IE500217 – Computer Graphics
Assignment 2 (Practical assignment 1)

Implementing a Ray Tracer in JavaScript

Due date: October 03, 2022, 23:59.

1 Objectives

The goal of this assignment is to create the first elements of a ray tracer in JavaScript based on the Ray Tracing in One Weekend book [1]. More precisely, we will follow Sections 1 to 9 of [1], covering:

- 3D vectors;
- Rays;
- Perspective cameras;
- Spheres;
- Antialiasing;
- Diffuse and metal materials;
- Ray reflection.

2 Requirements

Your development files should be submitted on Blackboard in a *.zip* file containing:

- src/
  - materials/
    - lambertian.js
    - metal.js
  - records/
    - hit-record.js
    - scatter-record.js
  - camera.js
  - image-displayer.js
  - main.js
  - ray.js
  - objects/
    - sphere.js
  - vec3.js
  - world.js
- index.html
- package-lock.json
- package.json
- styles.css

A template containing these files is available on Blackboard:
Learning materials/Assignments/Assignment 02/Template
When programming, you should use the clean code principles detailed in Lecture 3.

3 Detailed description

This chapter details the different exercises that compose the assignment. Each section refers to a specific chapter of [1], so it is recommended to read the chapters before beginning a new section. **The goal of each section is to produce a particular image**.

3.1 Setting up the project

This section covers Chapters 1 and 2 of [1]. The goal is to produce an image similar to Figure 1.

Figure 1: First image.

1. Download, extract, and open with VS Code the template:
   Learning materials/Assignments/Assignment 02/Template

2. Open the VS Code console, and type:

```
npm install
npm run dev
```

3. A web page should open with the image shown in Figure 1.

4. The JavaScript sources files are located in the *src* directory. They are the only files to be modified for this assignment.

5. *src/image-displayer.js* should not be modified. It contains the *displayImage(imageWidth, imageHeight, image)* function that renders your image to the webpage and whose parameters are:

   - *imageWidth*: width of the image in pixels;
   - *imageHeight*: height of the image in pixels;
   - *image*: list of pixels; each pixel is represented as a 3-element set of numbers between 0 and 1 corresponding to the red, green, and blue value of the pixel. The first pixel of the list corresponds to the top left pixel of the image, and pixels then proceed from left to right, then downward, throughout the list.

   An example of the use of this function is shown in *src/main.js*.

6. *src/main.js* contains the code that will be executed by the browser. Eight functions have been declared, each one corresponding to a specific section (for example, *firstImage()* corresponds to this section). *firstImage()* has already been written and does not require any change. However, for the following sections, you will have to write your code in the corresponding functions.

7. The other JavaScript files contain empty classes that will have to be filled out in this assignment.

3.2 Blue-white gradient

This section covers Chapters 3 and 4 of [1]. The goal is to produce an image similar to Figure 2.

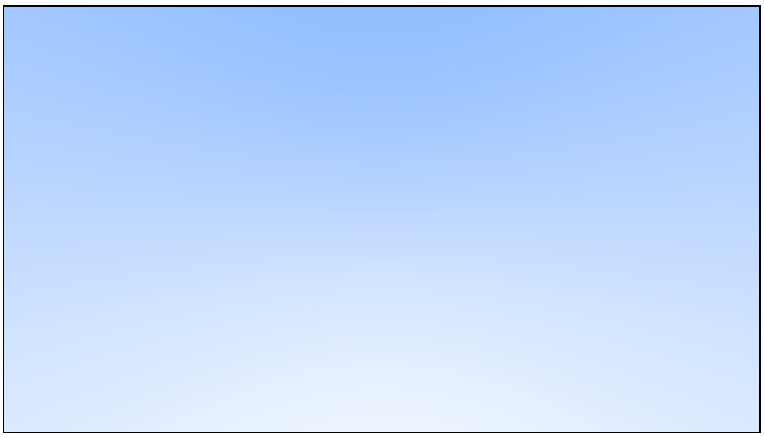

Figure 2: Blue-white gradient.

1. In the beginning of the *src/main.js* file, comment the "*firstImage();*" line and uncomment the "*blueWhiteGradient();*" line.
2. In the *src/vec3.js* file, the *Vec3* class is defined. This class represents a 3-dimensional vector and has x, y, and z attributes. Add the following functions to the *Vec3* class:
   - The *add(v)* function, that takes a parameter v of type *Vec3* and returns the sum of *this* and v as a new *Vec3*, has already been created. This function returns a new vector, and does not modify *this* or v. Use this function as an example for the implementation of others.
   - *subtract(v)*, same as *add(v)* but for the subtraction.
   - *multiply(f)*, that takes a parameter f of type *number* and returns the multiplication of *this* by f as a new *Vec3*. This function returns a new vector and does not modify *this*.
   - *multiplyByVector(v)*, that takes a parameter v of type *Vec3* and returns a new *Vec3* in which each component is equal to the multiplication of the corresponding components of *this* and v.
   - *squaredLength()*, that returns the squared length (or squared magnitude) of *this* as a *number*.
   - *length()*, that returns the length (or magnitude) of *this* as a *number*. You can use the *Math.sqrt(x)* function to determine the square root of a *number x*.
   - *unitVector()*, that returns the normalized vector of *this* as a new *Vec3*.
   - *dot(v)*, that takes a parameter v of type *Vec3* and returns the dot product of *this* and v as a *number*.
   - *toList()*, that returns a *list* containing the three components x, y, and z (in that order) of *this*.
   - *squareRoot()*, that returns a new *Vec3* containing the square root of the three components of *this*.
   - Leave the other functions empty for now.
     Usage example:

     ```
     const u = new Vec3(1, 1 ,1);
     const v = new Vec3(-2, 8, -9);
     const w = new Vec3(-6, 0 , 5);
     console.log(u.add(v).subtract(w).multiply(5).unitVector());
     // should show around { x: 0.30, y: 0.54, z: -0.78 }
     ```

3. In the *src/ray.js* file, the *Ray* class is defined. This class represents a ray that will be cast in the scene to detect the objects' position and color. It has an *origin* and a *direction* attribute. Define the *at(t)* function to the *Ray* class that takes a parameter t of type *number* and returns the point located at $this.origin + t \times this.direction$ (see **Listing 8** of [1]).
   Usage example:

```
const u = new Vec3(1, 1 ,1);
const v = new Vec3(-2, 8, -9);
const ray = new Ray(u, v);
console.log(ray.at(5));
// should show { x: -9, y: 41, z: -44 }
```

4. In the *src/main.js* file, add the following code to the *blueWhiteGradient()* function:

   - A function *rayToColor(ray)*, that takes a parameter *ray* of type *Ray* and returns the blue/white color corresponding to this ray as a *Vec3* (see the *ray_color()* function of **Listing 9** of [1]).
     Usage example:

     ```
     const u = new Vec3(1, 1 ,1);
     const v = new Vec3(-2, 8, -9);
     const ray = new Ray(u, v);
     console.log(rayToColor(ray));
     // should show around { x: 0.59, y: 0.75, z: 1 }
     ```

   - The code that renders the blue-white gradient of Figure 2. Use the code of the *firstImage()* function as a basis, and adapt the *main()* function of **Listing 9** of [1]). You can use the *toList()* function of *Vec3* to convert the color given by *rayToColor()* for the *displayImage()* function (*displayImage()* only accepts a list of list (or matrix) of numbers and not a list of *Vec3*).

3.3 Red sphere

This section covers Chapter 5 of [1]. The goal is to produce an image similar to Figure 3.

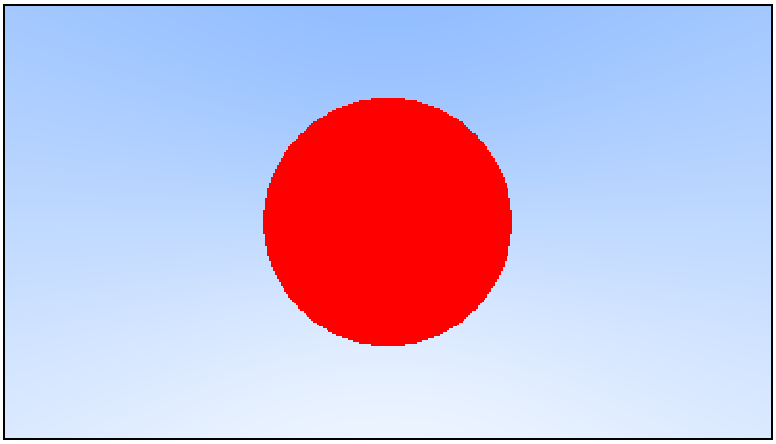

Figure 3: Red sphere.

1. In the beginning of the *src/main.js* file, comment the "*blueWhiteGradient();*" line and uncomment the "*redSphere();*" line.

2. In the *src/main.js* file, add the following code to the *redSphere()* function:

   - The code of the *blueWhiteGradient()* function.
   - A function *hitSphere(center, radius, ray)* that takes a parameter *center* of type *Vec3*, *radius* of type *number*, *ray* of type *Ray*, and returns a *boolean* indicating if the *ray* intersects with the sphere defined by *center* and *radius* (see **Listing 10** of [1]).
     Usage example:

     ```
     const u = new Vec3(1, 1 ,1);
     const v = new Vec3(-2, 8, -9);
     const ray = new Ray(u, v);
     const center = new Vec3(1, 2, 1);
     const radius = 5;
     console.log(hitSphere(center, radius, ray));
     // should show true
     ```

   - Adapt the *rayToColor()* function so that if the ray intersects with the sphere of center $(0, 0, -1)$ and radius 0.5, the color is red (see **Listing 10** of [1]).

3.4 Normals sphere

This section covers Section 6.1 of [1]. The goal is to produce an image similar to Figure 4.

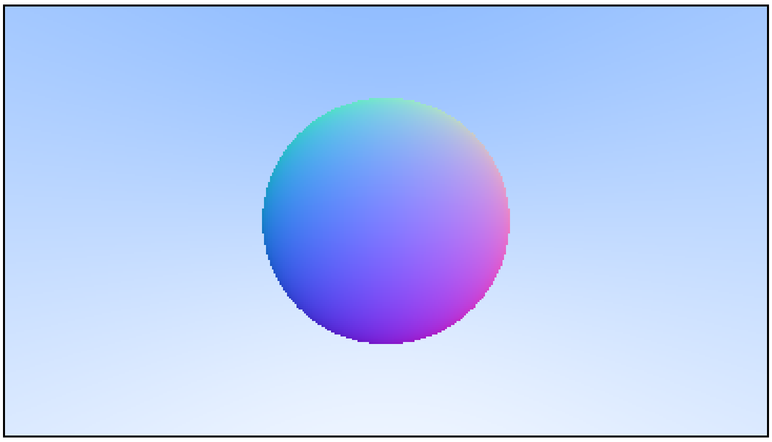

Figure 4: Normals sphere.

1. In the beginning of the *src/main.js* file, comment the "*redSphere();*" line and uncomment the "*normals-Sphere();*" line.

2. In the *src/main.js* file, add the following code to the *normalsSphere()* function:

   - The code of the *redSphere()* function.
   - Adapt the *hitSphere()* function so that it returns the value of t if the sphere is hit, and -1 else (see **Listing 11** of [1]).
     Usage example:

     ```
     const u = new Vec3(1, 1, 1);
     const v = new Vec3(-2, 8, -9);
     const ray = new Ray(u, v) ;
     const center = new Vec3(1, 2, 1);
     const radius = 5;
     console.log(hitSphere(center, radius, ray));
     // should show around -0.35
     ```

   - Adapt the *rayToColor()* function so that it colors the sphere according to its normal vectors (see **Listing 11** of [1]).

3.5 Sphere and ground

This section covers Sections 6.2 to 6.7 of [1]. The goal is to produce an image similar to Figure 5.

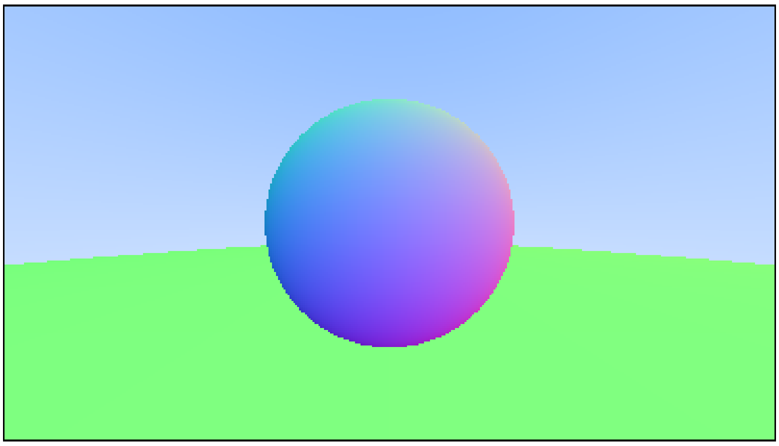

Figure 5: Sphere and ground.

1. In the beginning of the *src/main.js* file, comment the "*normalsSphere();*" line and uncomment the "*sphereAndGround();*" line.
2. In the *src/main.js* file, add the following code to the *sphereAndGround()* function:
   - The code of the *normalsSphere()* function.
   - Adapt the *hitSphere()* function with the simplification proposed in **Listing 13** of [1].
3. The *src/records/hit-record.js* file contains the definition of the *HitRecord* class. The class contains information about a ray hit with an object, such as:
   - The hit *point* of type *Vec3*.
   - The *normal* of the object at the hit point of type *Vec3*.
   - t such as $ray_{origin} + t \times ray_{direction} = hitPoint$ of type *number*.
   - The *material* of the object (that will be used later in the assignment).
   - Whether a *hit* occurred or not of type *boolean*.
4. In the *src/objects/sphere.js* file, the *Sphere* class represents a sphere object. It has a *center* of type *Vec3*, *radius* of type *Vec3* and *material* attribute (*material* will be used later in the assignment). In the *Sphere* class, define the *hit(ray, tMin, tMax)* function that takes a parameter *ray* of type *Ray*, *tMin* of type *number*, *tMax* of type *number*, and returns an object of type *HitRecord* containing information about the hit (see **Listing 15** of [1]). This function determines if there exists a t such as $t > tMin$, $t < tMax$, and $ray_{origin} + t \times ray_{direction}$ intersects with the sphere. There is a slight difference with **Listing 15** of [1], as you should not return a *boolean* but a *HitRecord*. The *HitRecord* class contains a *hit* property (*boolean)* that tells whether a hit has occurred or not. This function is similar to the *hitSphere()* function of the *src/main.js* file.
   Usage example:

```
const sphere = new Sphere(new Vec3(0, 0, -1), 0.5);
const ray = new Ray(new Vec3(0, 0, 0), new Vec3(0, 0, -0.5));
console.log(sphere.hit(ray, 0, Infinity));
/* should show:
Object {
    hit: true,
    material: Object {},
    normal: Object { x: 0, y: 0, z: 1 },
    point: Object { x: 0, y: 0, z: -0.5 },
    t: 1
}
*/
```

5. In the *src/records/hit-record.js* file, define the function *setFaceNormal(ray, outwardNormal)* that takes a parameter *ray* of type *Ray*, *outwardNormal* of type *Vec3*, and that sets the *normal* property of the object (see **Listing 18** of [1]).
   Hint: The code

   ```
   normal = front_face ? outward_normal :-outward_normal;
   ```

   of [1] has the same meaning as:

   ```
   if (front_face) {
       normal = outward_normal;
   } else {
       normal = -outward_normal;
   }
   ```

6. In the *src/objects/sphere.js* file, edit the *hit()* function so that it uses *setFaceNormal()* to the *HitRecord* returned by the function (see **Listing 19** of [1]).

7. In the *src/world.js* file, the *World* class represents the environment of the scene. It contains an *objects* attribute of type *list* which contains all the objects (spheres) of our scene. In this class, define the *hit(ray, tMin, tMax)* function that takes a parameter *ray* of type *Ray*, *tMin* of type *number*, *tMax* of type *number*, and that returns a *HitRecord* containing information about the hit of the ray with any of the object that *World* contains (see **Listing 20** of [1]). As for the *hit()* function of *Sphere*, you should not return a *boolean* but a *HitRecord*.

8. In the *src/main.js* file, add the following code to the *sphereAndGround()* function:
   - Remove the *hitSphere()* function.
   - Edit the *rayToColor()* function so that it takes an additional parameter *world* of type *World* and use the *hit()* function of *world* (see **Listing 24** of [1]).
     Hint: the JavaScript keyword *Infinity* can be used to represent an infinity *number*.
   - At the beginning of the *sphereAndGround()* function, define a new variable *world* of type *World* and add the following objects to it (with the *world.add()* function):
     - A *Sphere* of center (0, 0, -1) and radius 0.5.
     - A *Sphere* of center (0, -100.5, -1) and radius 100.

     (see **Listing 24** of [1]).
   - In the rendering loop, change the call from *rayToColor(ray)* to *rayToColor(ray, world)*.

3.6 Antialiasing

This Section covers chapter 7 of [1]. The goal is to produce an image similar to Figure 6 (zoom in to the edges of the sphere to see the difference with the previous image).

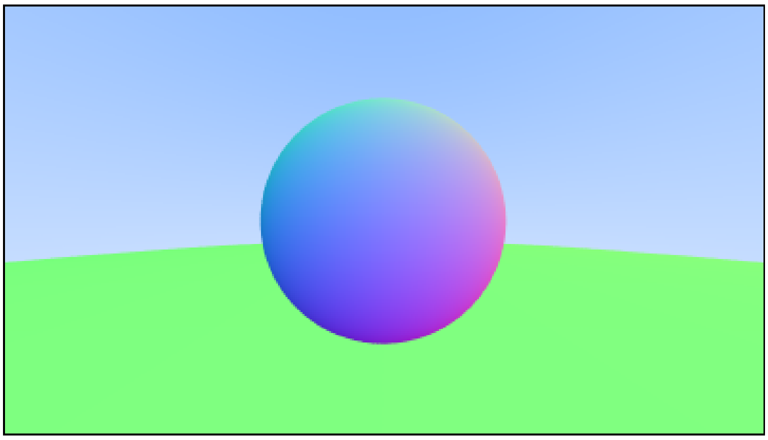

Figure 6: Antialiasing.

1. In the beginning of the *src/main.js* file, comment the "*sphereAndGround();*" line and uncomment the "*antialiasing();*" line.

2. In the *src/camera.js* file, the *Camera* class defines the parameters of the camera that were previously defined in *src/main.js*. In the *Camera* class, define the *getRay(u, v)* function that takes a parameter *u* of type *number*, *v* of type *number*, and that returns the *Ray* corresponding to the *(u,v)* coordinates (see **Listing 27** of [1]).
   Usage example:

```
const cam = new Camera();
console.log(cam.getRay(0.3, 0.6));
/* should show around:
Object {
    direction: Object { x: -0.71, y: 0.20, z: -1 },
    origin: Object { x: 0, y: 0, z: 0 }
}
*/
```

3. In the *src/main.js* file, add the following code to the *antialiasing()* function:

   - The code of the *sphereAndGround()* function.
   - Define and assign the *samplesPerPixel* variable to 100. This is the number of rays that will be sent for each pixel (see **Listing 30** of [1]).
   - Remove the variables previously used for the camera, and declare a new *Camera* instead (see **Listing 30** of [1]).
   - In the for loop determining the image, instead of sending one ray per pixel, send *samplesPerPixel* rays per pixel and define the color as the average value (see **Listing 30** of [1]). You will have to determine a random number between 0 and 1, so you can use the *Math.random()* function for this. You should also use the *getRay(u, v)* function of *camera*. Don't forget to multiply the pixel color by $1/samplesPerPixel$ (see **Listing 29** of [1]).

3.7 Diffuse sphere

This section covers Chapter 8 of [1]. The goal is to produce an image similar to Figure 7.

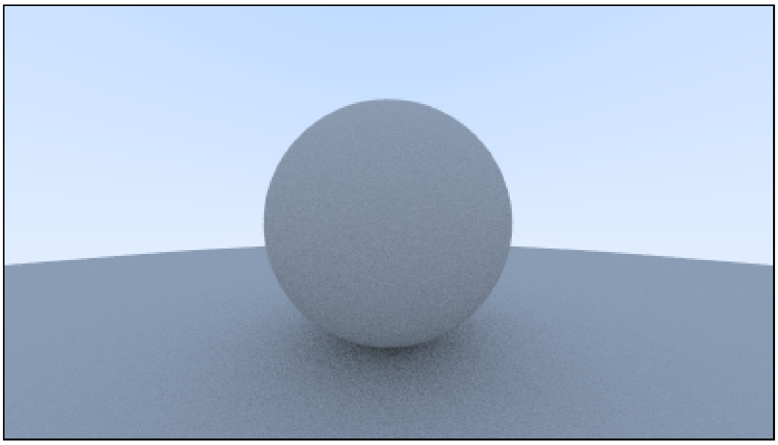

Figure 7: Diffuse sphere.

1. In the beginning of the *src/main.js* file, comment the "*antialiasing();*" line and uncomment the "*diffuseSphere();*" line.
2. In the *src/vec3.js* file, define the following functions:
   - *random(min, max)* that takes a parameter *min* of type *number*, *max* of type *number*, and that returns a new *Vec3* with each component being a random number between *min* and *max* (see **Listing 25** and **Listing 31** of [1]). This is a *static* function, which means you can call it like this:

     ```
     const vector = Vec3.random(-1, 1);
     ```

   - *randomInUnitSphere()* that returns a random point of type *Vec3* in a unit radius sphere (see **Listing 32** of [1]). This is a *static* function, which means you can call it like this:

     ```
     const vector = Vec3.randomInUnitSphere();
     ```

3. In the *src/main.js* file, add the following code to the *diffuseSphere()* function:
   - The code of the *antialiasing()* function.
   - Edit the *rayToColor()* function to use the new random direction generator (see **Listing 33** of [1]).
   - Edit the *rayToColor()* function to add a new parameter *depth* of type *number* that limits the maximum recursion depth (see **Listing 34** of [1]).
   - Add a *maxDepth* variable initialized to 50 and edit the call to *rayToColor()* to add this parameter (see **Listing 34** of [1]).
   - Perform the gamma 2 correction to each pixel color (see **Listing 35** of [1]). You can take the square root of the pixel color just after having multiplied it by $1/samplesPerPixel$ in the rendering loop.
   - Edit the *rayToColor()* function to ignore hits very near zero (see **Listing 36** of [1]).
   - Edit the *rayToColor()* function to use the normalized vector of *randomInUnitSphere()* (see **Listing 37** and **Listing 38** of [1]).

3.8 Metal spheres

This section covers Chapter 9 of [1]. The goal is to produce an image similar to Figure 8.

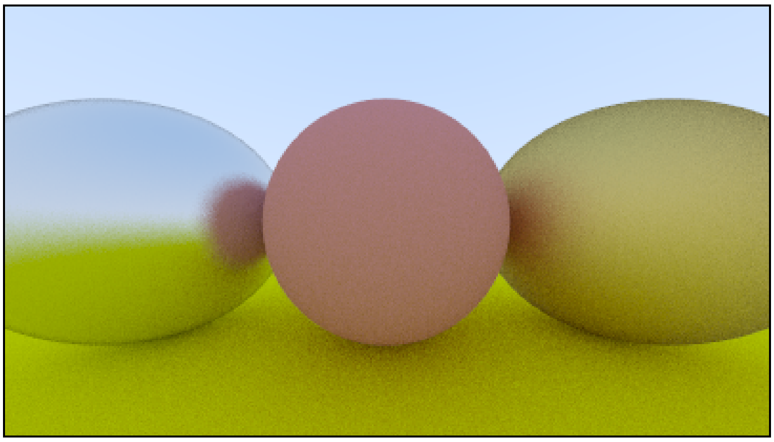

Figure 8: Metal spheres

1. In the beginning of the *src/main.js* file, comment the *diffuseSphere();* line and uncomment the *metalSpheres();* line.

2. In the *hit()* function of the *src/objects/sphere.js* file, store *this.material* to the returned *HitRecord* object (see **Listing 43** of [1]).

3. The *src/records/scatter-record.js* file contains the definition of the *ScatterRecord* class. This class contains information about how a ray scattered, such as:

    - The *scattered ray* of type *Ray*;
    - The *attenuation* color of type *Vec3*;
    - Whether a scatter has occurred or not (*isScattered*) of type *boolean*.

4. In the *src/materials/lambertian.js* file, the *Lambertian* class represents a diffuse material. It has an *albedo* attribute of type *Vec3*. In this class, define the *scatter(ray, rec)* function that takes a parameter *ray* of type *Ray*, *rec* of type *HitRecord*, and returns an object of type *ScatterRecord* containing information about the scattered ray (see **Listing 44** of [1]). There is a slight difference with **Listing 44** of [1], as you should not return a *boolean* but a *ScatterRecord*. The *ScatterRecord* class contains a *isScattered* attribute that tells whether a scatter has occured or not. Notice that the *ray* parameter of the *scatter(ray, rec)* function is not used, but you should keep it so that the function is consistent with the *Metal* class (which we are going to see in a few steps).

5. In the *src/vec3.js* file, add the *nearZero()* function that returns true if the vector is very close to zero in all dimensions (see **Listing 45** of [1]). You can use the *Math.abs(x)* function to get the absolute value of *x*.

6. In the *src/materials/lambertian.js* file, edit the *scatter()* function to catch degenerate scatter directions using the *Vec3.nearZero()* function (see **Listing 46** of [1]).

7. In the *src/vec3.js* file, add the *reflect()* function that takes a parameter *n* of type *Vec3* and returns the reflection of *this* with a normal vector *n* (see **Listing 47** of [1]).

8. In the *src/materials/metal.js* file, the *Metal* class represents a metal material. It has an *albedo* of type *Vec3* and a *fuzz* of type *number* attribute (more on the *fuzz* attribute in a few steps). In this class, define the *scatter(ray, rec)* function that takes a parameter *ray* of type *Ray*, a parameter *rec* of type *HitRecord*, and that returns an object of type *ScatterRecord* containing information about the scattered ray (see **Listing 48** of [1]). There is a slight difference with **Listing 48** of [1], as you should not return a *boolean* but a *ScatterRecord*. The *ScatterRecord* class contains a *isScattered* attribute that tells whether a scatter has occured or not.

9. In the *src/main.js* file, add the following code to the *metalSpheres()* function:

    - The code of the *diffuseSphere()* function.

- Edit the *rayToColor()* function so that the scattered rays are taken into account (see **Listing 49** of [1]).
- Remove the creation of the two spheres (in the world section), and instead add to *world*:
  - A *Sphere* of center (0, -100.5, -1), of radius 100 and of material a *Lambertian* of color (0.8, 0.8, 0).
  - A *Sphere* of center (0, 0, -1), of radius 0.5 and of material a *Lambertian* of color (0.7, 0.3, 0.3).
  - A *Sphere* of center (-1, 0, -1), of radius 0.5 and of material a *Metal* of color (0.8, 0.8, 0.8).
  - A *Sphere* of center (1, 0, -1), of radius 0.5 and of material a *Metal* of color (0.8, 0.6, 0.2).

  (see **Listing 50** of [1]).

10. In the *src/materials/metal.js* file, edit the *scatter()* function so that it takes the fuzziness parameter into account (see **Listing 51** of [1]).

11. In the *metalSpheres()* function of the *src/main.js* file, add a fuzziness of 0.3 to the *Sphere* centered in (-1, 0, -1), and a fuzziness of 1 to the *Sphere* centered in (1, 0, -1) (see **Listing 52** of [1]).

3.9 Dielectrics, positionable camera, and defocus blur (optional)

If you want, you can cover the rest of [1]. It consists of adding the dielectric material (that reflects and refracts ray) and additional features for the camera.

References

[1] P. Shirley. Ray tracing in one weekend, December 2020. `https://raytracing.github.io/books/RayTracingInOneWeekend.html`.