CG1 WS 23/24 - Exercise 5: Ray Tracing

Technische Universität Berlin - Computer Graphics
Date 18. January 2024 **Deadline** 31. January 2024
Prof. Dr. Marc Alexa, Tobias Djuren, Hendrik Meyer, Markus Worchel

Ray Tracing (6 points)

In this exercise, you are going to implement a Ray Tracer using Three.js. The ray tracer is implemented without WebGL. We only use WebGL for the preview window on the right. Therefore rendering happens on the CPU and is remarkably slower than WebGL rendering. *Hint:* You can decrease the rendering time significantly by reducing the size (width/height) of the rendering as well as disabling supersampling. The tasks in detail are:

1. (Basics) Divide the application into two windows with equal size and add the gui (`Settings` and `createGUI` are provided in `helper.ts`). On the right side setup the scene using `setupCamera`, `setupGeometry`, `setupLight` and `setupControls` in `helper.ts`. On the left show a `CanvasWidget` with the width and height given in the gui. When the save button is pressed the current canvas should be downloadable.

   We provide the `CanvasWidget` class in `canvasWidget.ts` to display the rendering. You have to pass a window instance to the constructor of `CanvasWidget` to add the widget to a window. You have to pass an initial width and height as well. You can change the dimensions whenever the gui update by calling `changeDimensions`. The save functionality is also provided by `savePNG`.

   Change the geometry, materials and light in SceneController for your solution so it looks different from the skeleton. *(0.5 points)*

2. (Raycasting) Implement a basic ray tracer where the color is simply the material color of the intersected object. You have to trace a ray from the camera through every pixel. The ray is intersected with the geometry using `Three.js`. The value of the pixel is the color of the material. At this point, no light has to be taken into account. Intersections can be calculated using `intersectObject(s)` of the `Raycaster` class. Render an image of the result with size 256px by 256px and include it in your submission (ex5_task2.png). (*1 point*)

   You can draw on the `CanvasWidget` using the function `setPixel` or `setImageData`.

3. (Intersection) The spheres in the scene are poorly sampled. But as we do not rely on the perspective transformation of vertices anymore we can use parametric geometry definitions instead of triangulated geometry. When the *Correct Spheres* checkbox in the gui is checked, use an updated intersection calculation for the sphere geometries. A correct ray sphere intersection should be calculated instead of the triangle ray intersection used by `intersectObject(s)`. For the other models `intersectObject(s)` can still be used. Render an image (256px by 256px) of the result and include it in your submission (ex5_task3.png). (*1 points*)

   Hint: Given a mesh with a `SphereGeometry` the origin of the sphere is given in `Mesh.position` and the radius is given in `Mesh.geometry.parameters.radius`.

4. (Illumination) Extend the Ray Tracer by evaluating the (Blinn-)Phong shading model **with light attenuation** at each point of intersection with **one light source**, whenever *Phong* is enabled in the gui. Take the `color` and `intensity` defined in the `PointLight` as well as `color`, `specular` and `shininess` defined in the `Material` into account. Alternatively, you may also use the Cook-Torrance model and extend the `Material` by an `roughness` value or reuse the `shininess` parameter with an adjusted value between 0 and 1. Render an image (256px by 256px) of the result and include it in your submission (ex5_task4.png). (*1.5 points*)

   Hints:

- To get the correct normals you can use the face normal of the intersection and transform them into world space for triangles, and the normalized vector between sphere origin and intersection point for the sphere geometries. All these information can be retrieved from the `Intersection` object in `Three.js` (if you implemented your custom sphere intersection nicely).
- Compute light attenuation before normalizing the light vector.
- We have no ambient light.
- The parameters for the shading model in `Three.js` are a bit different defined than our previous ones: *Diffuse light intensity* and *specular light intensity* are the same and are defined by `PointLight.color` scaled by `PointLight.intensity`. *Magnitude* is `shininess`. The *reflectance* of every material is defined as color and distinguished between the diffuse (`MeshPhongMaterial.color`) and the specular (`MeshPhongMaterial.specular`) one.
- As the `Three.js` Phong calculation is a bit different than our so the example solution needed to multiply the light intensity by 4 and the specular light component by the shininess divided by 50 to get comparable results.

Optional: So far we have only considered one light source. It is straightforward to include the other light sources by looping over all the lights and taking the sum of the colour contributions instead.

5. (Shadows) Raytracing allows for easy shadow calculation. Extend your illumination calculation by adding shadow checks, whenever *Shadows* is enabled in the gui. Do so by checking if some object is between the considered light source and the considered surface point. Render an image (256px by 256px) of the result and include it in your submission (ex5_task5.png). (*0.5 points*)

6. (Mirrors) Extend the ray tracer by taking mirrors into account whenever *Use Mirrors* is enabled in the gui. If a ray hits a reflective surface (the `Material` of the intersected object has the property `mirror` set to `true`), a new ray should be spawned from the intersection point to the proper reflected direction. The resulting color should be weighted between the reflection color and the Phong illumination color using the value of the `reflectivity` property in the `Material`. This is best implemented as recursion. The maximum recursion depth should be changed through a slider in the interface. Render an image and include it in your submission (ex5_task6.png). (*1 point*)

7. (Supersampling) To avoid aliasing implement the possibility of supersampling. That means shoot multiple rays per pixel and average the computed colors. The root value of the number of rays should be adjustable in the interface (2 means $2^2 = 4$ rays). The additional rays within a pixel should be shoot on a grid. Render an image with supersampling on and include it in your submission (ex5_task7.png). (*0.5 point*)

Requirements

- Exercises must be completed individually. Plagiarism will lead to exclusion from the course.
- Submit a `.zip` file of the `src` folder of your solution through ISIS by **31. January 2024, 23:59**.
- *Naming convention*: {firstname}_{lastname}_cg1_ex{#}.zip (for example: jane_doe_cg1_ex5.zip).
- You only hand in your `src` folder, make sure your code works with the rest of the provided skeleton.
- In addition to the source code, you must also submit the 6 ex5_task{#}.png files.