break;
            default: break;
        }
    }

    // Earth-like
    loadScene0(props) {
        this.sun = new Sun(this);
        this.planet = new Planet(this, props);
        this.atmos = new Atmosphere(this, props);
```

```
        this.interface.init();
        this.interface.goToPlanet();
        this.interface.loadHistoryInterface();
    }

    // Water world
    loadScene1(props) {
        this.sun = new Sun(this);
        this.planet = new Planet(this, props);
        this.atmos = new Atmosphere(this, props);
        this.atmos.size = 0.04;
        this.atmos.waveLengths = new THREE.Vector3(700, 530,
440);
        this.playing = true;

        this.interface.init();
        this.interface.goToPlanetFast();
        this.interface.loadBasicInterface();
        this.planet.updatePlanetName("Water world");
    }

    // Red planet
    loadScene2(props) {
        this.sun = new Sun(this);
        this.planet = new Planet(this, props);
        this.atmos = new Atmosphere(this, props);
        this.atmos.size = 0.02;
        this.atmos.waveLengths = new THREE.Vector3(400, 500,
550);
        this.playing = true;

        this.interface.init();
        this.interface.goToPlanetFast();
        this.interface.loadBasicInterface();
        this.planet.updatePlanetName("Not Mars");
    }


    render() {
        super.render();

        if (this.playing) {
            this.time += 0.016;
        }

        if (this.sceneId) {
            this.planet.update();
            this.atmos.update();
            this.sun.update();
            this.stars.update();
            this.interface.update();
        }
```

```
    }

}

new app();
```

INDEX.HTML

```
<!DOCTYPE html>
<html lang="en">

    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width,
initial-scale=1.0"/>
        <title>Procedural Planet Generator</title>
        <link rel='stylesheet'
href='https://unpkg.com/swiper/swiper-bundle.min.css'>
        <link rel='stylesheet'
href='https://cdnjs.cloudflare.com/ajax/libs/bootstrap/5.0.2/css/b
ootstrap.min.css'>
        <link rel="stylesheet" href="src/css/style.css">
    </head>

    <script type="importmap">
        {
            "imports": {
                "three":
"https://unpkg.com/three@0.139/build/three.module.js"
            }
        }
    </script>

    <body>

        <!-- Main APP -->
        <div id="container"></div>
        <script type="module" src="app.js"></script>


        <!-- Left panel -->
        <div class="left-panel hide">
            <!-- Play/pause simulation button -->
            <button id="play-pause-button" class='btn-interface'>
                <i class='fa-solid fa-play'></i>
            </button>

            <!-- Terrain -->
            <div class="terrain-panel">
                <button id="add-terrain-button" class='btn-
interface btn-group' title='Pujar altura del terreny'>
                    <i class='fa-solid fa-arrows-up-to-line'></i>
                </button>
                <button id="del-terrain-button" class='btn-
interface btn-group' title='Baixar altura del terreny'>
                    <i class='fa-solid fa-arrows-down-to-
line'></i>
                </button>
            </div>
```

```
            <!-- View mode -->
            <div class="view-panel">
                <button id="normal-view-button" class='btn-
interface btn-group active' title='Vista normal'>
                    <i class='fa-solid fa-earth-europe'></i>
                </button>
                <button id="plates-view-button" class='btn-
interface btn-group' title='Plaques tectòniques'>
                    <i class='fa-solid fa-globe'></i>
                </button>
                <button id="rivers-view-button" class='btn-
interface btn-group' title='Rius'>
                    <i class='fa-solid fa-water'></i>
                </button>
                <button id="wind-view-button" class='btn-interface
btn-group' title='Clima'>
                    <i class='fa-solid fa-wind'></i>
                </button>
                <button id="temperature-view-button" class='btn-
interface btn-group' title='Temperatura'>
                    <i class='fa-solid fa-temperature-half'></i>
                </button>
            </div>
        </div>


        <!-- Right info -->
        <div id="left-info-panel" class="swiper swiper-container
swiper-container--timeline hide">
            <div class="swiper-wrapper">
                <!-- Slides -->
                <div class="swiper-slide">
                    <span class="info-title">4.600 Ma</span>
                    <div class="info-description">
                        <p>La Terra primerenca estava coberta per
un oceà de magma: una capa de roca fosa de centenars de
quilòmetres de profunditat. <br><br> Qualsevol aigua present només
existiria com a vapor d'aigua a l'atmosfera. <br><br> Durant
aquest període, va estar rebent l'impacte d'una gran quantitat
d'asteroides.</p>
                    </div>
                </div>
                <div class="swiper-slide">
                    <span class="info-title">4.000 Ma</span>
                    <div class="info-description">
                        <p>Durant els pròxims centenars de milions
d'anys, l'oceà de magma es va refredar prou per formar una
superfície sòlida.</p>
                    </div>
                </div>
                <div class="swiper-slide">
                    <span class="info-title">3.900 Ma</span>
                    <div class="info-description">
```

```
                        <p>A mesura que la Terra es va refredar,
es va formar una atmosfera principalment a partir dels gasos
emesos pels volcans. <br><br> La pressió superficial també era
molt més alta, gairebé cent vegades la d'avui i l'atmosfera era
molt més alta, a causa de la superfície encara calenta. <br><br>
Aquestes característiques la feien més semblant a l'atmosfera de
Venus actual que a la de la Terra actual.</p>
                    </div>
                </div>
                <div class="swiper-slide">
                    <span class="info-title">3.800 Ma</span>
                    <div class="info-description">
                        <p>Quan la temperatura va baixar dels 100
°C, el vapor d'aigua finalment va començar a condensar-se en
núvols gruixuts i van començar les pluges. <br><br> Aquesta pluja
va continuar durant milions d'anys, i finalment va formar el
primer oceà de la Terra. <br><br> Es creu que part d'aquesta aigua
va ser lliurada pels cometes i meteorits que s'estavellaven a la
superfície.</p>
                    </div>
                </div>
                <div class="swiper-slide">
                    <span class="info-title">3.000 Ma</span>
                    <div class="info-description">
                        <p>A la Terra, es creu que el moviment
tectònic va començar fa uns 3.000 milions d'anys. <br><br>
Aquestes plaques es mouen lentament per la superfície del planeta
durant centenars de milions d'anys. <br><br> El seu moviment era
molt més elevat que en l'actualitat, ara per ara oscil·la entre 0
i 10 cm anualment.</p>
                    </div>
                </div>
                <div class="swiper-slide">
                    <span class="info-title">500 Ma</span>
                    <div class="info-description">
                        <p>Durant els següents centenars de
milions d'anys, a mesura que el planeta va canviar, gran part del
CO2 de l'atmosfera es va dissoldre als oceans. <br><br> Així,
l'oxigen va començar a acumular-se a l'atmosfera, mentre que els
nivells de diòxid de carboni van continuar baixant.</p>
                    </div>
                </div>
                <div class="swiper-slide">
                    <span class="info-title">Actualitat</span>
                    <div class="info-description">
                        <p></p>
                    </div>
                </div>
            </div>
        </div>

        <!-- Stats -->
        <div id="stats-container" class="hide"></div>
```

```
        <!-- Bottom timeline -->
        <div id="bottom-timeline" class="swiper-container hide"
style="background: linear-gradient(to top, rgb(56, 56, 56),
rgba(34, 34, 34, 0.7), rgba(0, 0, 0, 0)); ">
            <div class="swiper-container-wrapper swiper-container-
wrapper--timeline">

                <!-- Timeline points -->
                <ul class="swiper-pagination-custom">
                    <li class='swiper-pagination-switch first
active'><span class='switch-title'>4.600 Ma</span></li>
                    <li class='swiper-pagination-switch'><span
class='switch-title'>4.000 Ma</span></li>
                    <li class='swiper-pagination-switch'><span
class='switch-title'>3.900 Ma</span></li>
                    <li class='swiper-pagination-switch'><span
class='switch-title'>3.800 Ma</span></li>
                    <li class='swiper-pagination-switch'><span
class='switch-title'>3.000 Ma</span></li>
                    <li class='swiper-pagination-switch'><span
class='switch-title'>500 Ma</span></li>
                    <li class='swiper-pagination-switch
last'><span class='switch-title'>Actualitat</span></li>
                </ul>

                <!-- Progressbar -->
                <div class="swiper-pagination swiper-pagination-
progressbar swiper-pagination-horizontal"></div>

                <!-- Navigation buttons -->
                <div class="swiper-button-next">
                    <p>Fes clic per avançar</p>
                    <i class='fa-solid fa-chevron-right'></i>
                </div>

            </div>
        </div>

        <!-- Start Dialog -->
        <div id="dialog-start" title="Generador procedural de
planetes">

            <!-- Select scene -->
            <legend style="margin-bottom: 0;">Selecciona una
escena</legend>
            <table  class="checkboxradio" id="scene-table"
style="width:100%">
                <tr>
                    <td>
                        <label style="width:100%" for="scene-
1">Earth-like</label>
                        <input value="0" type="radio" name="scene-
```

```
1" id="scene-1" checked>
                    </td>
                    <td>
                        <p style="font-size: 0.8rem; margin: 0;
margin-left: 10px;">Genera un planeta com la terra i visualitza la
seva evolució</p>
                    </td>
                </tr>
                <tr>
                    <td>
                        <label style="width:100%" for="scene-
2">Water world</label>
                        <input value="1" type="radio" name="scene-
1" id="scene-2">
                    </td>
                    <td>
                        <p style="font-size: 0.8rem; margin: 0;
margin-left: 10px;">Genera un planeta aquàtic</p>
                    </td>
                </tr>
                <tr>
                    <td>
                        <label style="width:100%" for="scene-
3">Red planet</label>
                        <input value="2" type="radio" name="scene-
1" id="scene-3">
                    </td>
                    <td>
                        <p style="font-size: 0.8rem; margin: 0;
margin-left: 10px;">Genera un planeta semblant a Mart</p>
                    </td>
                </tr>
            </table>

            <hr>

            <!-- Select resolution -->
            <fieldset class="checkboxradio">
                <legend style="margin-bottom:
0;">Resolució</legend>
                <p style="font-size: 0.8rem; margin: 0;"><span
class="ui-icon ui-icon-alert" style="float:left; margin:2px 2px
0px 0;"></span>Valors més grans tenen un major impacte al
rendiment.</p>
                <label for="radio-1">256</label>
                <input value="256" type="radio" name="radio-1"
id="radio-1">
                <label for="radio-2">512</label>
                <input value="512" type="radio" name="radio-1"
id="radio-2">
                <label for="radio-3">1024</label>
                <input value="1024" type="radio" name="radio-1"
id="radio-3" checked>
```

```
                <label for="radio-4">2048</label>
                <input value="2048" type="radio" name="radio-1"
id="radio-4">
                <label for="radio-5">4096</label>
                <input value="4096" type="radio" name="radio-1"
id="radio-5">
            </fieldset>

            <hr>

            <!-- Seed -->
            <fieldset>
                <legend style="margin-bottom: 0;">Seed</legend>
                <p style="font-size: 0.8rem; margin: 0;">Gestiona
la generació aleatòria.</p>
                <input type="text" name="seed-1" id="seed-1"
maxlength="10">

                <!-- Random button -->
                <button id="random-seed-button" class="button-
small" style="color: #3383BB; border: none;"><i class='fa-solid
fa-dice'></i></button>
            </fieldset>
        </div>

        <!-------------------------------------------------- Script
imports ------------------------------------------------------->
        <script src="https://kit.fontawesome.com/d56ac1debb.js"
crossorigin="anonymous"></script>           <!-- FontAwesome -->
        <script
src='https://cdnjs.cloudflare.com/ajax/libs/jquery/3.6.0/jquery.mi
n.js'></script>           <!-- jQuery -->
        <script
src='https://cdnjs.cloudflare.com/ajax/libs/jqueryui/1.12.1/jquery
-ui.min.js'></script>     <!-- jQuery UI -->
        <link rel="stylesheet"
href="https://code.jquery.com/ui/1.12.1/themes/overcast/jquery-
ui.css"/>   <!-- jQuery UI CSS -->
        <script
src='https://cdnjs.cloudflare.com/ajax/libs/Swiper/6.8.4/swiper-
bundle.min.js'></script>    <!-- Swiper -->
        <!-------------------------------------------------------
------------------------------------------------------------->
    </body>
</html>
```

[D] .SRC

[D] CSS

STYLE.CSS

```css
html,
body {
  position: relative;
  height: 100%;
  overscroll-behavior: none;
  overflow: hidden;
  background-color: rgb(0, 0, 0);
}
body {
  margin: 0;
  padding: 0;
}
#container {
  width: 100vw;
  height: 100vh;
}
canvas {
  display: block;
}
.hide {
  display: none;
}


/* Top right interface box */
#infoBoxHolder {
  position: absolute;
  z-index: 1000;
  top: 0px;
  right: 20px;
  color: #FFFFFF;
  background-color: rgba(0, 0, 0, 0.8);
  width: 245px;
}
#infoBox {
  padding: 12px 12px 12px 12px;
  font-size: 14px;
  font-family: "Helvetica-Light", sans-serif;
  line-height: 18px;
}
#planetName {
  font-size: 24px;
  line-height: 28px;
}


/* Buttons style  */
```

```
.btn-interface {
  position: absolute;
  background-color: rgba(41, 42, 45, 0.6);
  border: none;
  color: rgb(151, 151, 151);
  width: 50px;
  height: 50px;
  font-size: 17px;  /* Icon size */
  cursor: pointer;
  border-radius: 10px;
  justify-content: center;
  align-items: center;
}
.btn-interface:hover, .btn-interface.active { /* Darker background
on mouse-over */
  background-color: rgba(65, 105, 225, 0.603);
  color: rgb(255, 255, 255);
}
.btn-group {
  position: relative;
  width: 44px;
  height: 44px;
}


/* Left Panel */
.left-panel {
  position: absolute;
  margin: auto;
  left: 10px;
  text-align: center;
  border-radius: 10px;
}
.view-panel {
  position: absolute;
  bottom: 315px;
  border: 2px solid rgba(41, 42, 45, 0.6);
  padding: 2px;
  border-radius: 15px;
}
.terrain-panel {
  position: absolute;
  bottom: 190px;
  border: 2px solid rgba(41, 42, 45, 0.6);
  padding: 2px;
  border-radius: 15px;
}
#play-pause-button {
  bottom: 110px;
}


/* Bottom timeline panel */
```

```
.swiper-container {
  position: absolute;
  margin: auto;
  left: 0;
  right: 0;
  bottom: 0px;
  color: #FFFFFF;
  width: auto;
  height: 100px;
  text-align: center;
}
.swiper-container-wrapper--timeline .swiper-slide {
  position: absolute;
  top: 0;
  display: flex;
  /* background: rgba(201, 201, 201, 0.8); */
  min-height: 300px;
  align-items: center;
  justify-content: center;
  border-radius: 10px;
}
.swiper-container-wrapper--timeline .swiper-slide .container {
  padding: 0;
  width: 100%;
}
.swiper-container-wrapper--timeline .swiper-slide .title {
  font-size: 18px;
  opacity: 0;
  transition: 0.5s ease 0.5s;
}
.swiper-container-wrapper--timeline .swiper-slide-active .title {
  opacity: 1;
}
.swiper-container-wrapper--timeline .swiper-pagination-progressbar
{
  position: relative;
  margin-bottom: 70px;
  background-color: transparent;
  height: 4px;
  border-bottom: 1px solid #888;
  width: 75%;
}
.swiper-container-wrapper--timeline .swiper-pagination-
progressbar-fill {
  background-color: rgba(65, 105, 225, 0.603);
  height: 3px;
  top: 2px;
}
.swiper-container-wrapper--timeline .swiper-pagination-
progressbar:before {
  position: absolute;
  top: 2px;
  left: -100%;
```

```
  width: 100%;
  height: 3px;
  background-color: #000;
  content: "";
}
.swiper-container-wrapper--timeline .swiper-pagination-
progressbar:after {
  position: absolute;
  top: 3px;
  right: -15%;
  width: 300px;
  height: 1px;
  background: linear-gradient(to right, rgb(151, 151, 151) ,
rgba(0, 0, 0, 0));
  content: "";
  z-index: -1;
}
.swiper-container--timeline {
  background: rgba(41, 42, 45, 0.2);
  position: absolute;
  bottom: 150px;
  left: auto;
  right: 20px;
  width: 300px;
  height: 400px;
  border-radius: 10px;
}
.info-text {
  text-align: left;
}
.swiper-container-wrapper--timeline .swiper-pagination-custom {
  position: relative;
  list-style: none;
  margin: 1rem 0;
  padding: 0;
  display: flex;
  line-height: 1.66;
  bottom: 0;
  z-index: 11;
  width: 75%;
  display: flex;
}
.swiper-container-wrapper--timeline .swiper-pagination-custom
.swiper-pagination-switch {
  position: relative;
  width: 100%;
  height: 30px;
  line-height: 30px;
  display: block;
}
.swiper-container-wrapper--timeline .swiper-pagination-custom
.swiper-pagination-switch .switch-title {
  position: absolute;
```

```
  font-weight: 400;
  right: 0;
  transform: translateX(50%);
  transition: 0.2s all ease-in-out;
  transition-delay: 0s;
  cursor: pointer;
  z-index: 1;
}
.swiper-container-wrapper--timeline .swiper-pagination-custom
.swiper-pagination-switch .switch-title:after {
  position: absolute;
  top: calc(100% + 19px);
  right: 50%;
  transform: translateX(50%) translateY(-50%);
  width: 12px;
  height: 12px;
  background: rgb(65, 105, 225);
  border-radius: 2rem;
  content: "";
  transition: 0.2s all ease-in-out;
  transition-delay: 0s;
  z-index: 1;
}
.swiper-container-wrapper--timeline .swiper-pagination-custom
.swiper-pagination-switch.active .switch-title {
  font-weight: 400;
  transition-delay: 0.4s;
}
.swiper-container-wrapper--timeline .swiper-pagination-custom
.swiper-pagination-switch.active .switch-title:after {
  background: rgb(44, 96, 255);
  width: 25px;
  height: 25px;
  transition-delay: 0.4s;
}
.swiper-container-wrapper--timeline .swiper-pagination-custom
.swiper-pagination-switch.active ~ .swiper-pagination-switch
.switch-title {
  color: #888;
  font-weight: 16px;
}
.swiper-container-wrapper--timeline .swiper-pagination-custom
.swiper-pagination-switch.active ~ .swiper-pagination-switch
.switch-title:after {
  background: #888;
}
.swiper-button-next {
  white-space: nowrap;
  right: 100px;
  bottom: 22px;
  top: auto;
  justify-content: center;
  color: rgba(255, 255, 255);
```

```
}
.swiper-button-next p {
  margin-right: 10px;
  height: 28%;
}
```

[D] JS

ATMOSPHERE.JS

```
import * as THREE from 'three';
import fragmentShader from '../shaders/atmosphere/atmosFrag.js'
import vertexShader from '../shaders/vertexShader.js'


class Atmosphere {

  constructor(app) {

    // Main app
    this.app = app;
    this.view = new THREE.Object3D();

    // Atmosphere properties
    this.size = 0;
    this.densityFalloff = 4.;
    this.opticalDepthPoints = 12.;
    this.inScatterPoints = 12.;
    this.waveLengths = new THREE.Vector3(400, 500, 550);
    this.scatteringStrength = 64.;

    // Animation properties
    this.startTime = 10;
    this.endTime = 22;
    this.finalSize = 0.1;
    this.startWaveLengths = new THREE.Vector3(400, 500, 550);

    this.startTime2 = 22;
    this.endTime2 = 30;
    this.finalSize2 = 0.02;
    this.finalWaveLengths = new THREE.Vector3(700, 530, 440);

    this.createScene();
  }

  createScene() {

    this._target = new THREE.WebGLRenderTarget(window.innerWidth,
window.innerHeight);
    this._target.texture.minFilter = THREE.NearestFilter;
    this._target.texture.magFilter = THREE.NearestFilter;
    this._target.texture.generateMipmaps = false;
    this._target.stencilBuffer = false;
    this._target.depthBuffer = true;
    this._target.depthTexture = new THREE.DepthTexture();
    this._target.depthTexture.format = THREE.DepthFormat;
    this._target.depthTexture.type = THREE.FloatType;

    this._postCamera = new THREE.OrthographicCamera( - 1, 1, 1, -
1, 0, 1 );
```

```
    const uniforms = {
      cameraNear: { value: this.app.camera.near },
      cameraFar: { value: this.app.camera.far },
      cameraPosition: { value: this.app.camera.position },
      tDiffuse: { value: null },
      tDepth: { value: null },
      inverseProjection: { value: null },
      inverseView: { value: null },
      planetPosition: { value: new THREE.Vector3(0, 0, 0) },
      planetRadius: { value: this.app.planet.size },
      atmosphereRadius: { value: null },
      sunPosition: { value: this.app.sun.position },
      densityFalloff: { value: this.densityFalloff },
      opticalDepthPoints: { value: this.opticalDepthPoints },
      inScatterPoints: { value: this.inScatterPoints },
      scatteringCoefficients: { value: null },
    }

    this.material =  new THREE.ShaderMaterial({
      uniforms: uniforms,
      fragmentShader: fragmentShader,
      vertexShader: vertexShader,
      depthWrite: false,
    })

    this.postPlane = new THREE.PlaneBufferGeometry( 2, 2 );
    this.postQuad = new THREE.Mesh(this.postPlane, this.material);

    this._postScene = new THREE.Scene();
    this._postScene.add( this.postQuad );
  }

  render() {
    if (this.size) {

      // First, render the main scene
      renderer.setRenderTarget(this._target);
      renderer.render(this.app.scene, this.app.camera);
      renderer.setRenderTarget( null );


      // Update uniform values
      this.material.uniforms.tDiffuse.value =
this._target.texture;
      this.material.uniforms.tDepth.value =
this._target.depthTexture;

      this.material.uniforms.inverseProjection.value =
this.app.camera.projectionMatrixInverse;
      this.material.uniforms.inverseView.value =
this.app.camera.matrixWorld;
      this.material.uniforms.cameraPosition.value =
```

```
this.app.camera.position;

      this.material.uniforms.atmosphereRadius.value =
this.app.planet.size + this.size;
      this.material.uniforms.densityFalloff.value =
this.densityFalloff;
      this.material.uniforms.opticalDepthPoints.value =
this.opticalDepthPoints;
      this.material.uniforms.inScatterPoints.value =
this.inScatterPoints;

      var scatterR = Math.pow(400 / this.waveLengths.x, 4) *
this.scatteringStrength;
      var scatterG = Math.pow(400 / this.waveLengths.y, 4) *
this.scatteringStrength;
      var scatterB = Math.pow(400 / this.waveLengths.z, 4) *
this.scatteringStrength;
      var scatteringCoefficients = new THREE.Vector3(scatterR,
scatterG, scatterB);
      this.material.uniforms.scatteringCoefficients.value =
scatteringCoefficients;

      // Render
      renderer.render( this._postScene, this._postCamera );
    }
  }

  update() {
    if (this.app.playing) {

      // Increase the size of the atmosphere progressively.
Starting atmosphere, more thick and redish.
      if (this.app.time > this.startTime && this.app.time <
this.endTime) {
        this.size = 0.0001 +
(THREE.MathUtils.smoothstep((this.app.time-
this.startTime)/(this.endTime-this.startTime), 0., 1.) *
this.finalSize);
      }
      // Final atmosphere, more thin and blueish.
      else if (this.app.time > this.startTime2 && this.app.time <
this.endTime2) {
        this.size = this.finalSize -
((THREE.MathUtils.smoothstep((this.app.time-
this.startTime2)/(this.endTime2-this.startTime2), 0., 1.) *
(this.finalSize - this.finalSize2)));
        this.waveLengths.x = this.startWaveLengths.x -
((THREE.MathUtils.smoothstep((this.app.time-
this.startTime2)/(this.endTime2-this.startTime2), 0., 1.) *
(this.startWaveLengths.x - this.finalWaveLengths.x)));
        this.waveLengths.y = this.startWaveLengths.y -
((THREE.MathUtils.smoothstep((this.app.time-
this.startTime2)/(this.endTime2-this.startTime2), 0., 1.) *
```

```
(this.startWaveLengths.y - this.finalWaveLengths.y)));
        this.waveLengths.z = this.startWaveLengths.z -
((THREE.MathUtils.smoothstep((this.app.time-
this.startTime2)/(this.endTime2-this.startTime2), 0., 1.) *
(this.startWaveLengths.z - this.finalWaveLengths.z)));
      }
    }

    // Render the atmosphere if we are looking at the planet in
sphere mode
    if (this.app.planet.geo.type == 'SphereGeometry') {
      this.render();
    }
  }



}

export default Atmosphere;
```

BASEAPP.JS

```
import * as THREE from 'three';
import { OrbitControls } from
'https://unpkg.com/three@0.139/examples/jsm/controls/OrbitControls
.js';
import Stats from
"https://unpkg.com/three@0.139/examples/jsm/libs/stats.module";

// Browse effects here ->
https://unpkg.com/browse/three@0.139/examples/jsm/postprocessing/
import { EffectComposer  } from
'https://unpkg.com/three@0.139/examples/jsm/postprocessing/EffectC
omposer.js';
import { RenderPass } from
'https://unpkg.com/three@0.139/examples/jsm/postprocessing/RenderP
ass.js';
import { UnrealBloomPass } from
'https://unpkg.com/three@0.139/examples/jsm/postprocessing/UnrealB
loomPass.js';
import { SMAAPass } from
'https://unpkg.com/three@0.139/examples/jsm/postprocessing/SMAAPas
s.js';


class BaseApp {

    constructor() {

        // Camera
        const fov = 35;
        const aspect = window.innerWidth / window.innerHeight;
        const near = 0.1;
        const far = 999999.0;
        this.camera = new THREE.PerspectiveCamera(fov, aspect,
near, far);
        this.camera.position.z = 7;
        window.camera = this.camera;

        // Main scene
        this.scene = new THREE.Scene();

        // Renderer
        this.renderer = new THREE.WebGLRenderer({
            alpha: true,
            antialias: true
        });
        this.renderer.setPixelRatio( window.devicePixelRatio );
        this.renderer.setSize( window.innerWidth,
window.innerHeight );
        this.renderer.toneMapping = THREE.ACESFilmicToneMapping;
        // this.renderer.autoClear = false;
        this.renderer.setClearColor( 0x000000, 0 );
        window.renderer = this.renderer;
```

```
        // Add to html
        document.getElementById('container').appendChild(
this.renderer.domElement );

        // Stats
        this.stats = Stats()
        document.body.appendChild(this.stats.dom)

        // Lights
        this.ambientLight = new THREE.AmbientLight(0xffffff,
0.04);
        this.scene.add(this.ambientLight);

        this.directionalLight = new THREE.DirectionalLight(
0xffffff, 1.0 );
        // this.directionalLight.position.set( 1, 1, 0);
        this.scene.add(this.directionalLight);
        window.light = this.directionalLight;

        // Camera controls
        this.controls = new OrbitControls( this.camera,
this.renderer.domElement );
        this.controls.enableDamping = true;
        this.controls.dampingFactor = 0.05;
        this.controls.rotateSpeed = 0.5;
        this.controls.autoRotate = false
        this.controls.autoRotateSpeed = 2.0;
        this.controls.zoomSpeed = 0.2;

        // Mouse
        this.raycaster = new THREE.Raycaster();
        this.pointer = new THREE.Vector2();
        this.pointerLeft = false;
        this.pointerRight = false;

        // Events
        window.addEventListener( 'resize',
this.onWindowResize.bind(this));
        window.addEventListener( 'pointermove',
this.onPointerMove.bind(this));
        window.addEventListener( 'pointerup',
this.onPointerUp.bind(this));
        window.addEventListener( 'pointerdown',
this.onPointerDown.bind(this));
    }

    postProcessing() {
        this.composer = new EffectComposer( this.renderer );
        this.composer.addPass(  new RenderPass( this.scene,
this.camera ) );
        this.composer.addPass(  new UnrealBloomPass( new
THREE.Vector2( window.innerWidth, window.innerHeight ), 2.5, 1.,
```

```
0.6 ) );
        this.composer.addPass(  new SMAAPass( window.innerWidth,
window.innerHeight) );
        //
https://threejs.org/examples/webgl_postprocessing_unreal_bloom.htm
l
    }

    updatePostProcessing() {
        // Update post-processing bloom
        if (this.playing && this.time > 3 && this.time < 8) {
            this.composer.passes[1].strength = 2.5 -
(THREE.MathUtils.smoothstep((this.time-3.)/8., 0., 1.)*2.5);
            this.composer.passes[1].threshold = 0.6 +
THREE.MathUtils.smoothstep((this.time-3.)/8., 0., 1.)*0.4;
        }
    }

    render(timestamp) {
        requestAnimationFrame( this.render.bind(this) );
        this.controls.update();
        this.stats.update()
        this.updatePostProcessing();

        // Update the picking ray with the camera and pointer
position
        this.raycaster.setFromCamera( this.pointer, this.camera );

        // Render scene
        // this.renderer.render( this.scene, this.camera );
        this.composer.render();
    }

    onPointerMove( event ) {
        // calculate pointer position in normalized device
coordinates
        // (-1 to +1) for both components
        this.pointer.x = ( event.clientX / window.innerWidth ) * 2
- 1;
        this.pointer.y = - ( event.clientY / window.innerHeight )
* 2 + 1;
    }

    onPointerDown( event ) {
        // Left mouse button is down
        if (event.button === 0) this.mouseClick = true;
    }

    onPointerUp( event ) {
        // Left mouse button is up
        if (event.button === 0) this.mouseClick = false;
    }
```

```
    onWindowResize() {
        // Update camera
        this.camera.aspect = window.innerWidth /
window.innerHeight;
        this.camera.updateProjectionMatrix();

        // Update renderer
        this.renderer.setSize( window.innerWidth,
window.innerHeight );
    }

}

export default BaseApp;
```

INTERFACE.JS

```
import * as THREE from 'three';
import { GUI } from
'https://cdn.skypack.dev/three@0.136/examples/jsm/libs/lil-
gui.module.min.js';
import gsap from 'https://cdn.skypack.dev/gsap';
// INFO: https://lil-gui.georgealways.com/

class Interface {

    constructor(app) {
      this.app = app;      // Main app
      this.loadDialog();
    }

    init() {
      // Create the GUI
      window.gui = new GUI({ autoPlace: false });
    }

    loadDialog() {
      var that = this; // context

      $( function() { // Document ready

        // Dialog
        $( "#dialog-start" ).dialog({
          dialogClass: "no-close",
          resizable: false,
          height: "auto",
          width: 600,
          modal: false,
          buttons: {
            Ok: function() {
              // Save the values before closing the dialog
              var sceneId = $(':radio:checked', this)[0].value;
              var resolution = $(':radio:checked', this)[1].value;
              var seed = $('#seed-1', this)[0].value;

              that.app.loadScene({
                sceneId: sceneId,
                resolution: resolution,
                seed: seed
              });

              $( this ).dialog( "close" );
            }
          }
        });

        // Checkbox constructor parameters
        $( ".checkboxradio input" ).checkboxradio({
          icon: false
```

```
        });

        // Random button
        $( "#random-seed-button" ).click(function() {
          $('#seed-1').val(THREE.MathUtils.randInt(0,
9999999999));
        });
        $('#seed-1').val(THREE.MathUtils.randInt(0, 9999999999));
      });
    }

    loadHistoryInterface() {
      // Show elements
      $(".left-panel").removeClass("hide");
      $("#left-info-panel").removeClass("hide");
      $("#stats-container").removeClass("hide");
      $("#bottom-timeline").removeClass("hide");

      // Load the panels
      this.createRightPanel();
      this.createPlanetCategory();
      this.createAtmosphereCategory();
      this.createDebugCategory();
      this.createCameraCategory();
      this.createBottomPanel();
      this.createLeftPanel();

      // Seed
      window.gui.add(this.app, "seed").disable();

      // Time
      window.gui.add(this.app, "time", 0., 100.).listen();

      window.gui.close();
    }


    loadBasicInterface() {
      // Show elements
      $("#stats-container").removeClass("hide");

      // Load the panels
      this.createRightPanel();
      this.createPlanetCategory();
      this.createAtmosphereCategory();
      this.createDebugCategory();
      this.createCameraCategory();

      // Seed
      window.gui.add(this.app, "seed").disable();

      // Time
      window.gui.add(this.app, "time", 0., 100.).listen();
```

```
      window.gui.close();
    }


    createRightPanel() {

      // Main container
      let infoBoxHolder = document.createElement("div");
      infoBoxHolder.setAttribute("id", "infoBoxHolder");
      document.body.appendChild(infoBoxHolder);

      // Planet name
      let infoBox = document.createElement("div");
      infoBox.setAttribute("id", "infoBox");
      infoBox.innerHTML = "Planet<br><div id='planetName'>Earth-
like</div>";
      infoBoxHolder.appendChild(infoBox);

      // Open controls btn
      infoBoxHolder.appendChild(window.gui.domElement);
    }

    createPlanetCategory() {
      let matFolder = window.gui.addFolder('Planeta');

      matFolder.add(this.app.planet, "roughness", 0.0,
1.0).onChange(value => { this.app.planet.updateMaterial(); });
      matFolder.add(this.app.planet, "metalness", 0.0,
1.0).onChange(value => { this.app.planet.updateMaterial(); });
      matFolder.add(this.app.planet, "normalScale", -1.5,
1.5).listen().onChange(value => {
this.app.planet.updateMaterial(); });
      matFolder.add(this.app.planet, "displacementScale", -0.05,
0.1).listen().onChange(value => {
this.app.planet.updateMaterial(); });
      matFolder.add(this.app.planet, "subdivisions", 2, 512,
1).onChange(value => { this.app.planet.updateGeometry(); });

      matFolder.add(this.app.planet, "wireframe").onChange(value
=> { this.app.planet.updateMaterial(); });
      matFolder.add(this.app.planet, "rotate");

      matFolder.close();
    }


    createAtmosphereCategory() {
      let atmFolder = window.gui.addFolder('Atmosfera');

      atmFolder.add(this.app.atmos, "size", 0.0, 0.2).listen();
      atmFolder.add(this.app.atmos, "densityFalloff", 0., 64.0);
      atmFolder.add(this.app.atmos, "opticalDepthPoints", 0, 32,
```

```
1);
      atmFolder.add(this.app.atmos, "inScatterPoints", 0, 32, 1);
      atmFolder.add(this.app.atmos, "scatteringStrength", 0, 128);

      // Wavelengths
      let waveFolder = atmFolder.addFolder('Wavelengths (nm)');
      waveFolder.add(this.app.atmos.waveLengths, "x", 400.,
700.).name("Red").listen();
      waveFolder.add(this.app.atmos.waveLengths, "y", 400.,
700.).name("Green").listen();
      waveFolder.add(this.app.atmos.waveLengths, "z", 400.,
700.).name("Blue").listen();
      waveFolder.close();

      atmFolder.close();
    }


    createDebugCategory() {
      let debugFolder = window.gui.addFolder('Debug');
      debugFolder.add(this.app.planet, "displayMap",
["textureMap", "heightMap", "normalMap"]).onChange(value => {
this.app.planet.updateMaterial() });

      // Display button
      debugFolder.add(this.app.planet, "switchGeometry");

      debugFolder.close();
    }

    createCameraCategory() {
      let cameraFolder = window.gui.addFolder('Càmera');
      cameraFolder.add(this.app.controls,
"autoRotate").name('Rotar');
      cameraFolder.add(this.app.camera, "fov", 20,
120).name("FOV").onChange(value => {
this.app.camera.updateProjectionMatrix() });
      cameraFolder.close();
    }


    createBottomPanel() {

      // Timeline
        var swiper = new Swiper(".swiper", {
          autoHeight: true,
          speed: 8000,
          direction: "horizontal",
          navigation: {
            nextEl: ".swiper-button-next",
            prevEl: ".swiper-button-prev"
          },
          pagination: {
```

```
            el: ".swiper-pagination",
            type: "progressbar"
          },
          loop: false,
          effect: "slide",
          spaceBetween: 30,
          on: {
            init: function () {
              $(".swiper-pagination-custom .swiper-pagination-
switch").removeClass("active");
              $(".swiper-pagination-custom .swiper-pagination-
switch").eq(0).addClass("active");
            },
            slideChangeTransitionEnd: function () {
              $(".swiper-pagination-custom .swiper-pagination-
switch").removeClass("active");
              $(".swiper-pagination-custom .swiper-pagination-
switch").eq(swiper.realIndex).addClass("active");
            }
          }
        });

        $(".swiper-pagination-custom .swiper-pagination-
switch").click(function () {
          // swiper.slideTo($(this).index());
          // $(".swiper-pagination-custom .swiper-pagination-
switch").removeClass("active");
          // $(this).addClass("active");
        });

        // Add event listener for the continue button
        $(".swiper-button-next")[0].addEventListener("click",
this.onButtonNextClick.bind(this));

      }

    onButtonNextClick() {
      this.clickTime = this.app.time;
      this.app.playing = true;
      setTimeout(this.nextStep.bind(this), 8000);

      // Stars
      gsap.to(this.app.stars.view.rotation , {
        x: 0,
        y: this.app.stars.view.rotation.y - 4,
        z: 0 ,
        duration: 10,
        ease: "power2.inOut"
      });

      // Planet
      gsap.to(this.app.planet.ground.rotation , {
        x: 0,
```

```
        y: this.app.planet.ground.rotation.y + 8,
        z: 0 ,
        duration: 5,
        ease: "power1.inOut"
      });

      // Camera
      this.app.controls.autoRotate = true;
      setTimeout(this.stopCamera.bind(this), 5000);
    }

    stopCamera() {
      this.app.controls.autoRotate = false;
      this.app.controls.autoRotateSpeed = 2.0;

      // Camera to position
      gsap.to(this.app.camera.position , { x: 4.6, y: 1., z: 2.,
duration: 4, ease: "power3.inOut" });
    }

    // Restart values after animation step
    nextStep() {
      this.app.playing = false;
      this.app.planet.rotationSpeed = 0.0003;
    }


    // This is executed on every tick
    update() {
      var deltaTime = this.app.time - this.clickTime;

      // Create a normalFunction-like curve
      var duration = 4.;
      var smoothValue =
THREE.MathUtils.smoothstep(deltaTime/duration,0.,0.5) -
THREE.MathUtils.smoothstep(deltaTime/duration,0.5,1.)
      if (smoothValue)  this.app.controls.autoRotateSpeed =
smoothValue * -30.;
    }


    changeView(view) {
      this.normalViewButton.classList.remove("active");
      this.platesViewButton.classList.remove("active");
      this.riversViewButton.classList.remove("active");
      this.windViewButton.classList.remove("active");
      this.temperatureViewButton.classList.remove("active");
      this.app.planet.displayTextureMap = view;
      this.app.planet.renderScene();
    }


    createLeftPanel() {
```

```
      // View options
      this.normalViewButton = document.getElementById("normal-
view-button");
      this.normalViewButton.addEventListener("click", () => {
        this.changeView(0);
        this.normalViewButton.classList.toggle("active");
      });
      this.platesViewButton = document.getElementById("plates-
view-button");
      this.platesViewButton.addEventListener("click", () => {
        this.changeView(1);
        this.platesViewButton.classList.add("active");
      });
      this.riversViewButton = document.getElementById("rivers-
view-button");
      this.riversViewButton.addEventListener("click", () => {
        this.changeView(2);
        this.riversViewButton.classList.add("active");
      });
      this.windViewButton = document.getElementById("wind-view-
button");
      this.windViewButton.addEventListener("click", () => {
        this.changeView(3);
        this.windViewButton.classList.add("active");
      });
      this.temperatureViewButton =
document.getElementById("temperature-view-button");
      this.temperatureViewButton.addEventListener("click", () => {
        this.changeView(4);
        this.temperatureViewButton.classList.add("active");
      });


      // Add Terrain
      let addTerrainButton = document.getElementById("add-terrain-
button");
      this.app.addingTerrain = false;
      addTerrainButton.addEventListener("click", () => {
        addTerrainButton.classList.toggle("active");
        delTerrainButton.classList.remove("active");

        if (this.app.addingTerrain) {
          this.app.addingTerrain = false;
          this.app.controls.enabled = true;
        } else {
          this.app.addingTerrain = true;
          this.app.removingTerrain = false;
          this.app.controls.enabled = false;
        }
      });
      // Del Terrain
      let delTerrainButton = document.getElementById("del-terrain-
```

```
button");
      this.app.removingTerrain = false;
      delTerrainButton.addEventListener("click", () => {
        addTerrainButton.classList.remove("active");
        delTerrainButton.classList.toggle("active");

        if (this.app.removingTerrain) {
          this.app.removingTerrain = false;
          this.app.controls.enabled = true;
        } else {
          this.app.removingTerrain = true;
          this.app.addingTerrain = false;
          this.app.controls.enabled = false;
        }
      });


      // Play/stop simulation
      let playButton = document.getElementById("play-pause-
button");
      playButton.addEventListener("click", () => {
        playButton.classList.toggle("active");
        playButton.children[0].classList.toggle("fa-play");
        playButton.children[0].classList.toggle("fa-pause");
        this.app.playing = !this.app.playing;
      });
    }

    goToPlanet() {
      gsap.to(this.app.camera.position , { x: 4.6, y: 1., z: 2.,
duration: 8, ease: "power4.out" });
    }

    goToPlanetFast() {
      this.app.camera.position.set(4.6, 1., 2.);
    }

}

export default Interface;
```

PLANET.JS

```
import * as THREE from 'three';
import TextureMap from './maps/TextureMap.js'


class Planet {

  constructor(app, props) {

    // Main app
    this.app = app;
    this.view = new THREE.Object3D();

    // Planet properties
    this.resolution = props.resolution;
    this.seed = props.seed;
    this.scene = props.sceneId;
    this.subdivisions = 128;
    this.size = 1;
    this.rotationSpeed = 0.0003;

    // Material properties
    this.roughness = 0.8;
    this.metalness = 0.5;
    this.normalScale = 1.0;
    this.displacementScale = 0.002;
    this.wireframe = false;
    this.rotate = true;

    // Texture map to display
    this.displayTextureMap = 0;

    // Map to display
    this.displayMap = "textureMap";

    // Mouse position on planet
    this.mouseIntersectUV = new THREE.Vector2(0, 0);

    this.createScene();
    this.renderScene();
  }


  createScene() {
    this.textureMap = new TextureMap(this.app.renderer,
this.resolution);
    this.material = new THREE.MeshStandardMaterial();
    this.geo = new THREE.SphereGeometry( this.size,
this.subdivisions, this.subdivisions );
    this.ground = new THREE.Mesh(this.geo, this.material);

    // Add to main scene
    this.view.add(this.ground);
```

```
    this.app.scene.add(this.view);
  }


  renderScene() {
    this.updateNormalScaleForRes(this.resolution);

    this.textureMap.render({
      time: this.app.time,
      resolution: this.resolution,
      displayTextureMap: this.displayTextureMap,
      seed: this.seed,
      scene: this.scene,
      mouse: this.mouseIntersectUV,
      mouseClick: { value: false },
      addingTerrain: { value: false },
      removingTerrain: { value: false },
    });

    this.heightMap = this.textureMap.heightMapTexture;
    this.normalMap = this.textureMap.normalMapTexture;

    this.updateMaterial();
  }


  update() {
    if (this.rotate) {
      this.ground.rotation.y += this.rotationSpeed;
    }

    if (this.app.playing) {
      // Update shader
      this.textureMap.render({
        time: this.app.time,
        resolution: this.resolution,
        displayTextureMap: this.displayTextureMap,
        seed: this.seed,
        scene: this.scene,
        mouse: this.mouseIntersectUV,
        mouseClick: { value: this.app.mouseClick },
        addingTerrain: { value: this.app.addingTerrain },
        removingTerrain: { value: this.app.removingTerrain },
      });

      // Calculate objects intersecting the picking ray via
raycasting
      const intersects = this.app.raycaster.intersectObjects(
this.app.scene.children );
      for ( let i = 0; i < intersects.length; i ++ ) {
        if (intersects[i].object == this.ground) {
          this.mouseIntersectUV = intersects[i].uv;
        }
```

```
      }
    }
  }


  updateMaterial() {
    this.material.roughness = this.roughness;
    this.material.metalness = this.metalness;

    if (this.wireframe) {
      this.material.wireframe = true;
    } else {
      this.material.wireframe = false;
    }

    if (this.displayMap == "textureMap") {
      this.material.map = this.textureMap.texture;
      this.material.displacementMap = this.heightMap;
      this.material.displacementScale = this.displacementScale;

      this.material.normalMap = this.normalMap;
      this.material.normalScale = new
THREE.Vector2(this.normalScale, this.normalScale);
    }
    else if (this.displayMap == "heightMap") {
      this.material.map = this.heightMap;
      this.material.displacementMap = null;
      this.material.normalMap = null;
    }
    else if (this.displayMap == "normalMap") {
      this.material.map = this.normalMap;
      this.material.displacementMap = null;
      this.material.normalMap = null;
    }

    this.material.needsUpdate = true;
  }

  updatePlanetName(name) {
    let planetName = document.getElementById("planetName");
    if (planetName != null) {
      planetName.innerHTML = name;
    }
  }


  switchGeometry() {
    if (this.geo.type == 'SphereGeometry') {
      this.geo = new THREE.PlaneGeometry( 6, 3 );
      this.app.ambientLight.intensity = 1.6
      this.app.directionalLight.intensity = 0.
      this.rotate = false
    } else if (this.geo.type == 'PlaneGeometry') {
```

```
      this.geo = new THREE.SphereGeometry( this.size,
this.subdivisions, this.subdivisions );
      this.app.ambientLight.intensity = 0.04
      this.app.directionalLight.intensity = 1.
    }

    this.ground.geometry.dispose()
    this.ground.geometry = this.geo
  }


  updateGeometry() {
    this.geo = new THREE.SphereGeometry( this.size,
this.subdivisions, this.subdivisions );
    this.ground.geometry.dispose()
    this.ground.geometry = this.geo
  }


  updateNormalScaleForRes(value) {
    this.normalScale = value * 0.0003;
  }

} export default Planet;
```

STARS.JS

```
import * as THREE from 'three';
import vertShader from '../shaders/starsShaders/starsVertex.js'
import fragShader from '../shaders/starsShaders/starsFrag.js'

class Stars {

  constructor(app) {

    // Main app
    this.app = app;
    this.view = new THREE.Object3D();

    this.createScene();
  }
  createScene() {
    var geometry = new THREE.SphereBufferGeometry(600000, 32, 32);
    var material = new THREE.ShaderMaterial({
      uniforms: {
        u_resolution: { value: { x: window.innerWidth, y:
window.innerHeight } },
      },
      side: THREE.BackSide,
      vertexShader: vertShader,
      fragmentShader: fragShader
    });

    var mesh = new THREE.Mesh(geometry, material);
    this.view.add(mesh);

    this.app.scene.add(this.view);
  }

  update() {

  }

}

export default Stars;
```

SUN.JS

```
import * as THREE from 'three';
import vertShader from '../shaders/sunShaders/sunVertex.js'
import fragShader from '../shaders/sunShaders/sunFrag.js'

class Sun {

  constructor(app) {

    // Main app
    this.app = app;
    this.view = new THREE.Object3D();

    this.size = 8000;
    this.position = new THREE.Vector3(200000, 0, 0);
    this.app.directionalLight.position.copy(this.position);

    this.createScene();
  }

  createScene() {

    // Sun
    const geometryCorona = new
THREE.SphereBufferGeometry(this.size, 16, 16);
    const materialCorona = new THREE.ShaderMaterial({
      side: THREE.BackSide,
      uniforms: {
        uSize: {value: this.size },
        uCenter: {value: this.position }
      },
      vertexShader: vertShader,
      fragmentShader: fragShader,
      transparent: true
    });

    this.coronaMesh = new THREE.Mesh(geometryCorona,
materialCorona);
    this.coronaMesh.position.copy(this.position);
    this.view.add(this.coronaMesh);

    this.app.scene.add(this.view);
  }

  update() {
  }

}

export default Sun;
```

[D] MAPS

TEXTUREMAP.JS

```
import * as THREE from 'three';

import BufferManager from '../../shaders/BufferManager.js';
import BufferShader from '../../shaders/BufferShader.js';

import BUFFER_MAIN_FRAG from
'../../shaders/textureMap/mainFrag.js';
import BUFFER_GEO_FRAG from
'../../shaders/textureMap/geologyFrag.js'
import BUFFER_CIRCULATION_FRAG from
'../../shaders/textureMap/circulationFrag.js'
import BUFFER_WIND_FRAG from
'../../shaders/textureMap/windFrag.js'
import BUFFER_SOIL_FRAG from
'../../shaders/textureMap/soilFrag.js'
import BUFFER_HEIGHTMAP_FRAG from
'../../shaders/heightMap/heightMapFrag.js';
import BUFFER_NORMALMAP_FRAG from
'../../shaders/normalMap/normalMapFrag.js';

import VERT from '../../shaders/vertexShader.js'


class TextureMap {

  constructor(renderer, resolution) {
    this.renderer = renderer;
    this.resolution = resolution;
    this.setup();
  }

  setup() {
    this.counter = 0;
    this.resolutionVector = new THREE.Vector3(this.resolution,
this.resolution, window.devicePixelRatio);
    this.orthoCamera = new THREE.OrthographicCamera(-1, 1, 1, -1,
-1, 1)

    // Targets
    this.targetMain = new BufferManager(this.renderer, { width:
this.resolution, height: this.resolution });
    this.targetGeo = new BufferManager(this.renderer, { width:
this.resolution, height: this.resolution });
    this.targetCirculation = new BufferManager(this.renderer, {
width: this.resolution, height: this.resolution });
    this.targetWind = new BufferManager(this.renderer, { width:
this.resolution, height: this.resolution });
    this.targetSoil = new BufferManager(this.renderer, { width:
this.resolution, height: this.resolution });
```

```
    this.targetHeightMap = new BufferManager(this.renderer, {
width: this.resolution, height: this.resolution });
    this.targetNormalMap = new BufferManager(this.renderer, {
width: this.resolution, height: this.resolution });


    // Main buffer
    this.bufferMain = new BufferShader(
      VERT,
      BUFFER_MAIN_FRAG,
      {
        uTime: { value: 0 },
        uFrame: { value: 0 },
        uResolution: { value: this.resolutionVector },
        uSeed: { value: 0 },
        uScene: { value: 0 },
        uDisplayTextureMap: { value: 0 },
        iChannel0: { value: null },
        iChannel1: { value: null },
        iChannel2: { value: null },
        iChannel3: { value: null },
      });


    // Geo buffer
    this.bufferGeo = new BufferShader(
      VERT,
      BUFFER_GEO_FRAG,
      {
        uTime: { value: 0 },
        uFrame: { value: 0 },
        uResolution: { value: this.resolutionVector },
        uSeed: { value: 0 },
        uScene: { value: 0 },
        uMouse: { value: new THREE.Vector2(0, 0) },
        uMouseClick: { value: false },
        uAddingTerrain: { value: false },
        uRemovingTerrain: { value: false },
        iChannel0: { value: null },
      });


    // Circulation buffer
    this.bufferCirculation = new BufferShader(
      VERT,
      BUFFER_CIRCULATION_FRAG,
      {
        uTime: { value: 0 },
        uFrame: { value: 0 },
        uResolution: { value: this.resolutionVector },
        uSeed: { value: 0 },
        uScene: { value: 0 },
        iChannel0: { value: null },
```

```
        iChannel1: { value: null },
      });


    // Wind buffer
    this.bufferWind = new BufferShader(
      VERT,
      BUFFER_WIND_FRAG,
      {
        uTime: { value: 0 },
        uFrame: { value: 0 },
        uResolution: { value: this.resolutionVector },
        uSeed: { value: 0 },
        uScene: { value: 0 },
        iChannel0: { value: null },
        iChannel1: { value: null },
        iChannel2: { value: null },
        iChannel3: { value: null },
      });


    // Soil buffer
    this.bufferSoil = new BufferShader(
      VERT,
      BUFFER_SOIL_FRAG,
      {
        uTime: { value: 0 },
        uFrame: { value: 0 },
        uResolution: { value: this.resolutionVector },
        uSeed: { value: 0 },
        uScene: { value: 0 },
        iChannel0: { value: null },
        iChannel1: { value: null },
        iChannel2: { value: null },
        iChannel3: { value: null },
      });


    // HeightMap buffer
    this.bufferHeightMap = new BufferShader(
      VERT,
      BUFFER_HEIGHTMAP_FRAG,
      {
        uTime: { value: 0 },
        uFrame: { value: 0 },
        uResolution: { value: this.resolutionVector },
        iChannel0: { value: null }
      });


    // NormalMap buffer
    this.bufferNormalMap = new BufferShader(
      VERT,
```

```
      BUFFER_NORMALMAP_FRAG,
      {
        uResolution: { value: this.resolutionVector },
        uHeightMap: { value: null }
      });

  }


  render(props) {

    // Geo buffer
    this.bufferGeo.uniforms.uTime.value = props.time;
    this.bufferGeo.uniforms.uFrame.value = this.counter;
    this.bufferGeo.uniforms.uResolution.value = new
THREE.Vector3(props.resolution, props.resolution,
window.devicePixelRatio);
    this.bufferGeo.uniforms.uSeed.value = props.seed;
    this.bufferGeo.uniforms.uScene.value = props.scene;
    this.bufferGeo.uniforms.iChannel0.value =
this.targetGeo.readBuffer.texture;
    this.bufferGeo.uniforms.uMouse.value = props.mouse;
    this.bufferGeo.uniforms.uMouseClick.value =
props.mouseClick.value;
    this.bufferGeo.uniforms.uAddingTerrain.value =
props.addingTerrain.value;
    this.bufferGeo.uniforms.uRemovingTerrain.value =
props.removingTerrain.value;
    this.targetGeo.render(this.bufferGeo.scene, this.orthoCamera);

    // Circulation buffer
    this.bufferCirculation.uniforms.uTime.value = props.time;
    this.bufferCirculation.uniforms.uFrame.value = this.counter;
    this.bufferCirculation.uniforms.uResolution.value = new
THREE.Vector3(props.resolution, props.resolution,
window.devicePixelRatio);
    this.bufferCirculation.uniforms.uSeed.value = props.seed;
    this.bufferCirculation.uniforms.uScene.value = props.scene;
    this.bufferCirculation.uniforms.iChannel0.value =
this.targetGeo.readBuffer.texture;
    this.bufferCirculation.uniforms.iChannel1.value =
this.targetCirculation.readBuffer.texture;
    this.targetCirculation.render(this.bufferCirculation.scene,
this.orthoCamera);

    // Wind buffer
    this.bufferWind.uniforms.uTime.value = props.time;
    this.bufferWind.uniforms.uFrame.value = this.counter;
    this.bufferWind.uniforms.uResolution.value = new
THREE.Vector3(props.resolution, props.resolution,
window.devicePixelRatio);
    this.bufferWind.uniforms.uSeed.value = props.seed;
    this.bufferWind.uniforms.uScene.value = props.scene;
```

```
    this.bufferWind.uniforms.iChannel0.value =
this.targetGeo.readBuffer.texture;
    this.bufferWind.uniforms.iChannel1.value =
this.targetCirculation.readBuffer.texture;
    this.bufferWind.uniforms.iChannel2.value =
this.targetWind.readBuffer.texture;
    this.targetWind.render(this.bufferWind.scene,
this.orthoCamera);

    // Soil buffer
    this.bufferSoil.uniforms.uTime.value = props.time;
    this.bufferSoil.uniforms.uFrame.value = this.counter;
    this.bufferSoil.uniforms.uResolution.value = new
THREE.Vector3(props.resolution, props.resolution,
window.devicePixelRatio);
    this.bufferSoil.uniforms.uSeed.value = props.seed;
    this.bufferSoil.uniforms.uScene.value = props.scene;
    this.bufferSoil.uniforms.iChannel0.value =
this.targetGeo.readBuffer.texture;
    this.bufferSoil.uniforms.iChannel1.value =
this.targetCirculation.readBuffer.texture;
    this.bufferSoil.uniforms.iChannel2.value =
this.targetWind.readBuffer.texture;
    this.bufferSoil.uniforms.iChannel3.value =
this.targetSoil.readBuffer.texture;
    this.targetSoil.render(this.bufferSoil.scene,
this.orthoCamera);

    // Main buffer
    this.bufferMain.uniforms.uTime.value = props.time;
    this.bufferMain.uniforms.uFrame.value = this.counter;
    this.bufferMain.uniforms.uResolution.value = new
THREE.Vector3(props.resolution, props.resolution,
window.devicePixelRatio);
    this.bufferMain.uniforms.uSeed.value = props.seed;
    this.bufferMain.uniforms.uScene.value = props.scene;
    this.bufferMain.uniforms.uDisplayTextureMap.value =
props.displayTextureMap;
    this.bufferMain.uniforms.iChannel0.value =
this.targetGeo.readBuffer.texture;
    this.bufferMain.uniforms.iChannel1.value =
this.targetCirculation.readBuffer.texture;
    this.bufferMain.uniforms.iChannel2.value =
this.targetWind.readBuffer.texture;
    this.bufferMain.uniforms.iChannel3.value =
this.targetSoil.readBuffer.texture;
    this.targetMain.render(this.bufferMain.scene,
this.orthoCamera);

    // HeightMap buffer
    this.bufferHeightMap.uniforms.uTime.value = props.time;
    this.bufferHeightMap.uniforms.uResolution.value = new
THREE.Vector3(props.resolution, props.resolution,
```

```
window.devicePixelRatio);
    this.bufferHeightMap.uniforms.iChannel0.value =
this.targetGeo.readBuffer.texture;
    this.targetHeightMap.render(this.bufferHeightMap.scene,
this.orthoCamera);

    // NormalMap buffer
    this.bufferNormalMap.uniforms.uResolution.value = new
THREE.Vector3(props.resolution, props.resolution,
window.devicePixelRatio);
    this.bufferNormalMap.uniforms.uHeightMap.value =
this.targetHeightMap.readBuffer.texture;
    this.targetNormalMap.render(this.bufferNormalMap.scene,
this.orthoCamera);

    // Save the result textures
    this.texture = this.targetMain.readBuffer.texture;
    this.heightMapTexture =
this.targetHeightMap.readBuffer.texture;
    this.normalMapTexture =
this.targetNormalMap.readBuffer.texture;

    // ToDo: Unify shaders and use just one time measurement
    this.counter = props.time*60.0;
  }


} export default TextureMap;
```

[D] shaders

BUFFERMANAGER.JS

```
import * as THREE from 'three';

class BufferManager {

    constructor(renderer = THREE.WebGLRenderer, {width, height}) {
        this.readBuffer = new THREE.WebGLRenderTarget(width,
height, {
            minFilter: THREE.LinearFilter,
            magFilter: THREE.LinearFilter,
            format: THREE.RGBAFormat,
            type: THREE.FloatType,
            stencilBuffer: false
        })

        this.writeBuffer = this.readBuffer.clone()
    }

    swap() {
        const temp = this.readBuffer
        this.readBuffer = this.writeBuffer
        this.writeBuffer = temp
    }

    render(scene, camera, toScreen = false) {
        renderer.setRenderTarget(this.writeBuffer);
        if (toScreen) {
            renderer.render(scene, camera);
        } else {
            renderer.render(scene, camera, this.writeBuffer,
true);
        }
        renderer.setRenderTarget(null);
        this.swap()
    }

    dispose() {
        this.readBuffer.dispose();
        this.writeBuffer.dispose();
    }

}

export default BufferManager;
```

BUFFERSHADER.JS

```
import * as THREE from 'three';

class BufferShader {

    constructor(vertexShader, fragmentShader, uniforms = {}) {
        this.uniforms = uniforms;
        this.material = new THREE.ShaderMaterial({
            vertexShader,
            fragmentShader,
            uniforms
        })
        this.scene = new THREE.Scene();
        this.scene.add(new THREE.Mesh(new
THREE.PlaneBufferGeometry(2, 2), this.material));
    }

}

export default BufferShader;
```

VERTEXSHADER.JS

```
// https://threejs.org/docs/#api/en/renderers/webgl/WebGLProgram
const vertexShader = /* glsl */ `


varying vec2 vUv; // 2d Vertex position
varying vec3 vPosition; // Vertex position

void main() {

    // 2d Vertex position
    vUv = uv;

    // 3d Vertex position
    vPosition = position;

    gl_Position = projectionMatrix * modelViewMatrix * vec4(
position, 1.0 );
}



`;
export default vertexShader;
```

[D] ATMOSPHERE

ATMOSFRAG.JS

```
// https://threejs.org/docs/#api/en/renderers/webgl/WebGLProgram
// Based on SimonDev and Sebastian Lague's work

const atmosFrag = /* glsl */ `


//
########################################################################
// ||                    VARIABLES &
QUALIFIERS                    ||
//
########################################################################


    #define FLT_MAX 3.402823466e+38

    ////////////////////////////////////////////////////////////////
////////////////

    in vec2 vUv;

    ////////////////////////////////////////////////////////////////
////////////////

    uniform sampler2D tDiffuse;
    uniform sampler2D tDepth;

    uniform mat4 inverseProjection;
    uniform mat4 inverseView;

    uniform vec3 sunPosition;

    uniform vec3 planetPosition;
    uniform float planetRadius;

    uniform float atmosphereRadius;
    uniform float densityFalloff;
    uniform float opticalDepthPoints;
    uniform float inScatterPoints;
    uniform vec3  scatteringCoefficients;


//
########################################################################
//
||                          FUNCTIONS                             ||
//
########################################################################
```

```
    // Convert screen coordinates to world coordinates
    vec3 _ScreenToWorld(vec3 pos) {
        vec4 posP = vec4(pos.xyz * 2. - 1.0, 1.);
        vec4 posVS = inverseProjection * posP;
        vec4 posWS = inverseView * vec4((posVS.xyz / posVS.w),
1.0);

        return posWS.xyz;
    }


    // Returns vector (dstToSphere, dstThroughSphere)
    // If ray origin is inside sphere, dstToSphere = 0
    // If ray misses sphere, dstToSphere = maxValue;
dstThroughSphere = 0
    vec2 raySphere(vec3 sphereCentre, float sphereRadius, vec3
rayOrigin, vec3 rayDir) {
        vec3 offset = rayOrigin - sphereCentre;
        float a = 1.0;
        // float a = dot(rayDir, rayDir); // If rayDir is not
normalized
        float b = 2.0 * dot(offset, rayDir);
        float c = dot(offset, offset) - sphereRadius *
sphereRadius;
        float d = b * b - 4.0 * a * c;

        // Number of intersections: 0 when d < 0, 1 when d = 0, 2
when d > 0
        if (d > 0.0) {
            float s = sqrt(d);
            float dstToSphereNear = max(0.0, (-b - s) / (2.0 *
a));
            float dstToSphereFar = (-b + s) / (2.0 * a);

            // Ignore intersections behind the ray
            if (dstToSphereFar >= 0.0) {
                return vec2(dstToSphereNear, dstToSphereFar -
dstToSphereNear);
            }
        }

        // Ray did not intersect the sphere
        return vec2(FLT_MAX, 0.0);
    }


    float densityAtPoint(vec3 densitySamplePoint) {
        float heightAboveSurface = length(densitySamplePoint -
planetPosition) - planetRadius;
        float height01 = heightAboveSurface / (atmosphereRadius -
planetRadius);
        float localDensity = exp(-height01 * densityFalloff) *
```

```
(1.0 - height01);

        return localDensity;
    }


    // Calculates the average density of the atmosphere along a
ray
    float opticalDepth(vec3 rayOrigin, vec3 rayDir, float
rayLength) {
        vec3 densitySamplePoint = rayOrigin;
        float stepSize = rayLength / (opticalDepthPoints -1.0);
        float opticalDepth = 0.0;

        for (float i = 0.; i < opticalDepthPoints; i++) {
            float localDensity =
densityAtPoint(densitySamplePoint);
            opticalDepth += localDensity * stepSize;
            densitySamplePoint += rayDir * stepSize;
        }

        return opticalDepth;
    }


    vec3 calculateLight(vec3 rayOrigin, vec3 rayDir, float
rayLength, vec3 originalCol) {
        vec3 dirToSun = normalize(sunPosition);
        vec3 inScatterPoint = rayOrigin;
        float stepSize = rayLength / (inScatterPoints - 1.0);
        vec3 inScatteredLight = vec3(0.0);
        float viewRayOpticalDepth = 0.0;

        for (float i = 0.; i < inScatterPoints; i++) {
            // Ray to atmosphere
            float sunRayLength = raySphere(planetPosition,
atmosphereRadius, inScatterPoint, dirToSun).y;
            float sunRayOpticalDepth =
opticalDepth(inScatterPoint, dirToSun, sunRayLength);
            viewRayOpticalDepth = opticalDepth(inScatterPoint, -
rayDir, stepSize * i);
            vec3 transmittance = exp(-(sunRayOpticalDepth +
viewRayOpticalDepth) * scatteringCoefficients);
            float localDensity = densityAtPoint(inScatterPoint);

            inScatteredLight += localDensity * transmittance *
scatteringCoefficients * stepSize;
            inScatterPoint += rayDir * stepSize;
        }
        float originalColTransmittance = exp(-
viewRayOpticalDepth);
        return originalCol * originalColTransmittance +
inScatteredLight;
```

```
    }


//
########################################################################
//
||                              MAIN                                  ||
//
########################################################################


    void main() {
        // Get planet texture
        vec4 originalCol = texture(tDiffuse, vUv);
        gl_FragColor = originalCol;

        // Calculate world coordinates relative to screen
        float z = texture2D(tDepth, vUv).x;
        vec3 posWorldScreen = _ScreenToWorld(vec3(vUv, z));

        // Setup Raytrace origin and direction
        vec3 rayOrigin = cameraPosition;
        vec3 rayDirection = normalize(posWorldScreen -
cameraPosition);

        // Raytrace to the planet to get the distance
        float dstToSurface = raySphere(planetPosition,
planetRadius, rayOrigin, rayDirection).x;

        // Raytrace to the atmosphere
        vec2 hitInfo = raySphere(planetPosition, atmosphereRadius,
rayOrigin, rayDirection);
        float dstToAtmosphere = hitInfo.x;
        float dstThroughAtmosphere = min(hitInfo.y, dstToSurface -
dstToAtmosphere);

        if (dstThroughAtmosphere > 0.0) {
            float epsilon = 0.0001; // This is for removing the
noise artifacts caused by precision issues
            vec3 pointInAtmosphere = rayOrigin + rayDirection *
(dstToAtmosphere + epsilon);
            vec3 light = calculateLight(pointInAtmosphere,
rayDirection, dstThroughAtmosphere - epsilon * 2.,
originalCol.xyz);
            gl_FragColor = vec4(light, 0.0);
        }

    }


`; export default atmosFrag;
```

[D] HEIGHTMAP

HEIGHTMAPFRAG.JS

```
// https://threejs.org/docs/#api/en/renderers/webgl/WebGLProgram
import COMMON from '../../shaders/textureMap/commonFrag.js';

const fragmentShader = COMMON + /* glsl */ `


//
//////////////////////////////////////////////////////////////////
// ||                    VARIABLES &
QUALIFIERS                    ||
//
//////////////////////////////////////////////////////////////////


varying vec2 vUv; // The "coordinates" in UV mapping
representation
uniform float uTime; // Time in seconds since load
uniform vec2 uResolution; // Canvas size (width,height)

// Buffers
uniform sampler2D iChannel0;

//////////////////////////////////////////////////////////////////
/////////////

#define buf(p) textureLod(iChannel0,(p)/uResolution.xy,0.)


//
//////////////////////////////////////////////////////////////////
//
||                            MAIN                                ||
//
//////////////////////////////////////////////////////////////////


void main() {
    vec2 p = vUv * uResolution.xy;
    float height = MAP_HEIGHT( buf(p).z );
    gl_FragColor =  vec4(height,height,height,1.0);
}


`; export default fragmentShader;
```

[D] NORMALMAP

NORMALMAPFRAG.JS

```
// https://threejs.org/docs/#api/en/renderers/webgl/WebGLProgram
const fragShader = /* glsl */ `


//
//////////////////////////////////////////////////////////////////
// ||                    VARIABLES &
QUALIFIERS                    ||
//
//////////////////////////////////////////////////////////////////


varying vec2 vUv; // The "coordinates" in UV mapping
representation
uniform vec2 uResolution; // Canvas size (width,height)
uniform sampler2D uHeightMap; // Height map texture


//
//////////////////////////////////////////////////////////////////
//
||                          FUNCTIONS                           ||
//
//////////////////////////////////////////////////////////////////


float getBrightness(vec4 color) {
    float bright = 1.0 - (0.2126*color.r + 0.7152*color.g +
0.0722*color.b);
    return bright;
}


//
//////////////////////////////////////////////////////////////////
//
||                            MAIN                              ||
//
//////////////////////////////////////////////////////////////////


void main() {

    float x = vUv.x;
    float y = vUv.y;
    float pixelSize = 1.0 / uResolution.x;

    // float strength = scale.Y / 16;
    float strength = 0.8;
```

```
    float tl = getBrightness(texture2D(uHeightMap, vec2(x-
pixelSize, y-pixelSize)));
    float l = getBrightness(texture2D(uHeightMap, vec2(x-
pixelSize, y)));
    float bl = getBrightness(texture2D(uHeightMap, vec2(x-
pixelSize, y+pixelSize)));
    float b = getBrightness(texture2D(uHeightMap, vec2(x,
y+pixelSize)));
    float br = getBrightness(texture2D(uHeightMap,
vec2(x+pixelSize, y+pixelSize)));
    float r = getBrightness(texture2D(uHeightMap,
vec2(x+pixelSize, y)));
    float tr = getBrightness(texture2D(uHeightMap,
vec2(x+pixelSize, y-pixelSize)));
    float t = getBrightness(texture2D(uHeightMap, vec2(x, y-
pixelSize)));

    // Compute dx using Sobel:
    //           -1 0 1
    //           -2 0 2
    //           -1 0 1
    float dX = tr + 2.0 * r + br - tl - 2.0 * l - bl;

    // Compute dy using Sobel:
    //           -1 -2 -1
    //            0  0  0
    //            1  2  1
    float dY = bl + 2.0 * b + br - tl - 2.0 * t - tr;


    vec3 N = vec3(dX, dY, 1.0 / strength);

    normalize(N);
    N = N * 0.5 + 0.5;

    gl_FragColor = vec4(N, 1.0);
}


`; export default fragShader;
```

[D] STARSSHADERS

STARSFRAG.JS

```
// https://threejs.org/docs/#api/en/renderers/webgl/WebGLProgram
const fragShader = /* glsl */ `

// ------------------------------------------------------------------
-------------

float PI = 3.14159265358979323846264338;
varying vec2 vUv; // The "coordinates" in UV mapping
representation

uniform vec2 u_resolution;

// ------------------------------------------------------------------
-------------

// A hash function
float hash( const in float n ) {
    return fract(sin(n)*4378.5453);
}

// 3D Perlin noise
float pnoise(in vec3 o) {
    vec3 p = floor(o);
    vec3 fr = fract(o);

    float n = p.x + p.y*57.0 + p.z * 1009.0;

    float a = hash(n+  0.0);
    float b = hash(n+  1.0);
    float c = hash(n+ 57.0);
    float d = hash(n+ 58.0);

    float e = hash(n+  0.0 + 1009.0);
    float f = hash(n+  1.0 + 1009.0);
    float g = hash(n+ 57.0 + 1009.0);
    float h = hash(n+ 58.0 + 1009.0);


    vec3 fr2 = fr * fr;
    vec3 fr3 = fr2 * fr;

    vec3 t = 3.0 * fr2 - 2.0 * fr3;

    float u = t.x;
    float v = t.y;
    float w = t.z;

    float res1 = a + (b-a)*u +(c-a)*v + (a-b+d-c)*u*v;
    float res2 = e + (f-e)*u +(g-e)*v + (e-f+h-g)*u*v;
```

```
    float res = res1 * (1.0- w) + res2 * (w);

    return res;
}

float SmoothNoise( vec3 p ) {
    float f;
    mat3 m = mat3( 0.00,  0.80,  0.60,
                  -0.80,  0.36, -0.48,
                  -0.60, -0.48,  0.64 );
    f  = 0.5000*pnoise( p ); p = m*p*2.02;
    f += 0.2500*pnoise( p );

    return f * (1.0 / (0.5000 + 0.2500));
}

// Credits to uqone https://www.shadertoy.com/view/WdX3D4
vec3 getStars(in vec3 from, in vec3 dir, float power) {
    float scale = 2048.;
    float density = 20.;
    vec3 color = vec3(pow(SmoothNoise(dir*scale), density));
    return pow(color*2.25, vec3(power));
}


void main() {
    vec2 uvo=vUv*2.-1.;
    uvo.y*=(u_resolution.y/u_resolution.x);

    vec3 dir=normalize(vec3(uvo,.8));
    dir = normalize(dir);

    vec3 from=vec3(0.0);
    vec3 colorStars=clamp(getStars(from, dir, 0.9), 0.0, 1.0);
    vec3 color=clamp(colorStars,vec3(0.0),vec3(1.0));
    color = pow(color, vec3(1.2));

    gl_FragColor = vec4(color,1.0);
}

`;
export default fragShader;
```

STARSVERTEX.JS

```
// https://threejs.org/docs/#api/en/renderers/webgl/WebGLProgram
const vertShader = /* glsl */ `

varying vec2 vUv;

void main() {
    vUv = uv;
    gl_Position = projectionMatrix * modelViewMatrix * vec4(
position, 1.0 );
}

`;
export default vertShader;
```

[D] SUNSHADERS

SUNFRAG.JS

```
// https://threejs.org/docs/#api/en/renderers/webgl/WebGLProgram
const fragShader = /* glsl */ `

// ------------------------------------------------------------
------------

varying vec2 vUv; // The "coordinates" in UV mapping
representation
varying float distToCamera;

varying vec4 vCenter;
uniform float uSize;

// ------------------------------------------------------------
------------

// Brightness to color
vec3 brightnessToColor(float b) {
    b *= 0.25;

    return (vec3(b, b*b, b*b*b*b)/0.25)*0.5; // orange
    // return (vec3( b*b*b, b*b, b)/0.25)*0.5; // blue
    // return (vec3(b, b*b, b*b)/0.25)*0.5; // red
}


void main() {
    float radial = distToCamera - vCenter.w;
    radial *= (2./uSize);

    float brightness = 1. + (radial * 1.2);

    gl_FragColor.rgb = brightnessToColor(brightness)*radial;
    gl_FragColor.a = radial;
}

`;
export default fragShader;
```

SUNVERTEX.JS

```
// https://threejs.org/docs/#api/en/renderers/webgl/WebGLProgram
const vertShader = /* glsl */ `

varying vec2 vUv;
varying float distToCamera;
varying vec4 vCenter;

uniform vec3 uCenter;


void main() {
    vUv = uv;

    gl_Position = projectionMatrix * (modelViewMatrix *
vec4(position, 1.0));
    distToCamera = gl_Position.w;
    vCenter = projectionMatrix * viewMatrix * vec4( uCenter, 1.0
);
}

`;
export default vertShader;
```

[D] TEXTUREMAP

CIRCULATIONFRAG.JS

```
// https://threejs.org/docs/#api/en/renderers/webgl/WebGLProgram
// Atmospheric pressure and circulation model
// Inspired & based on David A. Robert's work <https://davidar.io>

import COMMON from './commonFrag.js';

const fragmentShader = COMMON + /* glsl */ `


//
////////////////////////////////////////////////////////////////
// ||                    VARIABLES &
QUALIFIERS                    ||
//
////////////////////////////////////////////////////////////////


varying vec2 vUv; // The "coordinates" in UV mapping
representation
varying vec3 vPosition; // Vertex position

uniform float uTime; // Time in seconds since load
uniform float uFrame; // Frame number
uniform vec3 uResolution; // Canvas size (width,height)

// Texture channels from other buffers
uniform sampler2D iChannel0;
uniform sampler2D iChannel1;

//////////////////////////////////////////////////////////////////
/////////////

#define buf(uv) texture(iChannel1, uv)
#define SIGMA vec4(6,4,1,0) // for gaussian blur


//
////////////////////////////////////////////////////////////////
//
||                          FUNCTIONS                          ||
//
////////////////////////////////////////////////////////////////


// Normal probability density function
vec4 normpdf(float x) {
    return 0.39894 * exp(-0.5 * x*x / (SIGMA*SIGMA)) / SIGMA;
}
```

```
// mean sea level pressure
vec4 mslp(vec2 uv) {
    float lat = 180. * (uv.y * uResolution.y / MAP_RES.y) - 90.;
    float y = textureLod(iChannel0, uv * MAP_ZOOM, MAP_LOD).z;
    float height = MAP_HEIGHT(y);
    vec4 r;
    if (y > OCEAN_DEPTH) { // land
        r.x = 1012.5 - 6. * cos(lat*PI/45.); // annual mean
        r.y = 15. * sin(lat*PI/90.); // January/July delta
    } else { // ocean
        r.x = 1014.5 - 20. * cos(lat*PI/30.); // annual mean
        r.y = 20. * sin(lat*PI/35.) * abs(lat)/90.; // delta
    }
    r.z = height;
    return r;
}


// horizontally blurred MSLP
vec4 pass1(vec2 uv) {
    vec4 r = vec4(0);
    for (float i = -20.; i <= 20.; i++)
        r += mslp(uv + i*E/uResolution.xy) * normpdf(i);
    return r;
}


// fully blurred MSLP
vec4 pass2(vec2 uv) {
    vec4 r = vec4(0);
    for (float i = -20.; i <= 20.; i++)
        r += buf(uv + i*N/uResolution.xy + PASS1) * normpdf(i);
    return r;
}


// time-dependent MSLP and temperature
vec4 pass3(vec2 uv) {
    vec4 c = buf(uv + PASS2);
    float t = mod(uTime/2., 12.); // simulated month of the year
    float delta = 1. - 2. * smoothstep(1.5, 4.5, t) + 2. *
smoothstep(7.5, 10.5, t);
    float mbar = c.x + c.y * delta;

    float lat = 180. * (uv.y * uResolution.y / MAP_RES.y) - 90.;
    float land = step(OCEAN_DEPTH, textureLod(iChannel0, uv *
MAP_ZOOM, MAP_LOD).z);
    float height = c.z;
    float temp = -27. + 73. * tanh(2.2 * exp(-0.5 * pow((lat + 5.
* delta)/30., 2.)));
    temp -= mbar - 1012.;
    temp /= 1.8;
```

```
    temp += 1.5 * land;
    float th = 4.;

    return vec4(mbar, temp - th * height, temp, 0);
}


// wind vector field
vec4 pass4(vec2 uv) {
    vec2 p = uv * uResolution.xy;
    float n = buf(mod(p + N, MAP_RES)/uResolution.xy + PASS3).x;
    float e = buf(mod(p + E, MAP_RES)/uResolution.xy + PASS3).x;
    float s = buf(mod(p + S, MAP_RES)/uResolution.xy + PASS3).x;
    float w = buf(mod(p + W, MAP_RES)/uResolution.xy + PASS3).x;
    vec2 grad = vec2(e - w, n - s) / 2.;
    float lat = 180. * fract(uv.y * uResolution.y / MAP_RES.y) -
90.;
    vec2 coriolis = 15. * sin(lat*PI/180.) * vec2(-grad.y,
grad.x);
    vec2 v = coriolis - grad;
    return vec4(v,0,0);
}


//
////////////////////////////////////////////////////////////////////
//
||                              MAIN                                ||
//
////////////////////////////////////////////////////////////////////


void main() {
    vec2 fragCoord = vUv * uResolution.xy;
    vec2 uv = fragCoord / uResolution.xy;

    if (uv.x < 0.5) {
        // Down left
        if (uv.y < 0.5) {
            gl_FragColor = pass1(uv - PASS1);

        // Up left
        } else {
            gl_FragColor = pass2(uv - PASS2);
        }
    } else {
        // Down right
        if (uv.y < 0.5) {
            gl_FragColor = pass3(uv - PASS3);
        // Up right
        } else {
            gl_FragColor = pass4(uv - PASS4);
        }
```

```
    }

}


`; export default fragmentShader;
```

COMMONFRAG.JS

```
// https://threejs.org/docs/#api/en/renderers/webgl/WebGLProgram
// Inspired & based on David A. Robert's work <https://davidar.io>

const fragmentShader = /* glsl */ `


//
////////////////////////////////////////////////////////////////////
// ||                  VARIABLES &
QUALIFIERS                    ||
//
////////////////////////////////////////////////////////////////////

uniform float uSeed;  // seed
uniform int uScene; // Scene number

#define PI 3.14159265359

// Events
#define OCEAN_START_TIME 15.
#define LAND_START_TIME 20.
#define OCEAN_END_TIME 25.
#define LAND_END_TIME 30.
#define TECTONICS_END_TIME 35.
#define STORY_END_TIME 35.

#define OCEAN_DEPTH ocean_depth(uTime)

#define MAP_HEIGHT(y) (0.4 * max(0., (y) - OCEAN_DEPTH))

#define MAP_LOD max(1., floor(log2(uResolution.x / 144.)))
#define MAP_ZOOM pow(2., MAP_LOD)
#define MAP_RES (uResolution.xy / MAP_ZOOM)

#define PASS1 vec2(0.0,0.0)
#define PASS2 vec2(0.0,0.5)
#define PASS3 vec2(0.5,0.0)
#define PASS4 vec2(0.5,0.5)

// Directions
#define N  vec2( 0, 1)
#define NE vec2( 1, 1)
#define E  vec2( 1, 0)
#define SE vec2( 1,-1)
#define S  vec2( 0,-1)
#define SW vec2(-1,-1)
#define W  vec2(-1, 0)
#define NW vec2(-1, 1)


//
////////////////////////////////////////////////////////////////////
```

```
//
||                          FUNCTIONS                              ||
//
#####################################################################


float hash13(vec3);
vec2 plate_move(float q, float uFrame, float uTime) {
    // q = plate color (0.-1.)

    // No movement outside the time range
    if (uTime >= TECTONICS_END_TIME && uTime < STORY_END_TIME)
return vec2(0);

    // Get a pseudorandom value between (0,0) and (1,1)
    vec2 v = vec2(cos(2.*PI*q), sin(2.*PI*q));

    // Chances of movement in the current frame (0.5 = 50% chance)
    // ToDo: Change this value via uniform
    if (hash13(vec3(v,uFrame)) < 0.05) {

        // Decide the direction of the movement
        if (hash13(vec3(v+1.,uFrame)) < abs(v.x) / (abs(v.x) +
abs(v.y))) {
            return vec2(sign(v.x),0.);
        } else {
            return vec2(0.,sign(v.y));
        }
    }
    return vec2(0);
}

float ocean_depth(float t) {
    if (uScene == 1) return 5.; // Water world
    if (uScene == 2) return 0.; // Red planet
    if (TECTONICS_END_TIME < t && t < STORY_END_TIME) t =
TECTONICS_END_TIME;
    float d = 7.25 + 0.25 * sin(t/5.);
    d *= smoothstep(OCEAN_START_TIME, OCEAN_END_TIME, t);
    return d;
}

float plant_growth(float moisture, float temp) {
    float growth = clamp(moisture / 3., 0., 1.);
    //
https://www2.nrel.colostate.edu/projects/century/MANUAL/html_manua
l/man96.html#FIG3-8B
    growth *= smoothstep(0., 25., temp) - smoothstep(25., 35.,
temp);
    return growth;
}
```

```
// Hash without Sine
// Creative Commons Attribution-ShareAlike 4.0 International
Public License
// Created by David Hoskins.
#define HASHSCALE1 .1031 * uSeed
#define HASHSCALE3 vec3(.1031, .1030, .0973) * uSeed
#define HASHSCALE4 vec4(.1031, .1030, .0973, .1099) * uSeed

//----------------------------------------------------------------
------------------------
//  1 out, 1 in...
float hash11(float p)
{
    vec3 p3  = fract(vec3(p) * HASHSCALE1);
    p3 += dot(p3, p3.yzx + 19.19);
    return fract((p3.x + p3.y) * p3.z);
}

//----------------------------------------------------------------
------------------------
//  1 out, 2 in...
float hash12(vec2 p)
{
    vec3 p3  = fract(vec3(p.xyx) * HASHSCALE1);
    p3 += dot(p3, p3.yzx + 19.19);
    return fract((p3.x + p3.y) * p3.z);
}

//----------------------------------------------------------------
------------------------
//  1 out, 3 in...
float hash13(vec3 p3)
{
    p3  = fract(p3 * HASHSCALE1);
    p3 += dot(p3, p3.yzx + 19.19);
    return fract((p3.x + p3.y) * p3.z);
}

//----------------------------------------------------------------
------------------------
//  2 out, 1 in...
vec2 hash21(float p)
{
    vec3 p3 = fract(vec3(p) * HASHSCALE3);
    p3 += dot(p3, p3.yzx + 19.19);
    return fract((p3.xx+p3.yz)*p3.zy);

}

//----------------------------------------------------------------
------------------------
///  2 out, 2 in...
vec2 hash22(vec2 p)
```

```
{
    vec3 p3 = fract(vec3(p.xyx) * HASHSCALE3);
    p3 += dot(p3, p3.yzx+19.19);
    return fract((p3.xx+p3.yz)*p3.zy);

}

//----------------------------------------------------------------
------------------------
///  2 out, 3 in...
vec2 hash23(vec3 p3)
{
    p3 = fract(p3 * HASHSCALE3);
    p3 += dot(p3, p3.yzx+19.19);
    return fract((p3.xx+p3.yz)*p3.zy);
}

//----------------------------------------------------------------
------------------------
//  3 out, 1 in...
vec3 hash31(float p)
{
   vec3 p3 = fract(vec3(p) * HASHSCALE3);
   p3 += dot(p3, p3.yzx+19.19);
   return fract((p3.xxy+p3.yzz)*p3.zyx);
}


//----------------------------------------------------------------
------------------------
///  3 out, 2 in...
vec3 hash32(vec2 p)
{
    vec3 p3 = fract(vec3(p.xyx) * HASHSCALE3);
    p3 += dot(p3, p3.yxz+19.19);
    return fract((p3.xxy+p3.yzz)*p3.zyx);
}

//----------------------------------------------------------------
------------------------
///  3 out, 3 in...
vec3 hash33(vec3 p3)
{
    p3 = fract(p3 * HASHSCALE3);
    p3 += dot(p3, p3.yxz+19.19);
    return fract((p3.xxy + p3.yxx)*p3.zyx);

}

//----------------------------------------------------------------
------------------------
// 4 out, 1 in...
vec4 hash41(float p)
```

```
{
    vec4 p4 = fract(vec4(p) * HASHSCALE4);
    p4 += dot(p4, p4.wzxy+19.19);
    return fract((p4.xxyz+p4.yzzw)*p4.zywx);

}

//----------------------------------------------------------------------
-----------------------
// 4 out, 2 in...
vec4 hash42(vec2 p)
{
    vec4 p4 = fract(vec4(p.xyxy) * HASHSCALE4);
    p4 += dot(p4, p4.wzxy+19.19);
    return fract((p4.xxyz+p4.yzzw)*p4.zywx);

}

//----------------------------------------------------------------------
-----------------------
// 4 out, 3 in...
vec4 hash43(vec3 p)
{
    vec4 p4 = fract(vec4(p.xyzx)  * HASHSCALE4);
    p4 += dot(p4, p4.wzxy+19.19);
    return fract((p4.xxyz+p4.yzzw)*p4.zywx);
}

//----------------------------------------------------------------------
-----------------------
// 4 out, 4 in...
vec4 hash44(vec4 p4)
{
    p4 = fract(p4  * HASHSCALE4);
    p4 += dot(p4, p4.wzxy+19.19);
    return fract((p4.xxyz+p4.yzzw)*p4.zywx);
}


// By David Hoskins, May 2014. @
https://www.shadertoy.com/view/4dsXWn
// License Creative Commons Attribution-NonCommercial-ShareAlike
3.0 Unported License.

float Noise(in vec3 p)
{
    vec3 i = floor(p);
        vec3 f = fract(p);
        f *= f * (3.0-2.0*f);

    return mix(
                mix(mix(hash13(i + vec3(0.,0.,0.)), hash13(i +
vec3(1.,0.,0.)),f.x),
```

```
                        mix(hash13(i + vec3(0.,1.,0.)), hash13(i +
vec3(1.,1.,0.)),f.x),
                        f.y),
                mix(mix(hash13(i + vec3(0.,0.,1.)), hash13(i +
vec3(1.,0.,1.)),f.x),
                        mix(hash13(i + vec3(0.,1.,1.)), hash13(i +
vec3(1.,1.,1.)),f.x),
                        f.y),
                f.z);
}

const mat3 m = mat3( 0.00,  0.80,  0.60,
                    -0.80,  0.36, -0.48,
                    -0.60, -0.48,  0.64 ) * 1.7;

float FBM( vec3 p )
{
    float f;

        f = 0.5000 * Noise(p); p = m*p;
        f += 0.2500 * Noise(p); p = m*p;
        f += 0.1250 * Noise(p); p = m*p;
        f += 0.0625   * Noise(p); p = m*p;
        f += 0.03125  * Noise(p); p = m*p;
        f += 0.015625 * Noise(p);
    return f;
}


`;
export default fragmentShader;
```

GEOLOGYFRAG.JS

```
// https://threejs.org/docs/#api/en/renderers/webgl/WebGLProgram
// Plate tectonics and hydraulic erosion model
// Inspired & based on David A. Robert's work <https://davidar.io>

import COMMON from '../../shaders/textureMap/commonFrag.js';

const fragmentShader = COMMON + /* glsl */ `


//
////////////////////////////////////////////////////////////////
// ||                    VARIABLES &
QUALIFIERS                    ||
//
////////////////////////////////////////////////////////////////


varying vec2 vUv; // The "coordinates" in UV mapping
representation
varying vec3 vPosition; // Vertex position

uniform float uTime; // Time in seconds since load
uniform float uFrame; // Frame number
uniform vec3 uResolution; // Canvas size (width,height)
uniform vec2 uMouse; // Mouse position (width,height)
uniform bool uMouseClick; // True if mouse is pressed
uniform bool uAddingTerrain; // True if adding terrain
uniform bool uRemovingTerrain; // True if removing terrain


// Texture channels from other buffers
uniform sampler2D iChannel0;

//////////////////////////////////////////////////////////////////
/////////////

#define buf(p) textureLod(iChannel0,fract((p) /
uResolution.xy),0.)


//
////////////////////////////////////////////////////////////////
//
||                          FUNCTIONS                          ||
//
////////////////////////////////////////////////////////////////


// Generate procedural craters based on
https://www.shadertoy.com/view/MtjGRD
float craters(vec3 x) {
    vec3 p = floor(x);
```

```
    vec3 f = fract(x); // Same as x - floor(x)
    float va = 0.;
    float wt = 0.;
    for (int i = -2; i <= 2; i++) {
        for (int j = -2; j <= 2; j++) {
            for (int k = -2; k <= 2; k++) {
                vec3 g = vec3(i,j,k);
                vec3 o = 0.8 * hash33(p + g);
                float d = distance(f - g, o);
                float w = exp(-4. * d);
                va += w * sin(2.*PI * sqrt(d));
                wt += w;
            }
        }
    }
    return abs(va / wt);
}


vec2 move(float q) {
    return plate_move(q, uFrame, uTime);
}


// Slope of the terrain between two points
float slope(vec2 p, vec2 q) {
    if (p == q) return 0.;
    return (buf(q).z - buf(p).z) / distance(p,q);
}


// Direction of water flow at point p
vec2 rec(vec2 p) {
    vec2 d = vec2(0);
    if (slope(p + N,  p) >= slope(p + d, p)) d = N;
    if (slope(p + NE, p) >= slope(p + d, p)) d = NE;
    if (slope(p + E,  p) >= slope(p + d, p)) d = E;
    if (slope(p + SE, p) >= slope(p + d, p)) d = SE;
    if (slope(p + S,  p) >= slope(p + d, p)) d = S;
    if (slope(p + SW, p) >= slope(p + d, p)) d = SW;
    if (slope(p + W,  p) >= slope(p + d, p)) d = W;
    if (slope(p + NW, p) >= slope(p + d, p)) d = NW;
    return d;
}


// Transforms a 2D vertex coordinate to 3D cartesian coordinates
given latitude and longitude
// This is used to deform and wrap a 2D plane into a 3D Sphere
vec3 planeToCartesian(vec2 uv) {
    float scale = 1.5;
    float lat = 180. * uv.y - 90.;
    float lon = 360. * uv.x;
```

```
    return scale * vec3( sin(lon*PI/180.) * cos(lat*PI/180.),
sin(lat*PI/180.), cos(lon*PI/180.) * cos(lat*PI/180.));
}


// Creates a heightmap of a terrain with craters
float protoplanet(vec2 uv) {
    float height = 0.;

     // Modify the vertex position to be projected into a sphere
     vec3 p = planeToCartesian(uv);

    // Multiple passes various crater sizes
    for (float i = 0.; i < 5.; i++) {

        // Generate the craters
        float c = 0.;
        if (uScene == 2) c = craters(vec3(0.15 * pow(2.5, i) *
p));
        else c = craters(vec3(0.4 * pow(2.2, i) * p));

        // Generate the FBM noise
        float noise = 0.4 * exp(-3. * c) * FBM(10. * p);

        // Constrain a value to lie between two further values
        float x = 3. * pow(0.4, i);
        float min = 0.;
        float max = 1.;
        float w = clamp(x, min, max);

        // Mix and add the result
        height += w * (c + noise);
    }

    // Play with the contrast
    return pow(height, 3.);
}


//
//////////////////////////////////////////////////////////////////////
//
||                          MAIN                                  ||
//
//////////////////////////////////////////////////////////////////////


void main() {
    vec2 p = vUv * uResolution.xy;


    //
```

```
    ////////////////////////////////////////////////////////////////
    //                                                            //
    //                          TERRAIN                           //
    //                                                            //
    ////////////////////////////////////////////////////////////////
    if(uTime < OCEAN_START_TIME && mod(uFrame, 50.) < 1. || uFrame
< 10.) {
        gl_FragColor = vec4(0);

        // Generate new plate boundaries
        gl_FragColor.x = -1.;

        // Rivers
        gl_FragColor.w = hash12(p);

        // Terrain elevation
        if (uScene == 1) gl_FragColor.z = clamp(15. - 15. *
protoplanet(p / uResolution.xy), 0., 15.);
        else if (uScene == 2) gl_FragColor.z = clamp(15. - 3. *
protoplanet(p / uResolution.xy), 0., 15.);
        else gl_FragColor.z = clamp(15. - 3.5 * protoplanet(p /
uResolution.xy), 0., 15.);

        return;
    }

    //
    ////////////////////////////////////////////////////////////////
    // ||                       HYDRAULIC
EROSION                       ||
    //
    ////////////////////////////////////////////////////////////////

    gl_FragColor = buf(p);

    if (uScene == 1 || uScene == 2) return;

    if (uTime < OCEAN_START_TIME) return;
    float smoothstart = smoothstep(OCEAN_START_TIME,
OCEAN_END_TIME, uTime);

    // Neighbour pixels
    vec4 n = buf(p + N);
    vec4 e = buf(p + E);
    vec4 s = buf(p + S);
    vec4 w = buf(p + W);

    if (uTime < TECTONICS_END_TIME || uTime > STORY_END_TIME) {
        // diffuse uplift through plate
        float dy = 0.;
        if (e.x == gl_FragColor.x) dy += e.y - gl_FragColor.y;
        if (w.x == gl_FragColor.x) dy += w.y - gl_FragColor.y;
        if (n.x == gl_FragColor.x) dy += n.y - gl_FragColor.y;
```

```
        if (s.x == gl_FragColor.x) dy += s.y - gl_FragColor.y;
        gl_FragColor.y = max(0., gl_FragColor.y + 0.1 * dy);
    }

    // Tectonic uplift
    float max_uplift = 1.;
    if (gl_FragColor.z - OCEAN_DEPTH > 1.) max_uplift = 1. /
(gl_FragColor.z - OCEAN_DEPTH);
    gl_FragColor.z += clamp(2. * gl_FragColor.y - 1., 0.,
max_uplift);

    if (gl_FragColor.z >= OCEAN_DEPTH - 0.05) {
        // Thermal erosion
        float dz = 0.;
        if (abs(e.z - gl_FragColor.z) > 1.) dz += e.z -
gl_FragColor.z;
        if (abs(w.z - gl_FragColor.z) > 1.) dz += w.z -
gl_FragColor.z;
        if (abs(n.z - gl_FragColor.z) > 1.) dz += n.z -
gl_FragColor.z;
        if (abs(s.z - gl_FragColor.z) > 1.) dz += s.z -
gl_FragColor.z;
        gl_FragColor.z = max(0., gl_FragColor.z + 0.02 * dz);

        // Flow accumulation
        gl_FragColor.w = 1. + fract(gl_FragColor.w);
        if (rec(p + N)  == -N)  gl_FragColor.w += floor(buf(p +
N).w);
        if (rec(p + NE) == -NE) gl_FragColor.w += floor(buf(p +
NE).w);
        if (rec(p + E)  == -E)  gl_FragColor.w += floor(buf(p +
E).w);
        if (rec(p + SE) == -SE) gl_FragColor.w += floor(buf(p +
SE).w);
        if (rec(p + S)  == -S)  gl_FragColor.w += floor(buf(p +
S).w);
        if (rec(p + SW) == -SW) gl_FragColor.w += floor(buf(p +
SW).w);
        if (rec(p + W)  == -W)  gl_FragColor.w += floor(buf(p +
W).w);
        if (rec(p + NW) == -NW) gl_FragColor.w += floor(buf(p +
NW).w);

        // No slope
        if (rec(p) == vec2(0)) { // local minima
            gl_FragColor.z += 0.001; // extra sediment

        // Slope
        } else {
            vec4 receiver = buf(p + rec(p)); // punt que reb el
flux
            if (gl_FragColor.z >= OCEAN_DEPTH) gl_FragColor.w =
floor(gl_FragColor.w) + fract(receiver.w); // basin colouring
```

```
            // hydraulic erosion with stream power law
            float pslope = (gl_FragColor.z - receiver.z) /
length(rec(p)); // slope of the stream
            float dz = min(pow(floor(gl_FragColor.w), 0.8) *
pow(pslope, 2.), gl_FragColor.z);
            dz *= smoothstart;
            gl_FragColor.z = max(gl_FragColor.z - 0.05 * dz,
receiver.z);
        }

    } else {
        gl_FragColor.w = fract(gl_FragColor.w);
    }

    // approximation of sediment accumulation
    if (uTime < TECTONICS_END_TIME || uTime > STORY_END_TIME) {
        gl_FragColor.z += 2.5e-4 * clamp(gl_FragColor.z + 2.5, 0.,
10.) * smoothstart;
    } else if (gl_FragColor.z >= OCEAN_DEPTH - 0.05) {
        gl_FragColor.z += 2.5e-4;
    }




    //
##################################################################
    //
||                          TECTONICS                           ||
    //
##################################################################

    bool subduct = false;

    // Generate new plate boundaries every 80 seconds
    if (mod(uFrame, 5000.) < 10.) {
        gl_FragColor.x = -1.;

    // No plate under this point yet
    } else if (gl_FragColor.x < 0.) {

        // Randomly select a new plate
        // Todo: Make this a uniform
        if (length(hash33(vec3(p,uFrame))) < 0.006) {
            // Seed a new plate with random velocity
            gl_FragColor.x = fract(hash13(vec3(p,uFrame)) + 0.25);

        // Accretion
        } else {
            // Expansion of the current plate
            int dir = int(4.*hash13(vec3(p,uFrame)));
            switch (dir) {
```

```
                case 0: gl_FragColor.x = s.x; break;
                case 1: gl_FragColor.x = w.x; break;
                case 2: gl_FragColor.x = n.x; break;
                case 3: gl_FragColor.x = e.x; break;
            }
        }
        gl_FragColor.y = clamp(gl_FragColor.y, 0., 1.);

    // Si el moviment es cap al sud
    } else if (move(n.x) == S) {
        if (move(gl_FragColor.x) != S) subduct = true;
        gl_FragColor = n;

    // Si el moviment es cap al oest
    } else if (move(e.x) == W) {
        if (move(gl_FragColor.x) != W) subduct = true;
        gl_FragColor = e;

    // Si el moviment es cap al nord
    } else if (move(s.x) == N) {
        if (move(gl_FragColor.x) != N) subduct = true;
        gl_FragColor = s;

    // Si el moviment es cap al est
    } else if (move(w.x) == E) {
        if (move(gl_FragColor.x) != E) subduct = true;
        gl_FragColor = w;

    // Rift
    } else if (move(gl_FragColor.x) != vec2(0) && buf(p -
move(gl_FragColor.x)).x >= 0.) {
        gl_FragColor.x = -1.;
        if (gl_FragColor.z < OCEAN_DEPTH) {
            gl_FragColor.y = 0.;
            gl_FragColor.z = 0.;
        }
        gl_FragColor.w = hash12(p);
    }

    // Subduction
    if (subduct) {
        gl_FragColor.y = 1.;
    } else if (uTime < TECTONICS_END_TIME || uTime >
STORY_END_TIME) {
        gl_FragColor.y = max(gl_FragColor.y - 0.0001, 0.);
    }


    //
##################################################################
    //
||                          TERRAFORM                           ||
    //
```

```
######################################################################
    float brushSize = 20.; // ToDo: make this a parameter
    vec2 r = (p - (uMouse.xy*uResolution.xy)) / brushSize;
    float magnitude = 0.;
    if (uAddingTerrain) magnitude = 0.5 * exp(-0.5 * dot(r,r));
    else if (uRemovingTerrain) magnitude = -1. * (0.5 * exp(-0.5 *
dot(r,r)));
    if (uMouseClick) gl_FragColor.z += magnitude;



    gl_FragColor.y = clamp(gl_FragColor.y, 0., 1.);
    gl_FragColor.z = max(gl_FragColor.z, 0.);
}


`; export default fragmentShader;
```

MAINFRAG.JS

```
// https://threejs.org/docs/#api/en/renderers/webgl/WebGLProgram
// Inspired & based on David A. Robert's work <https://davidar.io>

import COMMON from '../../shaders/textureMap/commonFrag.js';

const fragmentShader = COMMON + /* glsl */ `


//
////////////////////////////////////////////////////////////////////
// ||                    VARIABLES &
QUALIFIERS                    ||
//
////////////////////////////////////////////////////////////////////


varying vec2 vUv; // The "coordinates" in UV mapping
representation
varying vec3 vPosition; // Vertex position

uniform float uTime; // Time in seconds since load
uniform vec2 uResolution; // Canvas size (width,height)
uniform int uDisplayTextureMap; // The map to show

// Texture channels from other buffers
uniform sampler2D iChannel0;
uniform sampler2D iChannel1;
uniform sampler2D iChannel2;
uniform sampler2D iChannel3;

//////////////////////////////////////////////////////////////////
/////////////

#define buf(p) textureLod(iChannel0,(p)/uResolution.xy,0.)

#define DEEP_WATER vec4(0.01, 0.02, 0.08, 1)
#define SHALLOW_WATER vec4(0.11, 0.28, 0.51, 1)
#define WARM vec4(1.,0.5,0.,1) // Low pressure color (red)
#define COOL vec4(0.,0.5,1.,1) // High pressure color (blue)


//
////////////////////////////////////////////////////////////////////
//
||                          FUNCTIONS                             ||
//
////////////////////////////////////////////////////////////////////


vec3 fromlatlon(float lat, float lon) {
    return vec3(sin(lon*PI/180.) * cos(lat*PI/180.),
sin(lat*PI/180.), cos(lon*PI/180.) * cos(lat*PI/180.));
```

```
}


vec4 climate(vec2 fragCoord, vec2 pass) {
    vec2 p = fragCoord * MAP_RES / uResolution.xy;
    if (p.x < 0.5) p.x = 0.5;
    vec2 uv = p / uResolution.xy;
    return texture(iChannel1, uv + pass);
}


vec4 map_land(vec2 fragCoord, bool ocean) {
    vec2 p = fragCoord;
    vec2 grad = vec2(buf(p+E).z - buf(p+W).z, buf(p+N).z -
buf(p+S).z);
    float light = cos(atan(grad.y, grad.x) + 0.25*PI) * clamp(0.2
* length(grad), 0., 1.);
    vec4 fragColor = vec4(vec3(0.015 + 0.085 * light), 1);
    if (!ocean) fragColor.rgb *= 3.;
    float y = buf(fragCoord).z;
    if (y < OCEAN_DEPTH) {
        if (ocean)
            fragColor = mix(DEEP_WATER, SHALLOW_WATER, y /
OCEAN_DEPTH);
        else
            fragColor.rgb = vec3(0);
    }
    fragColor.w = MAP_HEIGHT(y);
    return fragColor;
}


vec4 map_flow(vec2 fragCoord) {
    // Get the pressure from the map
    float mbar = climate(fragCoord, PASS3).x;

    // Color based on pressure (mbar)
    vec4 r = WARM;
    r = mix(r, vec4(1), smoothstep(1000., 1012., floor(mbar)));
    r = mix(r, COOL, smoothstep(1012., 1024., floor(mbar)));

    vec2 p = fragCoord * MAP_RES / uResolution.xy;
    if (p.x < 1.) p.x = 1.;
    vec2 uv = p / uResolution.xy;
    vec2 v = texture(iChannel1, uv + PASS4).xy;

    // Add wind vector
    vec4 c = texture(iChannel2, fragCoord/uResolution.xy);
    float flow = (c.x > 0.) ? 1. : c.y;
    flow *= clamp(length(v), 0., 1.);

    // Mix land, temperature and wind vectors
    vec4 fragColor = map_land(fragCoord, false);
```

```
    fragColor = mix(fragColor, r, 0.5 * flow);
    return fragColor;
}


vec4 map_temp(vec2 fragCoord) {
    float height = MAP_HEIGHT(buf(fragCoord).z);
    float temp0 = climate(fragCoord, PASS3).z;
    float temp = temp0 - 3. * height;
    temp = floor(temp/2.)*2.;
    vec4 r = COOL;
    r = mix(r, vec4(1), smoothstep(-20.,  0., temp));
    r = mix(r, WARM,    smoothstep(  0., 25., temp));
    vec4 fragColor = map_land(fragCoord, false);
    fragColor = mix(fragColor, r, 0.35);
    return fragColor;
}


vec4 map_plates(vec2 fragCoord) {
    vec2 p = fragCoord;
    float q = buf(p).x;
    float uplift = buf(p).y;
    vec4 r = vec4(0,0,0,1);
    r.rgb = (q < 0.) ? vec3(1) : .6 + .6 * cos(2.*PI*q +
vec3(0,23,21));
    vec4 fragColor = map_land(fragCoord, false);
    fragColor = r * (5. * fragColor + 3. * clamp(2. * uplift - 1.,
0., 1.) + 0.05);
    return fragColor;
}


vec4 map_rivers(vec2 fragCoord) {
    vec4 fragColor = map_land(fragCoord, true);
    float flow = buf(fragCoord).w;
    fragColor.rgb = mix(fragColor.rgb, .6 + .6 * cos(2.*PI *
fract(flow) + vec3(0,23,21)),
                        clamp(0.15 * log(floor(flow)), 0., 1.));
    return fragColor;
}


vec4 map_sat(vec2 fragCoord) {
    vec2 p = fragCoord;
    vec2 uv = p / uResolution.xy;

    // Overal heightmap
    float y = buf(p).z;

    // Land heightmap
    float height = MAP_HEIGHT(y);
```

```
    // Ocean
    vec4 deepWaterColor = vec4(0.01, 0.02, 0.08, 1);
    vec4 shallowWaterColor = vec4(0.11, 0.28, 0.51, 1);
    vec4 ocean = mix(deepWaterColor, shallowWaterColor, y /
OCEAN_DEPTH);

    // Temperature
    float temp0 = climate(p, PASS3).z;
    float temp = temp0 - 3. * height;

    // Dry land
    vec3 dry = vec3(0.89, 0.9, 0.89);
    dry = mix(dry, vec3(0.11, 0.10, 0.05), smoothstep(-10., 0.,
temp));
    dry = mix(dry, vec3(1.00, 0.96, 0.71), smoothstep( 0., 20.,
temp));
    dry = mix(dry, vec3(0.81, 0.48, 0.31), smoothstep(20., 30.,
temp));

    // Vegetation
    vec3 veg = vec3(0.89, 0.9, 0.89);
    veg = mix(veg, vec3(0.56, 0.49, 0.28), smoothstep(-10., 0.,
temp));
    veg = mix(veg, vec3(0.18, 0.34, 0.04), smoothstep( 0., 20.,
temp));
    veg = mix(veg, vec3(0.05, 0.23, 0.04), smoothstep(20., 30.,
temp));

    float moisture = texture(iChannel3, uv).w;
    vec4 land = vec4(0,0,0,1);
    land.rgb = mix(dry, veg, plant_growth(moisture, temp));

    // Initial rock and heat
    if (uScene == 0 && uTime < LAND_END_TIME) {
        float c = (15. - y) / 3.5;

         // Sun position to add depth to the terrain, otherwise it
would be too flat
        vec2 grad = vec2(buf(p+E).z - buf(p+W).z, buf(p+N).z -
buf(p+S).z);

        // Heat
        float heat = clamp(8. / pow(uTime + 1., 2.), 0., .35);

        // Rock texture
        vec4 rock = mix(vec4(0.58, 0.57, 0.55, 1), vec4(0.15,
0.13, 0.1, 1), smoothstep(0., 3., c));
        rock *= clamp(0.2 * length(grad), 0., 1.);
        rock += 5. * c * heat * vec4(1., 0.15, 0.05, 1.);
        land = mix(rock, land, smoothstep(LAND_START_TIME,
LAND_END_TIME, uTime));
    }
```

```
    vec4 r = vec4(0,0,0,1);
    // Water world
    if (uScene == 1) {
        if (y < OCEAN_DEPTH)  r = ocean;
        else r = land;
    }

    // Red planet
    else if (uScene == 2) {
        dry = vec3(0.89, 0.9, 0.89);
        dry = mix(dry, vec3(0.11, 0.10, 0.05), smoothstep(-10., -
8., temp));
        dry = mix(dry, vec3(1.00, 0.4, 0.21), smoothstep( -8.,
30., temp));
        land.rgb = dry;
        r = land;
    }

    // Earth-like
    else if (y < OCEAN_DEPTH && uTime > OCEAN_START_TIME) {
        r = mix(land, ocean, smoothstep(0., 2., uTime -
OCEAN_START_TIME));
    } else {
        r = land;
    }

    if (uScene == 0) {
        float clouds = 1.;
        float vapour = texture(iChannel2, uv).w;
        r.rgb = mix(r.rgb, vec3(1), 0.5 * clouds * log(1. +
vapour) * smoothstep(0., LAND_END_TIME, uTime));
    }

    return r;
}


vec4 map(vec2 uv) {
    // Pixel coordinates normalized to the screen resolution
    vec2 p = uv * uResolution.xy;

    // Show map to view
    if (uDisplayTextureMap == 0) return map_sat(p);    // Normal
view
    if (uDisplayTextureMap == 1) return map_plates(p); // Plates
view
    if (uDisplayTextureMap == 2) return map_rivers(p); // Rivers
view
    if (uDisplayTextureMap == 3) return map_flow(p);   // Flow
view
    if (uDisplayTextureMap == 4) return map_temp(p);   //
Temperature view
```

```
    return map_sat(p);
}


//
//////////////////////////////////////////////////////////////////
//
||                          MAIN                                ||
//
//////////////////////////////////////////////////////////////////

// // // Some possible values // // //
// uResolution  -> 1024 (default)
// uv               -> 0 - 1
// p                -> (0,0)-(1024,1024)
// gl_FragColor.x  -> 0 - 1
// gl_FragColor.y  -> 0 - 1
// // // ////////////////// // // //

void main() {
    gl_FragColor = map(vUv);

    gl_FragColor.rgb = pow(gl_FragColor.rgb, vec3(1./1.5));
    gl_FragColor.a = 1.;
}


`; export default fragmentShader;
```

SOILFRAG.JS

```
// https://threejs.org/docs/#api/en/renderers/webgl/WebGLProgram
// Soil moisture and vegetation
// Inspired & based on David A. Robert's work <https://davidar.io>

import COMMON from './commonFrag.js';

const fragmentShader = COMMON + /* glsl */ `


//
////////////////////////////////////////////////////////////////////
// ||                    VARIABLES &
QUALIFIERS                    ||
//
////////////////////////////////////////////////////////////////////


varying vec2 vUv; // The "coordinates" in UV mapping
representation
varying vec3 vPosition; // Vertex position

uniform float uTime; // Time in seconds since load
uniform float uFrame; // Frame number
uniform vec3 uResolution; // Canvas size (width,height)

// Texture channels from other buffers
uniform sampler2D iChannel0;
uniform sampler2D iChannel1;
uniform sampler2D iChannel2;
uniform sampler2D iChannel3;

//////////////////////////////////////////////////////////////////////
/////////////

#define map(p) texture(iChannel0,fract((p)/uResolution.xy))
#define buf(p) texture(iChannel3,fract((p)/uResolution.xy))


//
////////////////////////////////////////////////////////////////////
//
||                          FUNCTIONS                             ||
//
////////////////////////////////////////////////////////////////////


vec2 move(float q) {
    return plate_move(q, uFrame, uTime);
}


vec4 climate(vec2 fragCoord, vec2 pass) {
```

```
    vec2 p = fragCoord * MAP_RES / uResolution.xy;
    if (p.x < 0.5) p.x = 0.5;
    vec2 uv = p / uResolution.xy;
    return texture(iChannel1, uv + pass);
}


vec2 offset(vec2 p) {
    vec4 c = map(p);
    vec4 n = map(p + N);
    vec4 e = map(p + E);
    vec4 s = map(p + S);
    vec4 w = map(p + W);

    if( (mod(uFrame, 3000.) < 10.) || c.x < 0.) { // no plate
under this point
        return vec2(0);
    } else if (move(n.x) == S) {
        return N;
    } else if (move(e.x) == W) {
        return E;
    } else if (move(s.x) == N) {
        return S;
    } else if (move(w.x) == E) {
        return W;
    } else if (move(c.x) != vec2(0) && map(p - move(c.x)).x >= 0.)
{ // rift
        return vec2(0);
    }
}


//
////////////////////////////////////////////////////////////////////
//
||                            MAIN                                ||
//
////////////////////////////////////////////////////////////////////


void main() {
    vec2 fragCoord = vUv * uResolution.xy;

    if (uTime < OCEAN_START_TIME) {
        gl_FragColor = vec4(0);
        return;
    }

    float height = MAP_HEIGHT(texture(iChannel0,
fragCoord/uResolution.xy).z);
    float temp = climate(fragCoord, PASS3).y;
    float vapour = texture(iChannel2, fragCoord/uResolution.xy).w;
    vec2 p = fragCoord + offset(fragCoord);
```

```
    vec4 c = buf(p);
    float moisture = c.w;
    moisture *= 1. - 1e-5 * moisture * clamp(temp, 0., 15.);
    if (uTime > OCEAN_END_TIME) moisture += 3. * clamp(vapour, 0.,
0.01);
    moisture = clamp(moisture, 0., 5.);
    if (height == 0.) moisture = 5.;

    if (uTime < OCEAN_END_TIME) {
        c.xyz = hash32(fragCoord) * vec3(0.5, 1., 2.);
    } else {
        c.xyz = vec3(0);
    }

    gl_FragColor = vec4(c.xyz, moisture);
}


`; export default fragmentShader;
```

WINDFRAG.JS

```
// https://threejs.org/docs/#api/en/renderers/webgl/WebGLProgram
// Wind flow map and atmospheric water vapour model
// Inspired & based on David A. Robert's work <https://davidar.io>

import COMMON from './commonFrag.js';

const fragmentShader = COMMON + /* glsl */ `


//
////////////////////////////////////////////////////////////////////
// ||                    VARIABLES &
QUALIFIERS                        ||
//
////////////////////////////////////////////////////////////////////


varying vec2 vUv; // The "coordinates" in UV mapping
representation
varying vec3 vPosition; // Vertex position

uniform float uTime; // Time in seconds since load
uniform float uFrame; // Frame number
uniform vec3 uResolution; // Canvas size (width,height)

// Texture channels from other buffers
uniform sampler2D iChannel0;
uniform sampler2D iChannel1;
uniform sampler2D iChannel2;

//////////////////////////////////////////////////////////////////////
////////////



//
////////////////////////////////////////////////////////////////////
//
||                            FUNCTIONS                             ||
//
////////////////////////////////////////////////////////////////////


float map(vec2 fragCoord) {
    return MAP_HEIGHT(texture(iChannel0,
fragCoord/uResolution.xy).z);
}


vec2 getVelocity(vec2 uv) {
    vec2 p = uv * MAP_RES;
    if (p.x < 0.5) p.x = 0.5;
```

```
    vec2 v = texture(iChannel1, p/uResolution.xy + PASS4).xy;
    if (length(v) > 1.) v = normalize(v);
    return v;
}


vec2 getPosition(vec2 fragCoord) {
    for(int i = -1; i <= 1; i++) {
        for(int j = -1; j <= 1; j++) {
            vec2 uv = (fragCoord + vec2(i,j)) / uResolution.xy;
            vec2 p = texture(iChannel2, fract(uv)).xy;
            if(p.x == 0.) {
                if (hash13(vec3(fragCoord + vec2(i,j), uFrame)) >
1e-4) continue;
                p = fragCoord + vec2(i,j) + hash21(float(uFrame))
- 0.5; // add particle
            } else if (hash13(vec3(fragCoord + vec2(i,j), uFrame))
< 8e-3) {
                continue; // remove particle
            }
            vec2 v = getVelocity(uv);
            p = p + v;
            p.x = mod(p.x, uResolution.x);
            if(abs(p.x - fragCoord.x) < 0.5 && abs(p.y -
fragCoord.y) < 0.5)
                return p;
        }
    }
    return vec2(0);
}


//
////////////////////////////////////////////////////////////////////
//
||                          MAIN                                ||
//
////////////////////////////////////////////////////////////////////


void main() {
    vec2 fragCoord = vUv * uResolution.xy;

    if (uFrame < 10.) {
        gl_FragColor = vec4(0);
        return;
    }

    vec4 c = texture(iChannel2, fragCoord/uResolution.xy);
    float particle = (c.x > 0.) ? 1. : 0.9 * c.y;
    vec2 p = getPosition(fragCoord);
    gl_FragColor.xy = (p == vec2(0)) ? vec2(0., particle) : p;
```

```
    vec2 uv = fragCoord/uResolution.xy;

    vec2 v = getVelocity(uv);
    vec2 vn = getVelocity(uv + N/uResolution.xy);
    vec2 ve = getVelocity(uv + E/uResolution.xy);
    vec2 vs = getVelocity(uv + S/uResolution.xy);
    vec2 vw = getVelocity(uv + W/uResolution.xy);
    float div = (ve - vw).x/2. + (vn - vs).y/2.;

    float height = map(fragCoord);
    float hn = map(fragCoord + N);
    float he = map(fragCoord + E);
    float hs = map(fragCoord + S);
    float hw = map(fragCoord + W);
    vec2 hgrad = vec2(he - hw, hn - hs)/2.;

    vec4 climate = texture(iChannel1, uv * MAP_RES /
uResolution.xy + PASS3);
    float mbar = climate.x;
    float temp = climate.y;
    c = texture(iChannel2, fract((fragCoord - v) /
uResolution.xy));

    // Water vapour advection
    float w = c.w;
    float noise = clamp(3. * FBM(vec3(5. *
fragCoord/uResolution.xy, uTime)) - 1., 0., 1.);
    if (uTime < OCEAN_END_TIME) w += 0.08 * noise * (1. -
smoothstep(OCEAN_START_TIME, OCEAN_END_TIME, uTime));

    // Evaporation
    if (height == 0.) w += noise * clamp(temp + 2., 0., 100.)/32.
* (0.075 - 3. * div - 0.0045 * (mbar - 1012.));

    // Precipitation
    w -= 0.005 * w;

    // Orographic lift
    w -= 0.3 * length(hgrad);

    gl_FragColor.w = clamp(w, 0., 3.);
}


`; export default fragmentShader;
```