Interactive Computer Graphics
Coursework – Task 5 (*assessed)

December 2, 2023

Task 5: GPU ray tracing

Task 5a: Simple GPU ray tracing *

In this exercise you will implement a very simple ray tracer. Since the render to texture shader as used in Task 5 provides already a rather static camera setup for rendering a screen aligned textured quad, you can use this camera also to virtually shoot rays into a scene. However, since efficient ray tracing usually requires space partitioning for polyhedral geometry, we simplify the test scene in this exercise to objects that can be described mathematically: **planes** and **spheres**. You can define them by adding struts for these objects to your R2T fragment shader

```
struct Sphere
{
  vec3 centre;
  float radius;
  vec3 colour;
};

struct Plane
{
  vec3 point;
  vec3 normal;
  vec3 colour;
};
```

and in the main() function of the R2T fragment shader you should simply hard code them with

```
  sphere[0].centre = vec3(-2.0, 1.5, -3.5);
  sphere[0].radius = 1.5;
  sphere[0].colour = vec3(0.8,0.8,0.8);
  sphere[1].centre = vec3(-0.5, 0.0, -2.0);
  sphere[1].radius = 0.6;
  sphere[1].colour = vec3(0.3,0.8,0.3);
  sphere[2].centre = vec3(1.0, 0.7, -2.2);
```

```
  sphere[2].radius = 0.8;
  sphere[2].colour = vec3(0.3,0.8,0.8);
  sphere[3].centre = vec3(0.7, -0.3, -1.2);
  sphere[3].radius = 0.2;
  sphere[3].colour = vec3(0.8,0.8,0.3);
  sphere[4].centre = vec3(-0.7, -0.3, -1.2);
  sphere[4].radius = 0.2;
  sphere[4].colour = vec3(0.8,0.3,0.3);
  sphere[5].centre = vec3(0.2, -0.2, -1.2);
  sphere[5].radius = 0.3;
  sphere[5].colour = vec3(0.8,0.3,0.8);
  plane.point = vec3(0,-0.5, 0);
  plane.normal = vec3(0, 1.0, 0);
  plane.colour = vec3(1, 1, 1);
  //scene definition end
```

Note that you will ignore the per-polygon shaders in this exercise and that they will become useless in the pipeline. You can therefore deactivate them.

The first step to build a working raytracing pipeline is to adjust your R2T vertex shader to produce a fixes aspect ratio so that every fragment in the R2T fragment shader will correspond to a pixel where you can start a ray. The aspect ratio is defined by the viewing directions at the vertices of the render quad. An example vertex shader to achieve this may look like this:

```
#version 300 es

// Vertex coordinates in object space for the render quad
in vec3 vertexPosition;

uniform float canvasWidth;
uniform float canvasHeight;

uniform vec3 cameraPosition;
uniform mat3 cameraRotation;

uniform bool isOrthographicProjection;
```

```
uniform float orthographicFOV;
uniform float perspectiveFOV;

out vec3 origin;
out vec3 dir;

void main() {
  float aspectRatio = canvasWidth/canvasHeight;
  vec3 origin_camSpace, dir_camSpace;

  if (isOrthographicProjection) {
    origin_camSpace = vec3(vertexPosition.x*orthographicFOV*aspectRatio,
                           vertexPosition.y*orthographicFOV,
                           0);
    dir_camSpace = vec3(0, 0, -1);
  }
  else { // perspective projection
    origin_camSpace = vec3(0);
    dir_camSpace = vec3(vertexPosition.x*aspectRatio,
                        vertexPosition.y,
                        -1.0/tan(radians(perspectiveFOV)));
  }

  origin = cameraPosition + cameraRotation*origin_camSpace;
  dir = normalize(cameraRotation*dir_camSpace);

  gl_Position = vec4(vertexPosition, 1.0);
}
}
```

Note that this vertex shader passes ray origin and direction as `out` variables to the fragment shader, which will correspond to correctly interpolated ray origins and ray directions in world space in the fragment shader. Hence, you will need to capture them in the fragment shader with corresponding `in` variables.

Your task is to implement the ray tracing algorithms in the R2T Fragment Shader. Therefore, you will need to follow every ray until it reaches a

maximum ray tracing depth. This value could be for example 42. For this you will need to implement the base ray tracing loop as a `for` loop in the main function. You may want to define structs to describe intersections and rays and additional functions to intersect a ray with a sphere and plane and a function to handle shadow computation.

For simple ray tracing you may test every ray for an intersection with every object in the scene until the ray does not hit any of the objects or until it reaches the maximum ray-trace depth. Objects can be defined by using a mathematical definition of spheres and a plane as shown above. For the submission you should use these.

You should also compute shadows by using additional **shadow rays**. You can do this calculation in a separate function. When computing shadow rays, you should slightly move the ray origin outwards of the object along the surface normal or alter the ray direction slightly using a pseudo random number generator (Note, there is not proper real number generator in glsl, you must use pseudo-random functions, which you can find on the internet with for example the ray origin as seed) .

The plane intersection should additionally vary the ray hit color, so that a checkerboard pattern results.

When your ray tracer is ready you should be able to produce an image similar to Figure 1.

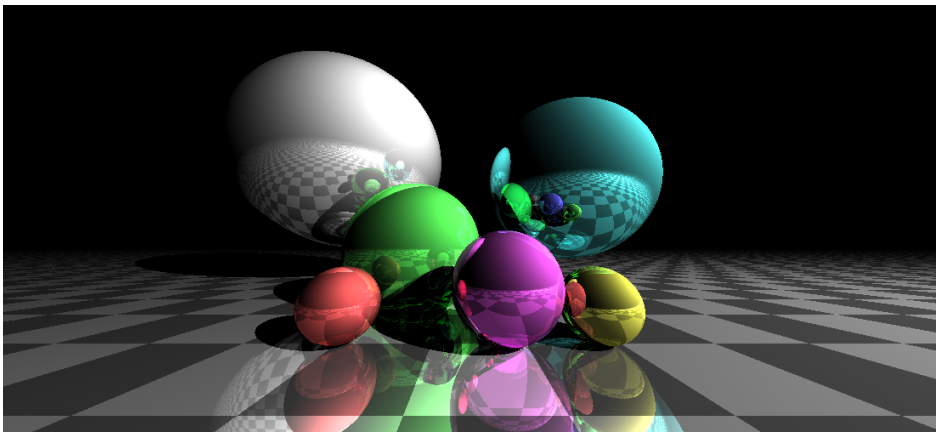

Figure 1: Example result from simple geometric ray tracing. Camera at (0, -0.003, 0.022), light source at (6, 4, 3).

You should also implement mouse-based scene interaction as you have been using it throughout the exercises. You can do this with `uniform` matrices attached to the model-view and projection matrices. These matrices will change when you use the mouse interaction scheme from as used in the framework. You may use them to **either** manipulate the initial ray direction **or** to alter the object's position. Think about the smarter option and make the right choice!

If you implemented the above described simple ray-tracer with shadow-rays and mouse based interaction successfully you will gain 75% of the maximum points for task 6. The remaining 25% can be achieved by extending this exercise with your own ray-tracing extensions. For example you could implement transparent and refracting objects, light scattering effects, caustics, soft shadows, and many more. It is up to you which extension you choose! Just be sure to **document which effects you implemented** and how to control them at the top of the R2T.

As well as submitting the jsons, please also add a screenshot showing off your work (as hi-res as possible, 6b if implemented otherwise from 6a) as a png file to your submission. Submissions will be ranked, the top three will be named and will likely get a small prize. Examples of cool submissions from previous years are on the course website.

Task 5b: Example extension: Soft Shadows and fog *

This is only one possible option for an extension. You may choose any extension you like. Soft shadows are used to simulate global illumination. A simple approximation can be achieved by sending several, slightly tilted rays instead of a single ray to the light source. This is an open ended task and you can implement as many path tracing features as you like. Be aware of the limitations of the used hardware. Processing many rays may quickly exhaust your GPU, which might lead to a crash of the rendering system and your browser.

Figure 2 shows an example for soft shadows and fog.

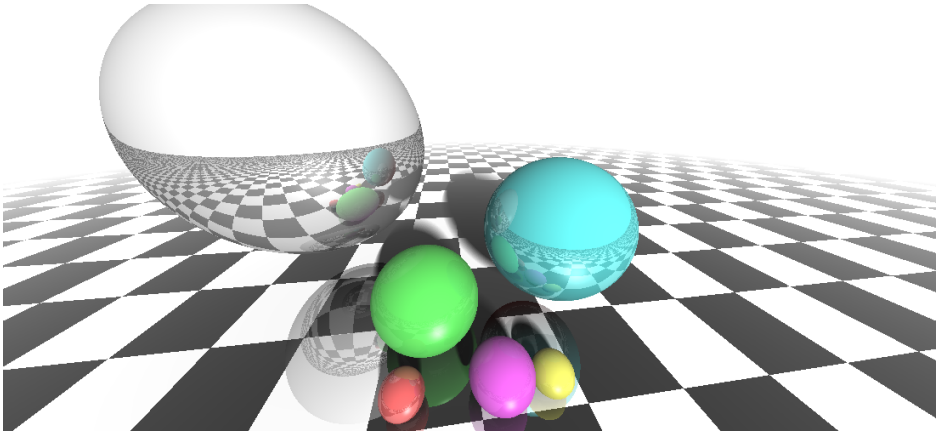

Figure 2: Example result for soft shadows and fog as implemented by hh4017 in 2019/2020

Exercise Files

You should each be access to a git repository on the department's GitLab server (https://gitlab.doc.ic.ac.uk/) which includes the skeleton files for each assessed component of this exercise, which are named:

- `task_5a.json`
- `task_5b.json`
- `task_5_extension_screenshot.png`

The main difference between the provided skeleton jsons and the default ShaderLabWeb json is the `lightPosition` and `lightInCamspace` uniforms, which you (as mentioned in the previous sections) should use in your answers.

Use git to clone the repository, then upload the skeletons to ShaderLabWeb using the "Upload State: .json", edit the shaders using the framework, then download them to overwrite the previous version in the git repository.

Autotesting on LabTS

Once you've committed and pushed your changes you can test them on LabTS. Log into https://teaching.doc.ic.ac.uk/labts and find the ex-

ercise for this coursework. On its page you can see all the commits you have pushed and buttons for requesting an autotest. Once the test has ran, you can check the result by viewing the "preview" pdf file.

For Graphics, the autotester will load your submissions onto ShaderLab-Web, resets the parameters of the framework, take a selection of screenshots with the camera and light in different positions and produces a report. The report shows you a record of how each screenshot was taken with a brief description of what it should show. It does NOT say whether your submission is correct or not. Instead, use the screenshots to try and spot any unusual or unnatural behaviour, then use the record of how the screenshot to replicate the case locally and investigate further.

Submission

On the same LabTS repository page, each commit has a "Submit this Commit" which you can click to submit that version of your code. You can submit again (as long as it is still in by the due date) and the newer submission will overwrite and supersede any earlier ones. We recommend making a submission as you complete each subtask.

HAVE A LOT OF FUN!!