Implementation of Kenny Mitchell's post-processing algorithm for volumetric light scattering effect

Computer Graphics and 3D
Professor Stefano Berretti

Davide Pucci
davide.pucci@stud.unifi.it

Introduction

- Under the right environmental circumstances, light occluding objects **cast volumes of shadow** and **appear to create rays of light**
- This phenomena is known as **crepuscular rays**

- In this work this effect is implemented through the usage of **Kenny Mitchell's post-processing algorithm** for volumetric light scattering

Analytic Model

- Mitchell's algorithm is based on a previous model for daylight scattering by **Hoffman and Preetham**, which allows to calculate the **illumination of each pixel** taking into account the scattering phenomena:

$$L(S,\theta) = L_0 e^{-\beta_{ex} S} + \frac{1}{\beta_{ex}} E_{sun} \beta_{sc}(\theta)(1 - e^{-\beta_{ex} S})$$

- The first term accounts for **light absorption** phenomena from the source to the viewpoint

- The second term calculates the additive amount due to **light scattering**

Analytic Model

$$L(S,\theta) = L_0 e^{-\beta_{ex} S} + \frac{1}{\beta_{ex}} E_{sun} \beta_{sc}(\theta)(1 - e^{-\beta_{ex} S})$$

In particular:

- L_0 : illumination of the light source
- S : distance traveled
- θ : angle between ray and the light source
- E_{sun} : irradiance of the light source
- β_{ex} : extinction constant, composed by absorption and out-scattering
- $\beta_{sc}()$: angular scattering term

→ This model can be **extended to account for occlusions**

Analytic Model

- **Occlusions** can be modeled as an **attenuation of the source of illumination**
- The resulting illumination at each pixel can be evaluated as:

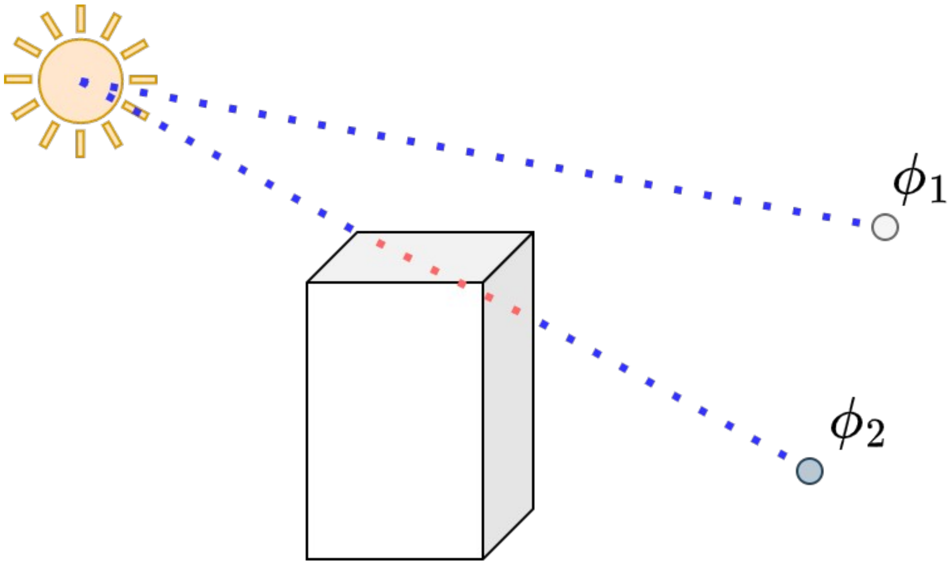
$$L(S,\theta,\phi) = (1 - D(\phi))L(S,\theta)$$

where:

  - $D(\phi)$: combined attenuated opacity of sun-occluding objects for the view location

- **Complication**: in order to determine the occlusion of the light source for every point we would need full volumetric information!
  - ➔ In screen space this is not available…
  - ➔ We can **estimate the probability of occlusions** summing samples along the ray!

Analytic Model



- The resulting model is:

$$L(S, \theta, \phi) = \sum_{i=0}^{n} \frac{L(S_i, \theta_i)}{n}$$

which can be parametrized in order to control the final effect

Analytic Model

- The following parametrization is proposed, in order to have full control over the summation:

$$L(S, \theta, \phi) = exposure \cdot \sum_{i=0}^{n} decay^i \cdot weight \cdot \frac{L(S_i, \theta_i)}{n}$$

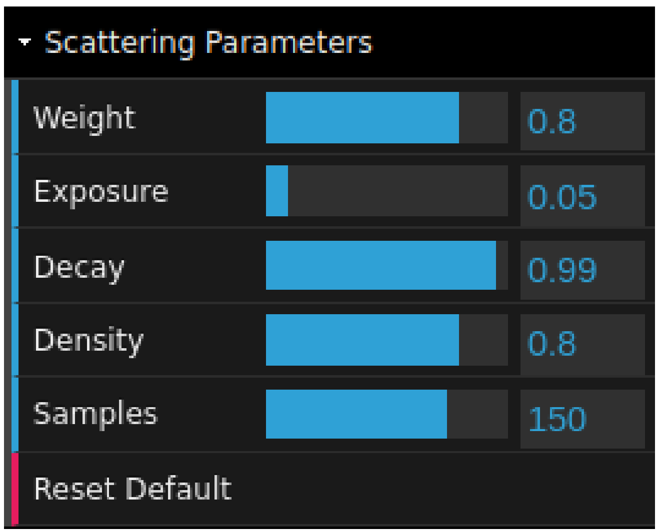
where:

  - **`exposure`**: controls the overall intensity of the post-processing
  - **`decay`**: dissipates sample contributions as these are further from the source
  - **`weight`**: controls the intensity of each sample

- Two parameters allow to control the number of samples (**`n`**) and their separation (**`density`**)

Implementation Note

- These 5 parameters can be controlled in the GUI:



Technologies

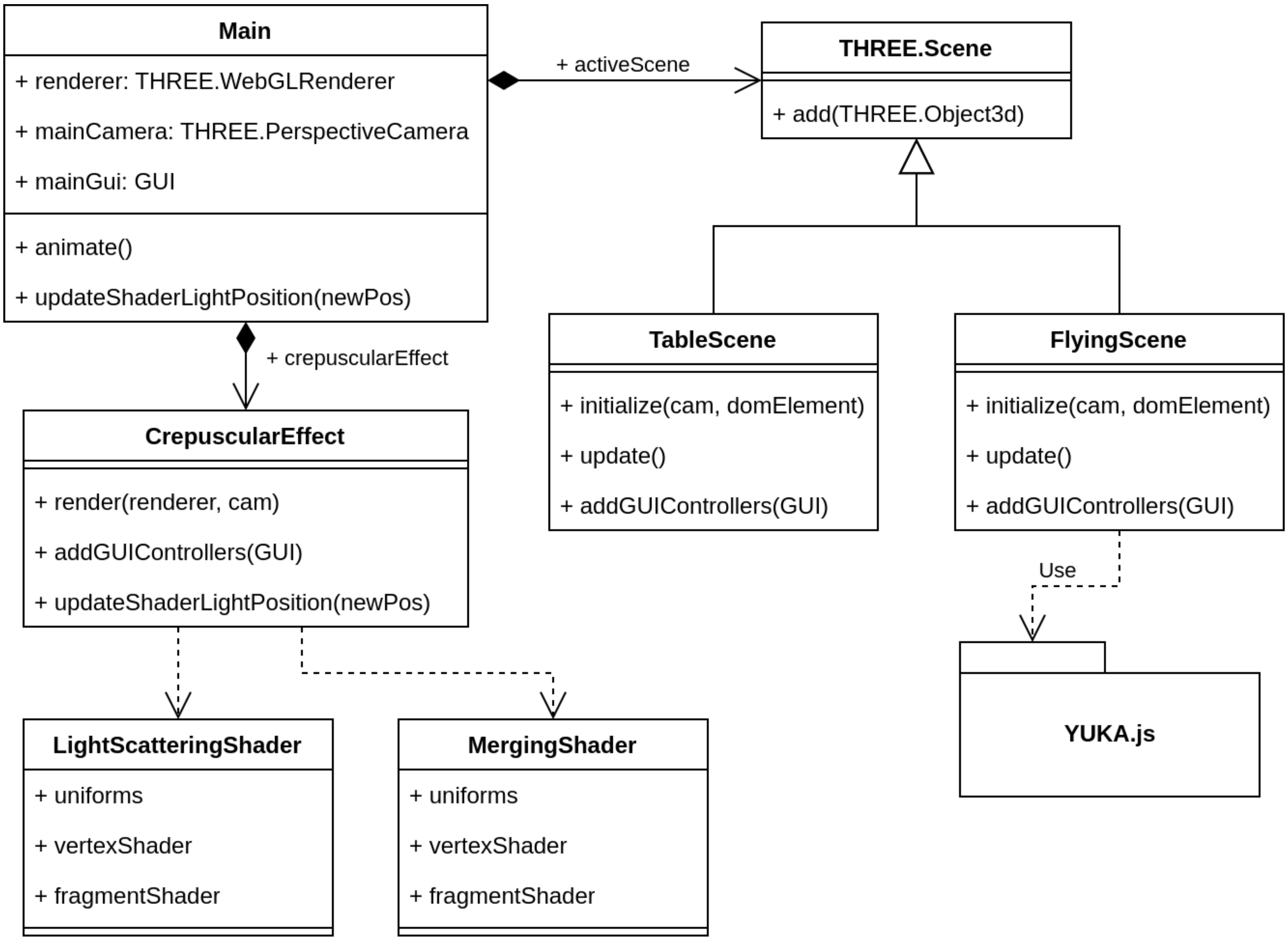
The following technologies were used in the implementation:

- `TypeScript`: a typed language built on top of `JavaScript`
- `three.js`: a cross-browser `JavaScript` library used to create and display 3D computer graphics using WebGL
- `Vite`: offers a local development server and a deployment tool for `TypeScript` applications

For the realization of the `FlyingScene` an extra library was used:

- `Yuka`: a `JavaScript` library for Game AI development

The Implementation



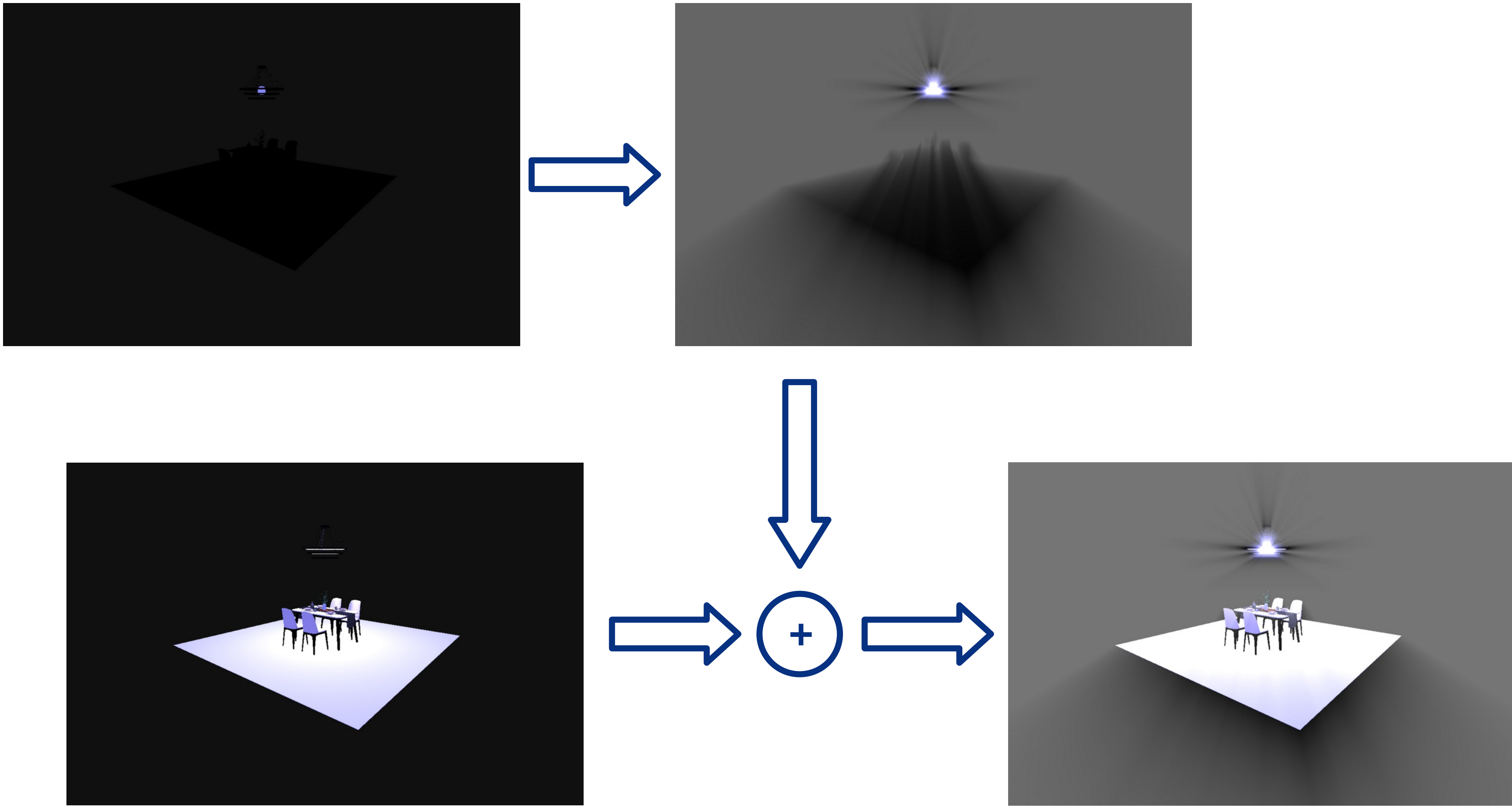
The Implementation

- The computation of the volumetric light scattering effect is sample-based in the screen space, hence **texture may cause undesired effects**

- To avoid so we can compute the post-processing algorithm on **black, untextured objects** and then **additively blend it** into the scene

- These steps can be performed in `three.js` through the usage of **`THREE.EffectComposers`** and **`THREE.Layers`**

The Implementation



The Implementation – CrepuscularEffect

- Two composers are needed to perform these steps.
  These are defined in **`CrepuscularEffect`** class:

  1. **`OcclusionComposer`**: renders the light scattering effect on untextured objects

  2. **`SceneComposer`**: merges the outcome of the scattering effect with the original scene, then renders to the screen

The Implementation – CrepuscularEffect

`OcclusionComposer`

```
// Target
let target = new WebGLRenderTarget(window.innerWidth / 2, window.innerHeight / 2, this.renderTargetParameters)

// Occlusion Composer
this.occlusionComposer = new EffectComposer(renderer, target);
this.occlusionComposer.addPass(new RenderPass(scene, camera));

// Scattering
let lightScatteringPass = new ShaderPass(LightScatteringShader);
this.scatteringUniforms = lightScatteringPass.uniforms;
this.occlusionComposer.addPass(lightScatteringPass);

// Copy Shader
this.occlusionComposer.addPass(new ShaderPass(CopyShader));
```

- As proposed by Mitchell, the scattering effect can be **rendered to a lower resolution** in order to speed-up the computation
- Uniforms are stored in the CrepuscolarEffect class in order to control these with the GUI

UNIVERSITÀ DEGLI STUDI FIRENZE

The Implementation – LightScatteringShader

- The following uniforms are defined for the shader, together with their default values:

```
uniforms: {
    tDiffuse: {value: null},
    lightPos: {value: new Vector2(0., 0.)},
    decay: {value: 0.99},
    density: {value: 0.8},
    weight: {value: 0.8},
    exposure: {value: 0.05},
    n_samples: {value: 150}
},
```

- The vertex shader is a *pass-through* that doesn't effect the scene geometry:

```
vertexShader: /* glsl */`

    varying vec2 vUv;

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

The Implementation – LightScatteringShader

```
fragmentShader: /* glsl */`

    uniform sampler2D tDiffuse;
    uniform vec2 lightPos;
    uniform float decay;
    uniform float density;
    uniform float exposure;
    uniform float weight;
    uniform int n_samples;

    varying vec2 vUv;

    void main() {
        vec2 ray = vUv - lightPos;
        vec2 delta = ray * (1. / float(n_samples)) * density;
        vec4 color = texture(tDiffuse, vUv);
        vec2 currentPos = vUv;
        float illuminationDecay = 1.;

        for (int i = 1; i < n_samples; ++i){
            currentPos -= delta;
            vec4 sampleColor = texture(tDiffuse, currentPos);

            sampleColor *= illuminationDecay * weight;
            color += sampleColor;

            illuminationDecay *= decay;
        }

        gl_FragColor = color * exposure;
    }`
```

- The fragment shader implements the volumetric light scattering algorithm as described by Mitchell
- The **ray** is defined as the difference of the current fragment and the light position
- **delta** is used to sample along the ray at regular intervals
- The **for()** loop implements the equation of slide 6:

$$L(S,\theta,\phi) = exposure \cdot \sum_{i=0}^{n} decay^i \cdot weight \cdot \frac{L(S_i,\theta_i)}{n}$$

The Implementation – CrepuscularEffect

`SceneComposer`

```
// Scene Composer
this.sceneComposer = new EffectComposer(renderer);
this.sceneComposer.addPass(new RenderPass(scene, camera));

// Merging Pass
let mergingPass = new ShaderPass(MergingShader);
mergingPass.uniforms.tOcclusion.value = target.texture;

this.sceneComposer.addPass(mergingPass);
```

- The outcome of the light scattering computation is passed as a uniform variable to the `MergingShader`, that is able to produce the final effect

The Implementation – MergingShader

- The following uniforms are defined for the shader:

```
uniforms: {
    tDiffuse: {value: null},
    tOcclusion: {value: null}
},
```

- The vertex shader is a *pass-through* as for the `LightScatteringShader`
- The fragment shader blends the two effects in the following way:

```
fragmentShader: /* glsl */`

    uniform sampler2D tDiffuse;
    uniform sampler2D tOcclusion;

    varying vec2 vUv;

    void main() {
        vec4 originalColor = texture(tDiffuse, vUv);
        vec4 scatteringColor = texture(tOcclusion, vUv);
        gl_FragColor = originalColor + scatteringColor;
    }`
```

The Implementation – Rendering

- **`Main.animate()`**

```
// LOOP
function animate() {
    requestAnimationFrame(animate);
    if(activeScene != undefined){
        activeScene.update();
        crepuscolarEffect.render(renderer, mainCamera);
    } else {
        // console.log("Scene is loading!");
    }
}
```

- **`CrepuscularEffect.render()`**

```
render(renderer: WebGLRenderer, camera: PerspectiveCamera,
        color: THREE.ColorRepresentation = "#101010"){

    camera.layers.set(OCCLUSION_LAYER);
    renderer.setClearColor(color)
    this.occlusionComposer.render();

    camera.layers.set(MAIN_LAYER);
    renderer.setClearColor("#000000")
    this.sceneComposer.render();

}
```

The Implementation – Light Position Update

- The light position is given in normalized device coordinates, we need to convert them to texture coordinates in order to use them as a uniform variable of the shader

- **`Main.updateShaderLightPosition()`**

```
function updateShaderLightPosition(newPos: THREE.Vector3) {
    crepuscularEffect.updateShaderLightPosition(mainCamera, newPos);
    crepuscularEffect.render(renderer, mainCamera);
}
```

- **`CrepuscularEffect.updateShaderLightPosition()`**

```
updateShaderLightPosition(camera: PerspectiveCamera, newPos: Vector3) {

    let screenPosition = newPos.project(camera);
    let newX = 0.5 * (screenPosition.x + 1);
    let newY = 0.5 * (screenPosition.y + 1);

    this.scatteringUniforms.lightPos.value.set(newX, newY);
}
```

The Implementation – FlyingScene

- In the **`FlyingScene`** two 3D models move in the scene with these behaviours:
  1. The **X-Wing** follows a fixed path
  2. The **TIE Fighter** chases the X-Wing

- This is achieved through the usage of **`Yuka`** JavaScript library, which allows to develop Game AI and is suitable to work together with **`three.js`**

- **Yuka's pipeline** is the following:
  1. Initialize Yuka's entity manager and time object:

```
this.entityManager = new YUKA.EntityManager();
this.time = new YUKA.Time();
```

  2. Define a 3D object and disable automatic matrix update:

```
// X-WING
const xwing = await this.loader.loadAsync(x_wing);
xwing.scene.matrixAutoUpdate = false;
this.add(xwing.scene);
```

The Implementation – FlyingScene

3. Create a **`YUKA.Vehicle`** and associate it to its render component, allowing Yuka to control its matrix computations:

```
this.xwingVehicle = new YUKA.Vehicle();
this.xwingVehicle.setRenderComponent(xwing.scene, this.sync);
this.xwingVehicle.maxSpeed = this.movementSettings.speed;
```

4. Set-up a **`YUKA.Path`** for the X-Wing:

```
const path = new YUKA.Path();

path.add(new YUKA.Vector3(0, 10, 0));
path.add(new YUKA.Vector3(50, 30, 100));

/* ... */

path.add(new YUKA.Vector3(-100, 30, 50));

path.loop = true;
```

The Implementation – FlyingScene

5. Create and assign the **`YUKA.Behavior`** of both vehicles:

```
this.xwingBehavior = new YUKA.FollowPathBehavior(path, 20);
this.tieBehavior = new YUKA.PursuitBehavior(this.xwingVehicle);
this.xwingVehicle.steering.add(this.xwingBehavior);
this.tieVehicle.steering.add(this.tieBehavior);
```

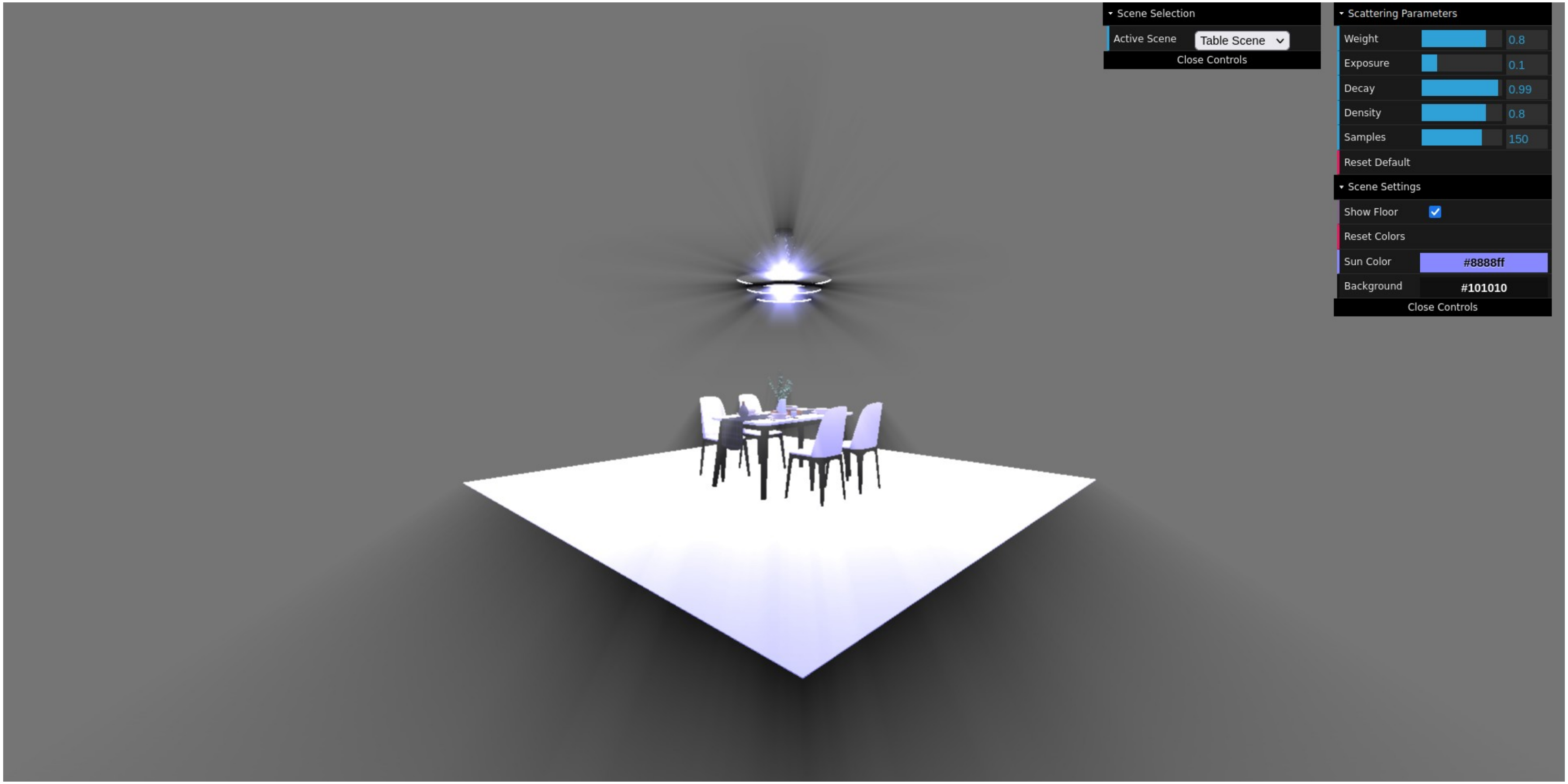
6. Add vehicles to the `EntityManager`:

```
this.entityManager.add(this.xwingVehicle);
this.entityManager.add(this.tieVehicle);
```

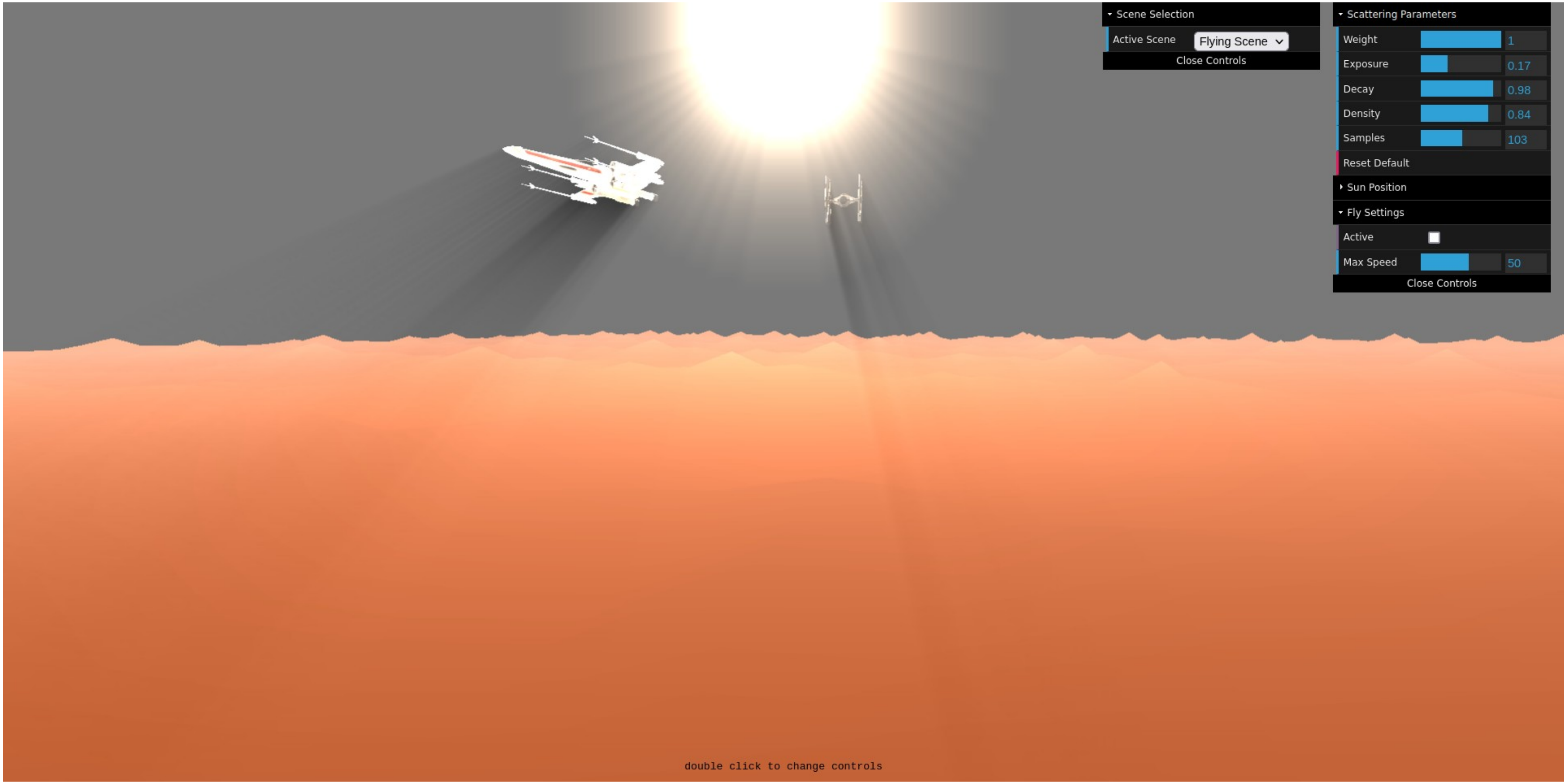
- Finally, in order to update the position of both 3D models when rendering, the **`FlyingScene.update()`** method contains:

```
let delta = this.time.update().getDelta();
this.entityManager.update(delta);
```

Demo Images – TableScene



Demo Images – FlyingScene



Live Demo

- Live Demo of the project is available here