Praktikum Echtzeit-Computergrafik
Assignment 8 – Ray Tracing (1)

Technische Universität München
Institut für Informatik
Lehrstuhl für Computergrafik & Visualisierung

Christoph Neuhauser, Simon Niedermayr – SS 24

In this assignment, we will do some simple form of ray tracing in ShaderLabWeb (https://vmwestermann10.in.tum.de/). In the next two assignments, we will implement a simple Whitted-style ray tracer.

Introduction

Rasterization, which was used for generating the images in the previous assignments, works by projecting our geometry onto a camera plane. Ray tracing, on the other hand, works by sending out view rays for every pixel into the scene and computing the intersections of the ray with the scene geometry. Nowadays, modern GPUs have hardware-accelerated functionality for ray-triangle and ray-box intersection tests. In order to speed up the number of triangles or boxes that need to be checked for intersection, an acceleration structure is built internally. For the sake of simplicity, we will do without special acceleration structures, like bounding volume hierarchies, in this course, and just check the intersection of the view rays with some simple geometric objects like planes and spheres analytically.

Preparing the ShaderLabWeb

- We provide a template for this task, start by opening it.

Ray Origin and Ray Direction

A ray in 3D space is defined as the set of points $\{r(t) = p + t \cdot d \mid t \in \mathbb{R}_0^+\}$, where $p \in \mathbb{R}^3$ is the ray origin and $d \in \mathbb{R}^3$ is the ray direction.

In order to be able to do ray tracing, we will first need to know the origin of our primary ray (i.e., the camera position) and the direction of the primary ray (i.e., into which direction a certain pixel looks).

From Screen Space to World Space

- Our ray origin is simply given as the position of the camera. The position can be obtained either as a separate uniform variable or is contained in the fourth column of the inverse view matrix.

  ```
  vec3 ro = camPos;
  ```

- We want to send the rays through the pixels of our image plane. For each pixel we are given texture coordinates in the normalized range $[0, 1]$. To calculate the direction in world space this pixel corresponds to, the first step is to move back to normalized device coordinates.

  The normalized device coordinate space (NDC space) was explained in Assignment 4. First, we compute the x and y normalized device coordinates belonging to the pixel.

  ```
  vec2 pixelNdc = 2. * fragmentTextureCoordinates - 1.;
  ```

- Additionally, we need to add the missing z and w component to get a complete vector in the homogenous coordinates format. By adding $z = 1$ and $w = 1$, we get a point that corresponds to a position in the far plane. The view space position of this point can be obtained by multiplying the NDC coordinates with the inverse of the projection matrix.

  ```
  vec3 pixelViewSpace = (ndcToView * vec4(pixelNdc,1.,1.)).xyz;
  ```

Note: You can get the inverse projection matrix directly as a uniform variable or by inverting the regular projection matrix.

- To go back from view space to world space, the inverse of the view matrix can be used. This is essentially the model matrix of the camera. Since the camera is the at the origin $(0, 0, 0)$ in the view space, the vector for the view space coordinates is equivalent to our ray direction. For the ray direction, we thus don't need any translation. However, since not all pixels are equally far away from the origin, we still need to normalize the direction.

```
  vec3 rd = normalize(mat3(viewToWorld) * pixelViewSpace);
```

Skybox

- Every ray that does not hit any geometry will need some kind of fallback value. The skybox is "infinitely" far away, so only the ray direction is needed.

  For a nicer appearance and to have some sense of direction, we will calculate a procedural skybox. Values for the color of the sky, the sun and the region below the horizon are provided, but feel free to change and play around with them.

```
uniform vec3 sunColor; // e.g. (1.64, 1.27, 0.99)
uniform vec3 sunDirection; // e.g. (1,0.75,1)
uniform vec3 skyColorLow; // e.g. (0.36, 0.45, 0.57)
uniform vec3 skyColorHigh; // e.g. (0.14, 0.21, 0.49)
uniform vec3 voidColor; // e.g. (0.025, 0.05, 0.075)

vec3 backgroundColor(vec3 rd){ // Fine tuned by pure arbitrariness
  if(rd.y < 0.0)
    return voidColor;
  const float skyGradient = 1./4.;
  vec3 skyColor = mix(skyColorLow, skyColorHigh, pow(rd.y, skyGradient));
  const float sunGradient = 2400.;
  float sunAmount = pow(max(dot(rd,normalize(sunDirection)), 0.),sunGradient);
  return mix(skyColor, sunColor, sunAmount);
}
```

Note: This function creates gradients using exponential functions (one based on how far you look up and one based upon how directly you look into the sun). This is rather artistic and not based on physics.

- As discussed in a previous assignment, lighting calculations are done in linear space, but our monitors typically present sRGB images. Further, our lighting calculations are in HDR, meaning we are not limited to the range $[0, 1]$. To avoid clipping color channels and to produce more accurate colors, a tone mapping function needs to be applied to convert from HDR to LDR.

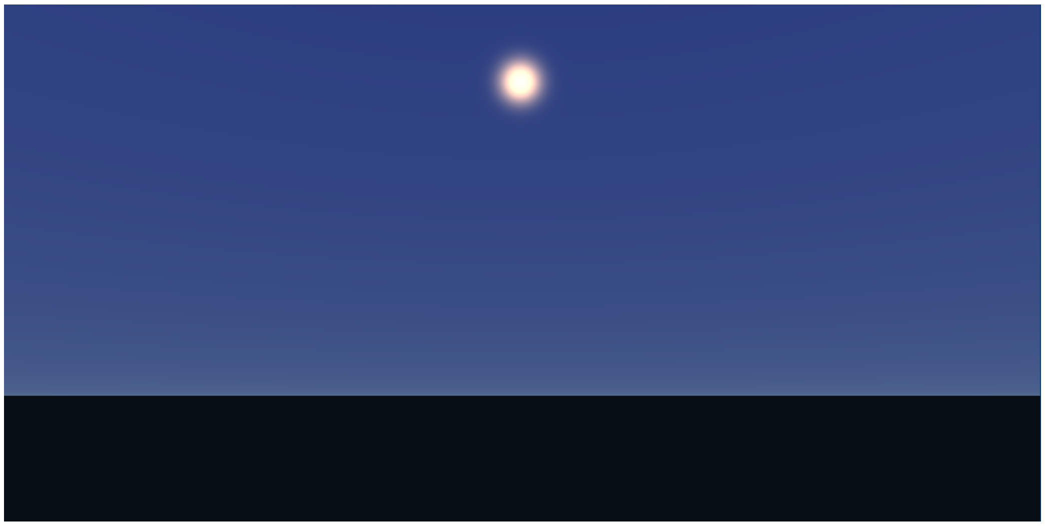
Figure 1: Skybox but too dark since tone mapping and color space conversion were not applied

We use an approximation of the ACES filmic tone mapping curve. It is not important to remember this exact formula.

```
vec3 tonemapAndGammaCorrect(const vec3 x){
  return pow(clamp((x*(2.51*x+0.03)/(x*(2.43*x+0.59)+0.14)),0.0,1.0),vec3(1.0/2.2));
}

void main() {
/*... code calculating ray origin (ro) and ray direction (rd) ...*/
  vec3 col = backgroundColor(rd);
  fragColor = vec4(tonemapAndGammaCorrect(col), 1.);
}
```

Note: It is often desirable to have more control over the virtual camera (e.g. exposure, white point, etc.). Also try to vary the input to the tone mapper or try some different functions.

- If everything was done correctly so far, you should be able to look around and find the sun.

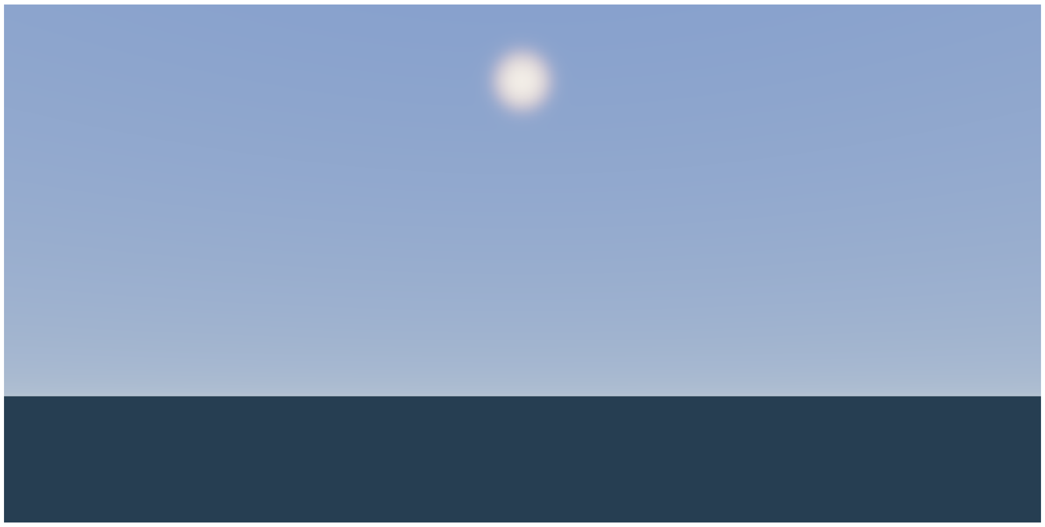

Figure 2: Praise the sun! A Skybox from exponential gradients based on height of view direction; tone mapping and color space conversion is applied.

Ray-Sphere Intersection

Sphere definition

- Spheres are simple geometric objects defined by a center point and a radius. For the sake of rendering a material determining the look of the sphere is also desirable. In our case, a simple color suffices. For more complex materials a material ID would be stored instead.

  We aggregate this information into a struct.

  ```
  struct Sphere{
    vec3 center;
    float radius;
    vec3 color;
  };
  ```

- To keep things simple, we first render a single sphere centered at the origin with a radius of 1. You can choose any color for this sphere, just be sure to choose one that stands out against the background.

  We will render a couple of spheres later on, so the storage will be provided by an array. You can define this array to be a uniform variable or directly embed it into the code as a constant global variable. Below is an example on how to do the latter.

  ```
  const Sphere spheres[] = Sphere[](
    Sphere(vec3(0.,0.,0.),1.0,vec3(1.0,0.1,0.1))
  );
  ```

Intersection test

- To test whether any of our view rays "see" / intersect the sphere, we use the function below. This function takes a ray origin, a ray direction and a sphere as defined above and returns the t value for which the ray intersects the sphere. Since we don't want to render anything that is behind the camera, a negative return value will be used when we don't "see the sphere". You can find the mathematical derivation which this function is based on in the appendix.

```
float sphereIntersect(vec3 ro, vec3 rd, Sphere s){
  ro -= s.center;
  float b = dot(ro,rd);
  float c = dot(ro,ro) - s.radius*s.radius;
  float d = b*b - c;

  if(d < 0.0) return -1.; // No intersection

  d = sqrt(d);
  float t1 = -b - d;
  float t2 = -b + d;

  if(t1 < 0.0) return t2; // Potentially inside

  return t1;
}
```

Note: More intersection functions can be found here → ***https://iquilezles.org/articles/intersectors/***

- Testing and using our intersection function can quickly be done like this inside the main function after calculating ray origin and ray direction:

```
  // inside main
  vec3 col = vec3(0);
  float t = sphereIntersect(ro,rd,spheres[0]);
  if(t > 0.0)
  col = spheres[0].color;
  else
  col = backgroundColor(rd);
```

If everything went well we should be able to move the camera around and find the sphere.

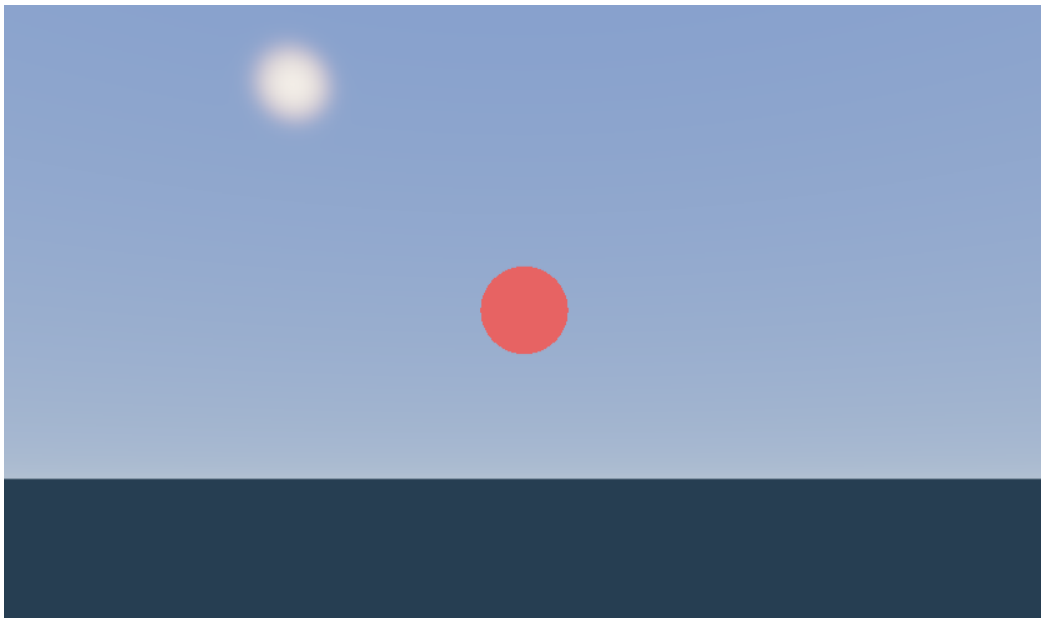
Figure 3: Successful ray sphere intersection

Light and Shadow

Light

- Since creating a physically based lighting simulation is out of scope for this assignment, we will use a simplified lighting model instead. Our function takes a point in world space as well as a surface normal and returns the light received at this point.

  The first part of this model is simple Lambertian diffuse shading for the direct illumination of the sun.

  ```
  vec3 lightAtPoint(vec3 point, vec3 N){
    vec3 L = normalize(sunDirection);
    float NdL = max(dot(N,L),0.0);
  ```

- The second part is the light coming from the sky. You can think of this as being part of the "ambient light" in the Phong illumination model. The contribution of the sky is approximated by how far the normal points upwards (i.e., towards the sky). This is done by remapping the y component of the normal vector from $[-1, 1]$ to $[0, 1]$.

```
  float NdSky = clamp(0.5*N.y+0.5, 0.0, 1.0 );
  return sunColor * NdL + skyColorHigh * NdSky;
}
```

- We can use this function to calculate the shading factor for our sphere. The intersection function returns us the value t for the closest hit point. The normalized vector from the sphere center to this hit point gives us the normal of the sphere at this point. With this information we can calculate how much light our sphere receives at this point and attenuate (i.e., multiply) it with the color of the sphere itself.

```
vec3 col = vec3(0);
float t = sphereIntersect(ro,rd,spheres[0]);
if(t > 0.0){
  vec3 hitPoint = ro + t*rd;
  vec3 normal = normalize(hitPoint-spheres[0].center);
  vec3 light = lightAtPoint(hitPoint,normal);
  col = light*spheres[0].color;
}
else //...
```

  With this lighting setup, our sphere should now look much nicer.

Shadow

- Our sphere currently doesn't have anything to cast shadows on, so we should extend our scene first. Add as many spheres as you want. Below is a suggestion for the scene setup.

```
Sphere spheres[] = Sphere[](
  Sphere(vec3(-2.0,1.5, -3.5),1.5,vec3(0.8,0.8,0.8)),
  Sphere(vec3(-0.5,0.0, -2.0),0.6,vec3(0.3,0.8,0.3)),
  Sphere(vec3( 1.0,0.7, -2.2),0.8,vec3(0.3,0.8,0.8)),
  Sphere(vec3( 0.7, -0.3, -1.2),0.2,vec3(0.8,0.8,0.3)),
  Sphere(vec3(-0.7, -0.3, -1.2),0.2,vec3(0.8,0.3,0.3)),
  Sphere(vec3( 0.2, -0.2, -1.2),0.3,vec3(0.8,0.3,0.8))
);
```

- Now is also the time to write a function that allows us to test an arbitrary amount of spheres. We will make use of the `out` keyword to write a function that effectively returns two things: The value t of the closest intersection as well as an ID for the sphere we hit.

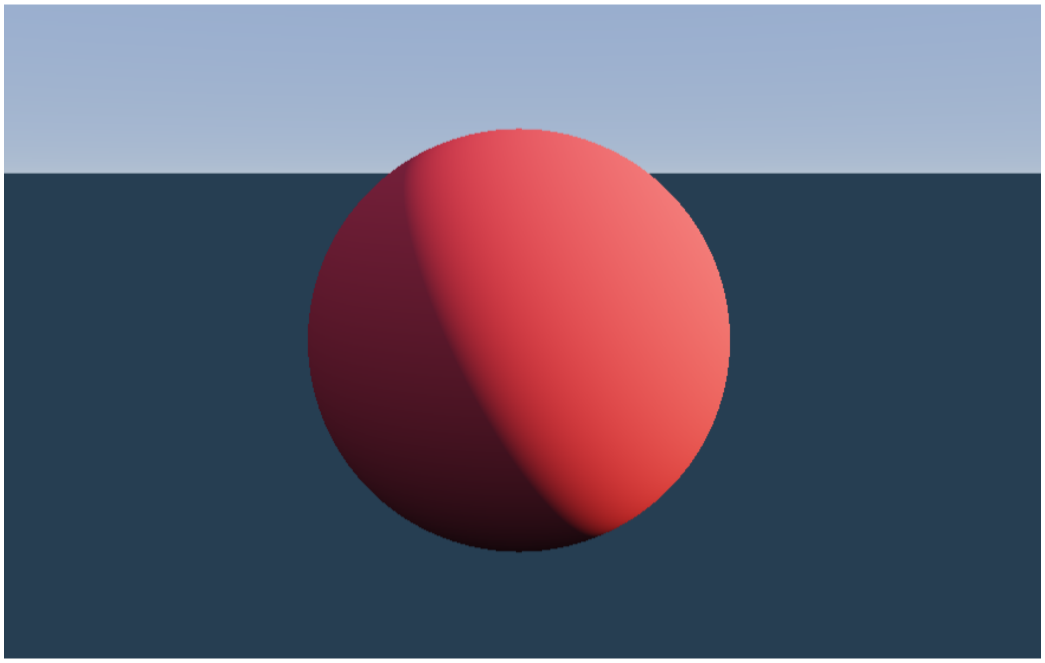

Figure 4: Sphere with direct illumination and skylight

```
float closestSphereIntersect(vec3 ro, vec3 rd, out int sphereIndex){
  /* Infinity in IEEE 754 floats. If this doesn't work use a very large number (e.g.
      1e42) instead.*/
  float closestHit = 1./0.;
  /* Is negative in case no sphere is hit. Could also use a check in the end whether
      closest hit is too far away to determine whether anything was hit at all.*/
  float bestT = -1.0;
  for(int i = 0; i < spheres.length(); i++){
    float t = sphereIntersect(ro,rd,spheres[i]);
    if(t < 0.0 || t > closestHit) continue;
    closestHit = bestT = t;
    sphereIndex = i;
  }
  return bestT;
}
```

Using this function is a simple matter of adjusting the main function.

```
vec3 col = vec3(0);
int sphereIndex;
float t = closestSphereIntersect(ro,rd,sphereIndex);
if(t > 0.0){
  vec3 hitPoint = ro + t*rd;
  vec3 normal = normalize(hitPoint-spheres[sphereIndex].center);
  vec3 light = lightAtPoint(hitPoint,normal);
  col = light*spheres[sphereIndex].color;
}
else //...
```

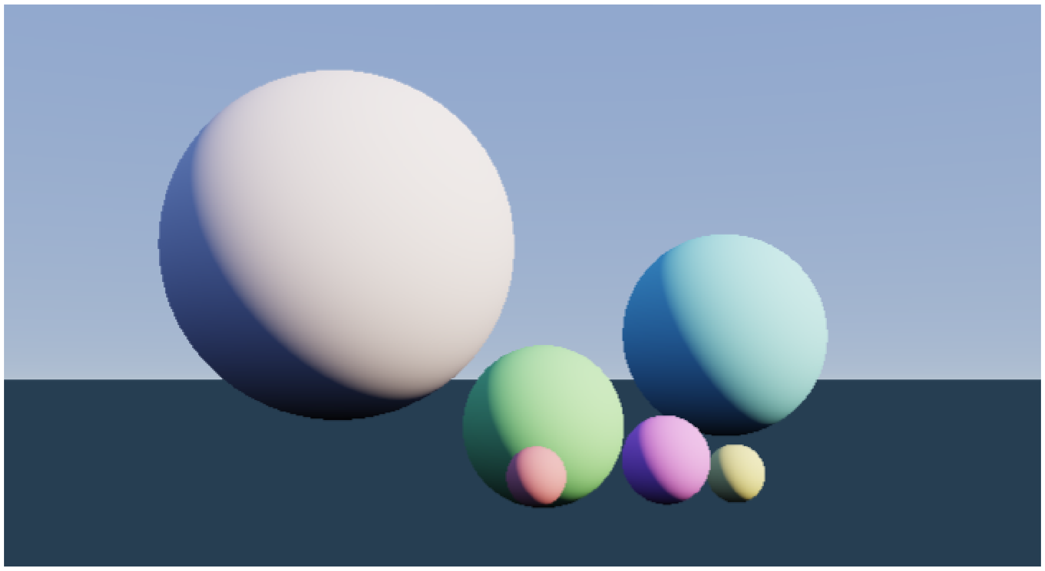
Figure 5: Group of Spheres – No shadow

- A point is in shadow when the path to the light is blocked by another object.

  We make use of our function which tests against all objects in the scene for this.

```
float shadowRay(vec3 ro, vec3 rd){
  int ignore;
  float tSphere = closestSphereIntersect(ro, rd, ignore);
  return 1.-step(0.0,tSphere); // 1 if tSphere < 0 else 0
}
```

- The "shadowRay" function returns a percentage value for how much light gets carried along this ray. We can use this to attenuate the direct illumination in our "lightAtPoint" function.

```
vec3 lightAtPoint(vec3 point, vec3 N){
  vec3 L = normalize(sunDirection);
  float NdL = max(dot(N,L),0.0);
  float NdSky = clamp(0.5*N.y+0.5, 0.0, 1.0 );
  float shadow = shadowRay(point + 1e-3*N, L);
  return sunColor * NdL * shadow + skyColorHigh * NdSky;
}
```

Note: To avoid self shadowing the point is offset by a small amount in the direction of the normal

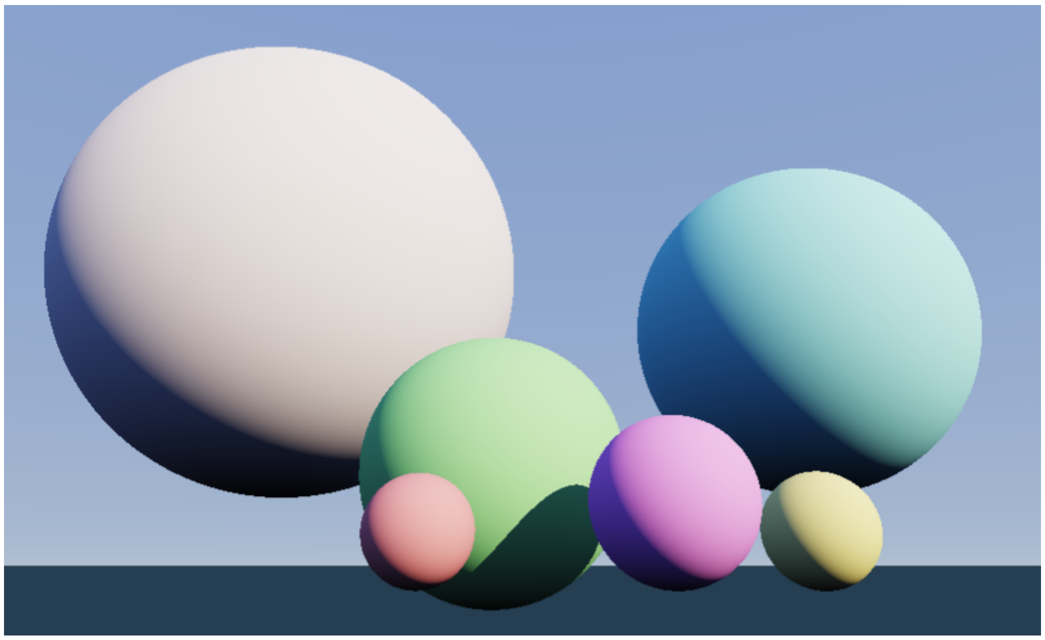
Figure 6: Spheres now cast shadows on each other

Ground Plane

Ray-Plane Intersection

- Right now the spheres are just floating in the void. To give more context to our scene, we render a ground plane. To find the intersection of a ray with a plane defined by a normal vector n and a point on the plane p, the function below can be used. As with the sphere intersection, you can find the mathematical derivation of this function in the appendix.

```
float planeIntersect(vec3 ro, vec3 rd, vec3 n, vec3 p){
  return dot(p-ro,n)/dot(rd,n);
}
```

Note: Division by zero – which happens when the view ray is parallel to the plane – is a well-defined operation on floating point numbers. The result is Infinity

Procedural Pattern

- Since a single color for our plane is rather monotonous and doesn't help much for orientation, we generate a procedural checkerboard pattern based on the current position in the world. The function takes in a 3D location and a tile size and discretized the plane into black and white squares. Additionally, a line on the x- and z- axis is drawn.

```
// checkerboard pattern based on position of point with a center line
vec3 checkerboard(vec3 point, float tileSize){
  vec2 p = point.xz / tileSize;
  vec2 t = floor(p+0.5);
  if(t.x == 0.0 || t.y == 0.0) // middle line for better orientation
  return vec3(1.0,0.8,0.4);
  return mix(vec3(1.),vec3(.1),mod(t.x+t.y,2.));
}
```

- You should be able to integrate the plane intersection into your ray tracer. Using a plane point of $(0, -0.5, 0)$ and a normal vector $(0, 1, 0)$ you should get to the result below.

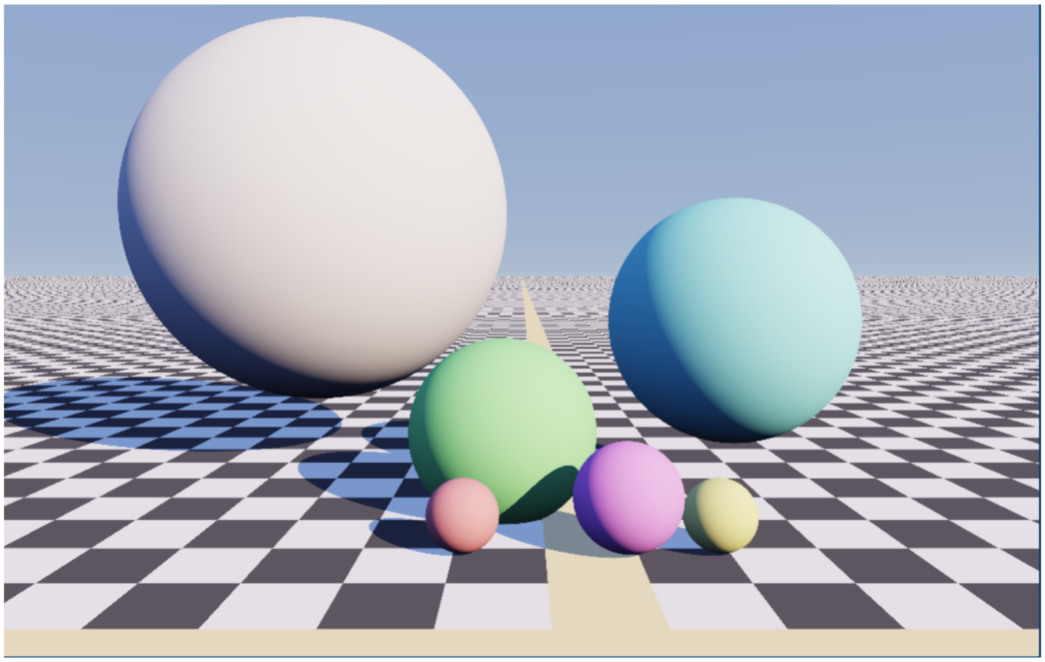

Figure 7: Spheres on Checkerboard

Appendix

Deriving the equation for ray-sphere intersection

With a sphere centered at the origin, all points with a distance of *radius* away from the origin are on the surface of the sphere. Hence, we need to check whether our ray has a point that is *radius* units away from the origin. If the sphere is not centered around the origin, simply subtract the sphere center from the ray origin, to shift the coordinate system.

$$\begin{aligned} ||ro + t * rd|| &= radius \\ \sqrt{(ro.x + t * rd.x)^2 + (ro.y + t * rd.y)^2 + (ro.z + t * rd.z)^2} &= radius \\ (ro.x + t * rd.x)^2 + (ro.y + t * rd.y)^2 + (ro.z + t * rd.z)^2 &= radius^2 \\ (ro.x + t * rd.x)^2 + (ro.y + t * rd.y)^2 + (ro.z + t * rd.z)^2 - radius^2 &= 0 \end{aligned} \tag{1}$$

Re-arranging the formula a bit gives:

$$\begin{aligned}
&= (ro.x + t * rd.x)^2 + (ro.y + t * rd.y)^2 + (ro.z + t * rd.z)^2 - radius^2 \\
&= ro.x^2 + 2 * (ro.x * t * rd.x) + (t * rd.x)^2 \\
&+ro.y^2 + 2 * (ro.y * t * rd.y) + (t * rd.y)^2 \\
&+ro.z^2 + 2 * (ro.z * t * rd.z) + (t * rd.x)^2 - radius^2 \\
&= ro.x^2 + ro.y^2 + ro.z^2 \\
&+2 * t * (ro.x * rd.x + ro.y * rd.y + ro.z * rd.z) \\
&+t^2 * (rd.x^2 + rd.y^2 + rd.z^2) - radius^2 \\
&= dot(ro, ro) + 2t * dot(ro, rd) + t^2 * dot(rd, rd) - radius^2 = 0
\end{aligned} \tag{2}$$

It is apparent that this is a quadratic equation we need to solve for t.

$$\begin{aligned}
t_1, t_2 &= (-b \pm \sqrt{b^2 - 4ac})/(2a) \\
a &= dot(rd, rd) = 1 \\
b &= 2 * dot(ro, rd) \\
c &= dot(ro, ro) - radius^2
\end{aligned} \tag{3}$$

We know $a = 1$ because rd is normalized. Plugging in the values into the quadratic equation, we get:

$$\begin{aligned}
&t_1, t_2 = \\
&= (-2 * dot(ro, rd) \pm \sqrt{(2 * dot(ro, rd))^2 - 4 * 1 * (dot(ro, ro) - radius^2)})/2 \\
&= (-2 * dot(ro, rd) \pm \sqrt{4 * (dot(ro, rd))^2 - 4 * (dot(ro, ro) - radius^2)})/2 \\
&= (-2 * dot(ro, rd) \pm 2 * \sqrt{(dot(ro, rd))^2 - (dot(ro, ro) - radius^2)})/2 \\
&= -dot(ro, rd) \pm \sqrt{(dot(ro, rd))^2 - (dot(ro, ro) - radius^2)}
\end{aligned} \tag{4}$$

The equation has up to two values for the intersection. There is only one unique value if the ray is tangent to the sphere, and zero real solutions if the ray misses the sphere entirely. A negative value for a solution indicates that the intersection point is behind the ray origin.

Deriving the equation for ray-plane intersection

A plane is uniquely identified by a normal vector n and any point p lying on that plane. For every other point q the direction vector defined by $q - p$ is perpendicular to the normal vector. For a ray plane intersection, we need to check whether any point along the ray also lies on the plane. We thus have to solve the following equation for t.

$$dot((ro + t * rd) - p, n) = 0 \tag{5}$$

Rearranging to isolate t

$$\begin{aligned} dot((ro + t * rd) - p, n) = 0 \\ = n.x * (ro.x + t * rd.x - p.x) \\ +n.y * (ro.y + t * rd.y - p.y) \\ +n.z * (ro.z + t * rd.z - p.z) \\ \\ = n.x * (ro.x - p.x) + t * rd.x * n.x \\ +n.y * (ro.y - p.y) + t * rd.y * n.y \\ +n.z * (ro.z - p.z) + t * rd.z * n.z \\ \\ = t * (rd.x * n.x + rd.y * n.y + rd.z * n.z) \\ +n.x * (ro.x - p.x) \\ +n.y * (ro.y - p.y) \\ +n.z * (ro.z - p.z) \\ \\ = t * dot(rd, n) + dot(ro - p, n) \end{aligned} \tag{6}$$

Now solving for t

$$\begin{aligned} t * dot(rd, n) + dot(ro - p, n) = 0 \\ t * dot(rd, n) = -dot(ro - p, n) \\ t = -dot(ro - p, n)/dot(rd, n) \\ t = dot(p - ro, n)/dot(rd, n) \end{aligned} \tag{7}$$