Praktikum Echtzeit-Computergrafik
Assignment 9 – Ray Tracing (2)

Technische Universität München
Institut für Informatik
Lehrstuhl für Computergrafik & Visualisierung

Christoph Neuhauser, Simon Niedermayr – SS 24

In this assignment, we will extend the ray tracing code from the last assignment in ShaderLabWeb (https://vmwestermann10.in.tum.de/). We will add reflections, anti aliasing and environment mapping. Additionally, we get to know a new rendering effect called parallax mapping.

1. Expanding the Ray Tracer

Preparation: Scene Intersection

- At the end of the last assignment sheet, the ray tracer was testing against a number of spheres and a single plane. Based on this information the color of the pixel was determined. To make expanding and working with this code simpler, this procedure is best extracted in its own function. For that, we define a struct that holds all the information required to do lighting calculations after shooting a ray.

```
struct RayQuery{
  float t; // t value of intersection, t < 0 if no intersection was found
  vec3 normal; // normal vector at the hit point
  vec3 color; // color of the surface -- Consider making this a material ID
};
```

- The function that intersects a ray with the entire scene and returns information regarding the hit point, the normal vector at the hit point as well as the color / material should look something like this:

```
RayQuery sceneIntersect(vec3 ro, vec3 rd){
  float closestHit = 1./0.;
  // Query with t < 0 has not hit anything
  RayQuery query = RayQuery(-1.0,vec3(0),vec3(0));
  //Spheres
  for(int i = 0; i < spheres.length(); i++){
    float t = sphereIntersect(ro,rd,spheres[i]);
    if(t < 0.0 || t > closestHit) continue;
    closestHit = query.t = t;
    query.color = spheres[i].color;
    query.normal = normalize((ro+t*rd)-spheres[i].center);
  }
  //Plane
  vec3 planeNormal = vec3(0,1,0);
  vec3 planePoint = vec3(0,-0.5,0);
  for(int i = 0; i < 1 /*planes.length*/;i++){
    float t = planeIntersect(ro,rd,planeNormal,planePoint);
    if(t < 0.0 || t > closestHit) continue;
    closestHit = query.t = t;
    query.color = checkerboard(ro+t*rd,1.);
    // uncomment sign calculation to make the plane properly two sided.
    query.normal = planeNormal;//*(-sign(dot(rd,planeNormal)));
  }
  return query;
}
```

Note: In cases where material evaluation is more expensive or more diverse, you may want to save just a material ID and defer the actual evaluation to after the closest hit was found.

- Using this function the visual result should not change when we calculate our pixel color like this:

```
vec3 col = vec3(0);
RayQuery query = sceneIntersect(ro,rd);
if(query.t > 0.0){
  vec3 hitPoint = ro + query.t * rd;
  col = query.color * lightAtPoint(hitPoint,query.normal);
}
else{
  col = backgroundColor(rd);
}
```

Reflections

- Simulating perfect reflections in a ray tracer is as simple as continuing to follow the ray when this ray encounters a reflective surface. Since the ray could bounce forever between two mirrors, we need to restrict how often this can happen. The number of bounces should be chosen such that there is a balance between image quality and performance.

- To get the final color of a ray query, the ray accumulates some part as direct diffuse light and the other part as specular light. This goes on until a fully diffuse object or the skybox is hit (or the maximum number of bounces has been reached).

- To calculate the new ray direction, you can use the built-in $reflect()$ function. To avoid self intersection, the new ray origin should be the hit position with a tiny displacement along the normal.

```
vec3 RTXOn(vec3 ro, vec3 rd){
  const float specularPercent = /*some percentage*/;
  const float diffusePercent = 1.-specularPercent;
  const int MAX_BOUNCES = 4;

  float contribution = 1.0;
  vec3 col = vec3(0);
  for(int i = 0; i < MAX_BOUNCES; i++){
    RayQuery query = sceneIntersect(ro,rd);
    if(query.t < 0.0){
      col += contribution*backgroundColor(rd);
      break;
```

```
    }
    vec3 hitPoint = ro+query.t*rd;
    vec3 light = lightAtPoint(hitPoint,query.normal);

    col += contribution * diffusePercent * (query.color * light);

    ro = hitPoint + 1e-3*query.normal;
    rd = reflect(rd,query.normal);

    contribution *= specularPercent;
  }
  return col;
}
```

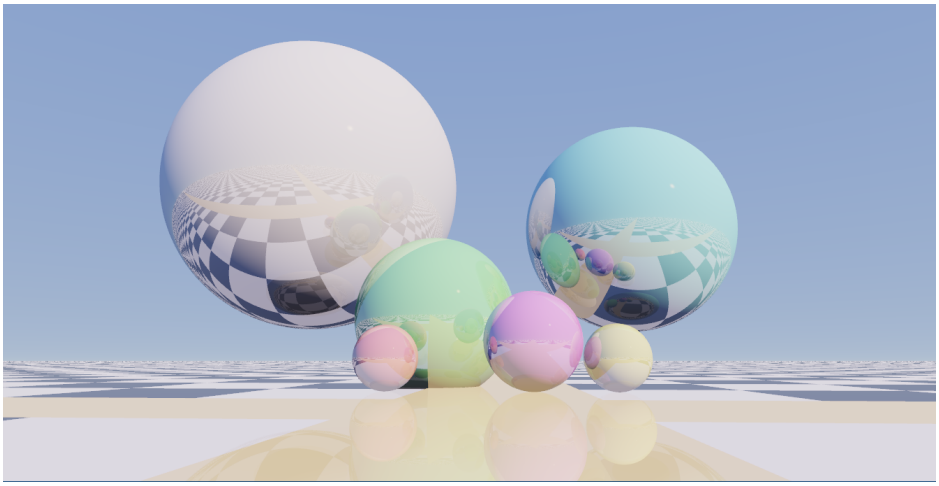

Figure 1: Reflections with a 0.368 percent reflectivity

Super-Sampling Anti-Aliasing (SSAA)

- If you take a close look at the boundaries of the ray traced geometry, jagged edges along the pixel borders can be seen. We see these aliasing artifacts because for each pixel we sent out only a single ray. A simple way to remedy this is to shoot more rays per pixel in slightly different directions and average the results. This effectively increases the resolution of your image which results in a clearer image.

- To make effective use the additional samples, it is important to choose good sub pixel offsets. For this exercise, we use the same pattern that is

used for 4xMSAA (Multi-Sample Anti-Aliasing). The standard sample locations for different sample counts can be found in the specification of graphics APIs such as Vulkan: https://registry.khronos.org/vulkan/specs/1.3/html/vkspec.html#primsrast-multisampling.

```
vec3 col = vec3(0);
vec3 ro = camPos;
const vec2 subPixelOffsets[] = vec2[](
  vec2(0.375,0.125)-vec2(0.5),
  vec2(0.875,0.375)-vec2(0.5),
  vec2(0.125,0.625)-vec2(0.5),
  vec2(0.625,0.875)-vec2(0.5)
);
vec2 pixelSize = vec2(1.)/vec2(imageWidth,imageHeight);
for(int i = 0; i < subPixelOffsets.length();++i){
  vec2 uv = fragmentTextureCoordinates + subPixelOffsets[i] * pixelSize;
  vec2 pixelNdc = 2. * uv - 1.;
  vec3 pixelViewSpace = (ndcToView * vec4(pixelNdc,1.,1.)).xyz;
  vec3 rd = normalize(mat3(viewToWorld) * pixelViewSpace);

  col += /*color for this ray*/
}
col *= 1./float(subPixelOffsets.length());
```

Note: The difference between SSAA and MSAA is that MSAA only increases the samples around the edges of polygons.

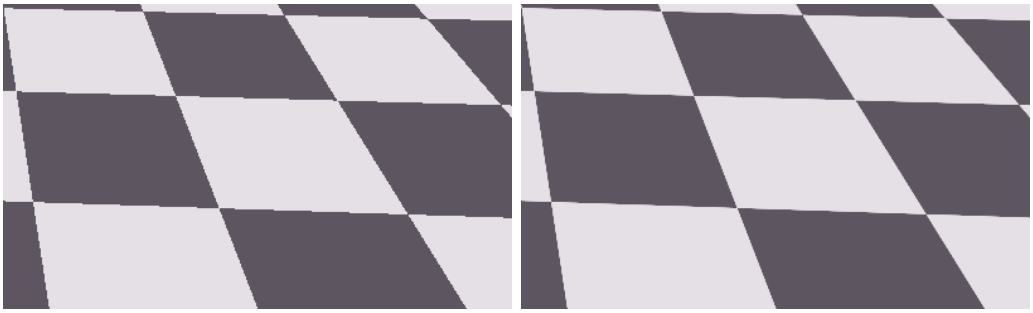

Figure 2: Left: No anti aliasing; Right: 16x SSAA

Environment Mapping

- Previously, we used a procedural color for the skybox. Now, we will now use a type of environment map called a *cube map*.

Cube maps store the texture faces of an object we can imagine as an infinitely far away cube surrounding our scene. We can use the ray direction to look up the corresponding value in the cube map.

Figure 3: The six sides of the cube map used for this assignment.

- Your task: Replace the procedural sky color with a texture lookup from the environment map. Add a `uniform samplerCube` texture object to the shader and upload the textures provided on Moodle in the `Uniforms` tab.
- Now, read the color of the environment map using `vec3 envMapColor = texture(environmentMap, rayDirection).rgb;` and convert the colors from sRGB to linear space such that the light calculations work correctly. You can use the accurate or the approximate conversion (`linear = pow(sRGB,vec3(2.2))`) for this.

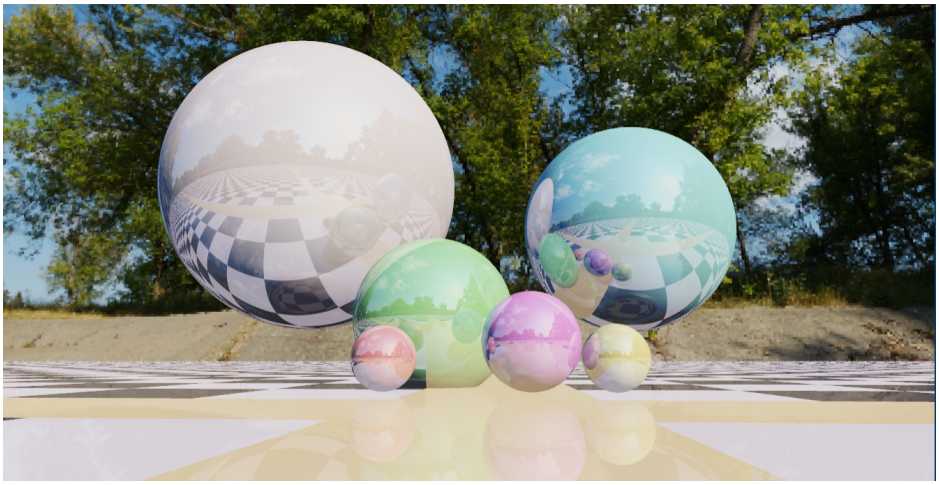

Figure 4: The scene from last assignment now using an environment map.

2. Parallax Mapping

For this task, you can use the solution of Assignment 6 (task 3) as a template, as we will continue to work with the textures mapped onto a plane geometry. In Assignment 6, there was one texture we ignored, namely the height texture. In this task, you will implement an effect called parallax mapping.

On this page, an in-depth view into parallax mapping is given: https://learnopengl.com/Advanced-Lighting/Parallax-Mapping

We recommend reading this article and follow it step by step. The best results can be achieved using parallax occlusion mapping, which is introduced at the end of the article. You do not need to bother with the tangent space, as we have a simple plane geometry. You can use the simplified code below, which computes `vec3 viewDir` from the article as the vector `vec3(-v.x, v.z, v.y)`. This works, as our plane has constant tangent and bitangent directions.

```
vec2 texCoords = parallaxMapping(fragmentTextureCoordinates, vec3(-v.x, v.z, v.y));
```

To achieve the images below, a maximum number of layers of 256 and a height scale of 0.1 was chosen. The brick textures are provided on Moodle.

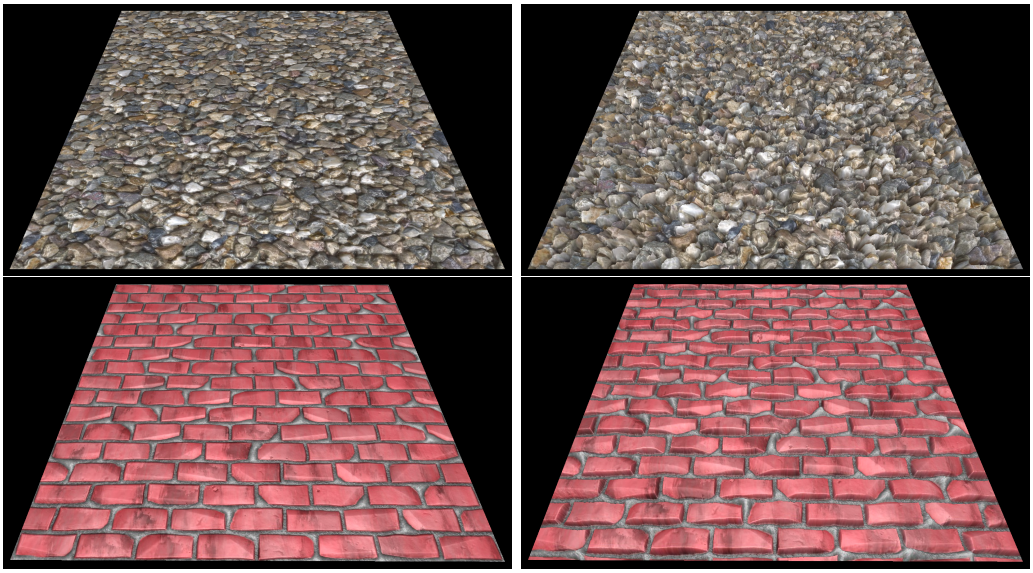

Figure 5: Left: Solution to task 3 from Assignment 6 with normal mapping. Right: Solution to this task using parallax occlusion mapping.