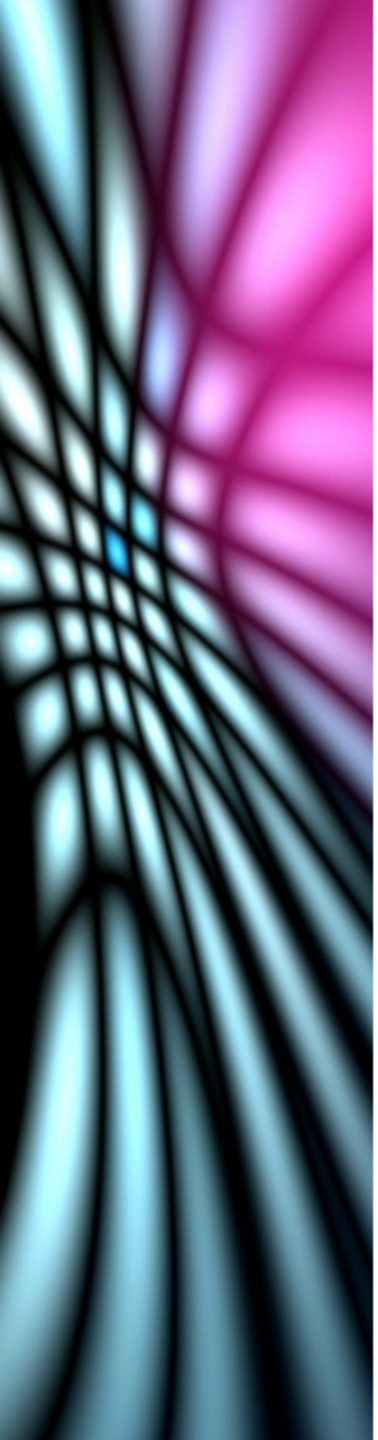
Rendering Competition 2022

Xiana Carrera Alonso



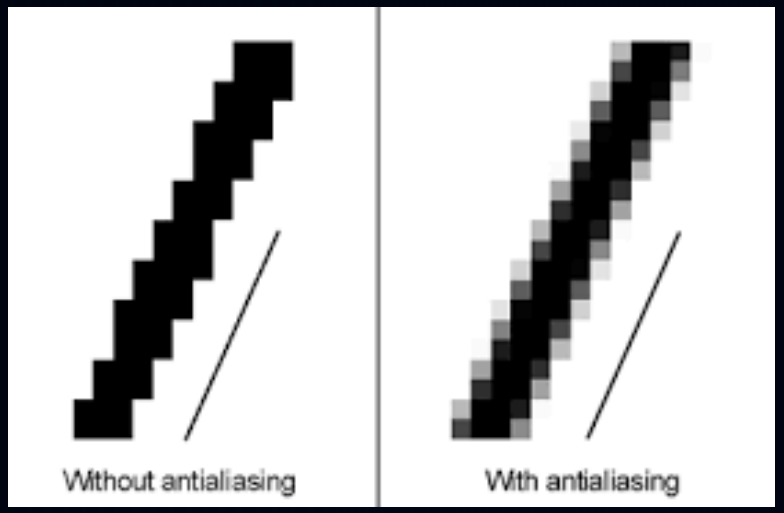
Goals

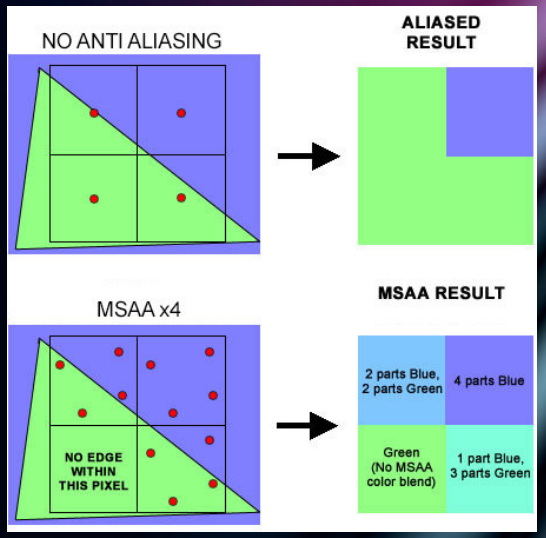
- Antialiasing
- Depth of Field
- Triangle meshes
  - **Ray - triangle intersection (pyramid)**
  - **Mesh reader (.obj)**
  - **Smooth normal interpolation**
- Acceleration structure: kd-tree
- Advanced Reflectance Model (Ward)
- Soft shadows
- Perlin noise

Antialiasing

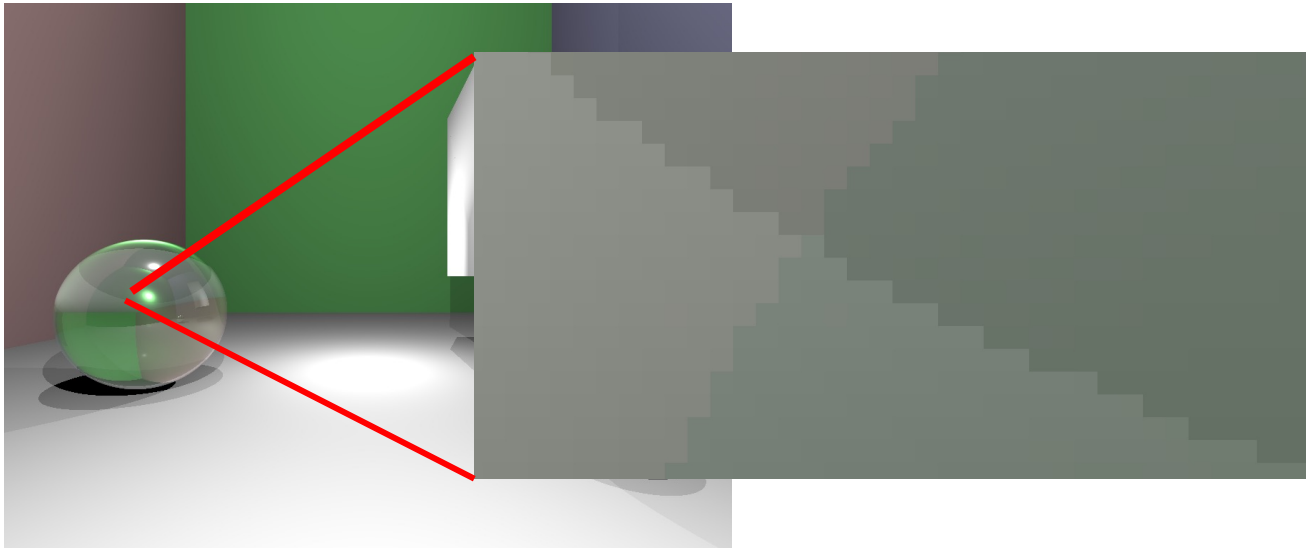
- Smooth sharp edges
- Shoot more than 1 ray per pixel and average the color sum

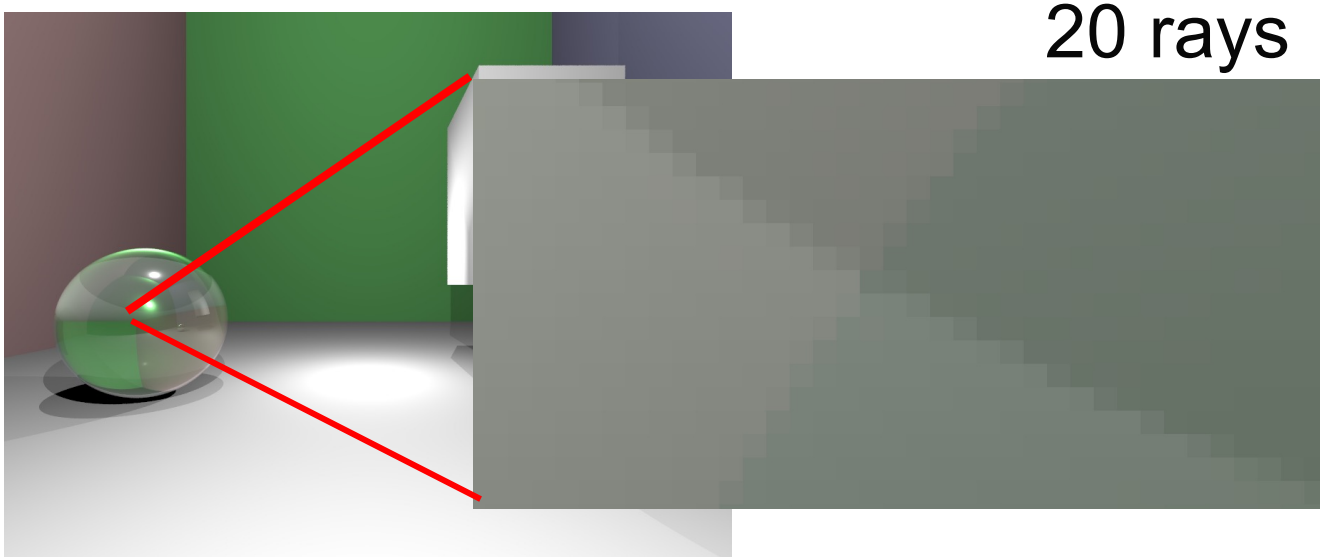
- Stochastic supersampling
  - **Shoot random rays to one pixel**





Antialiasing results

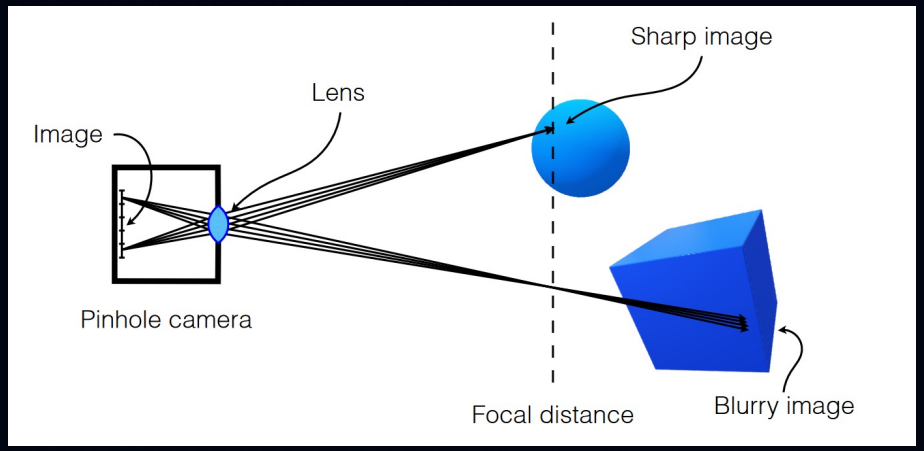


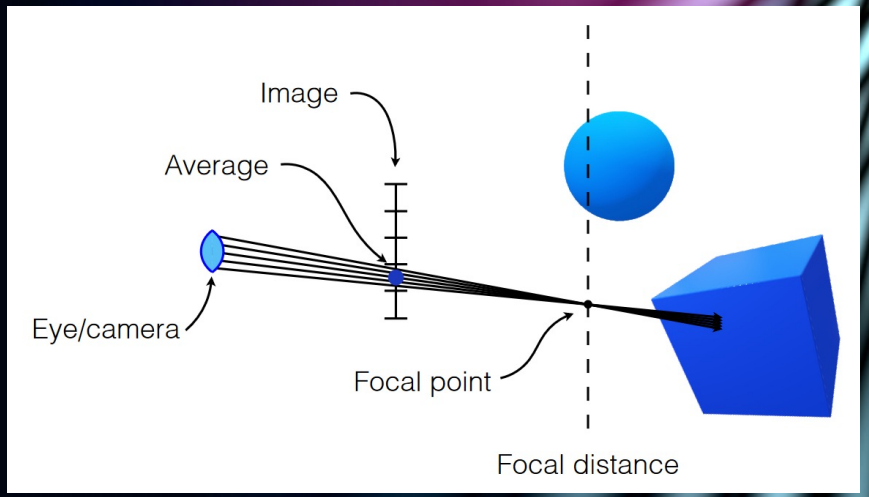


20 rays

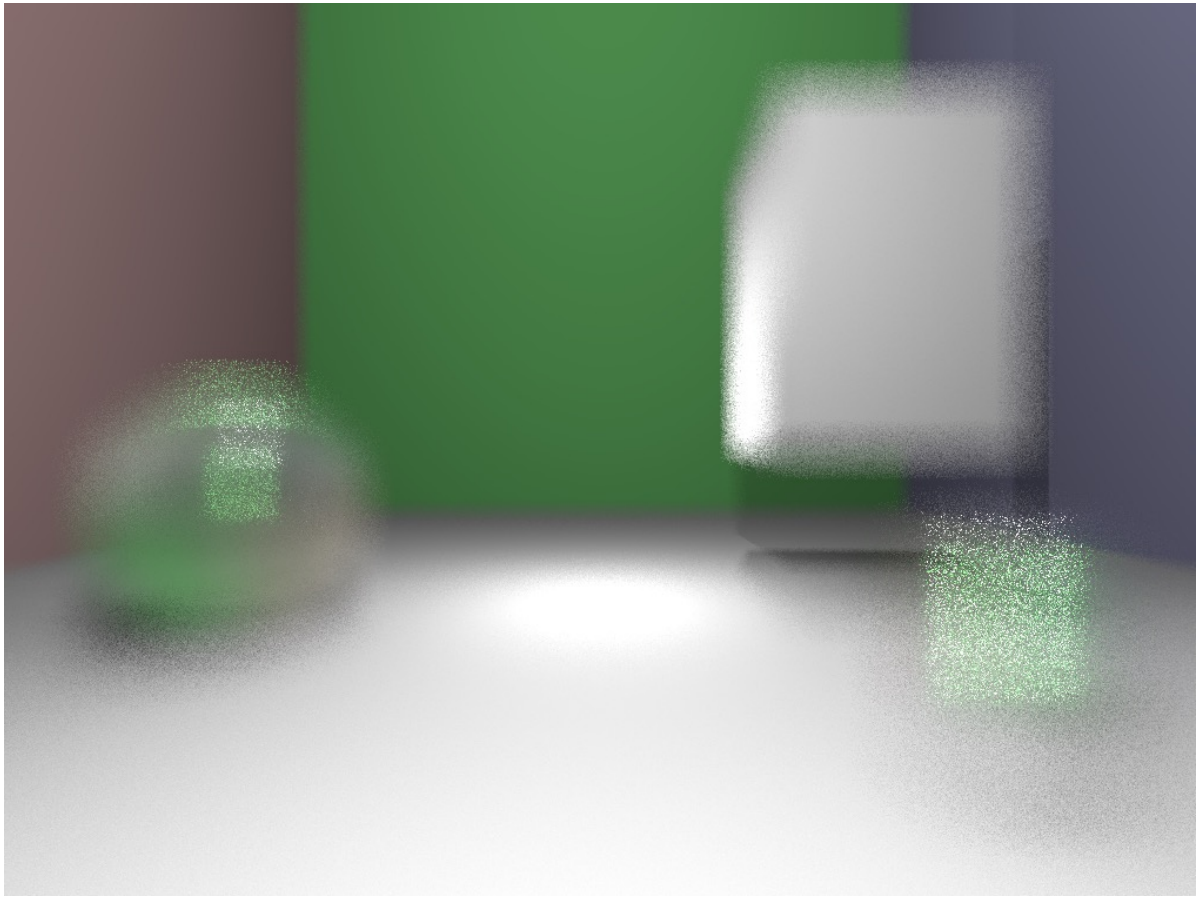
Depth of Field

- Objects at a certain distance appear focused and sharp.
- Farther away, they become blurry.
- Shoot several rays, each starting from one camera position within a certain aperture, and average the color sum.



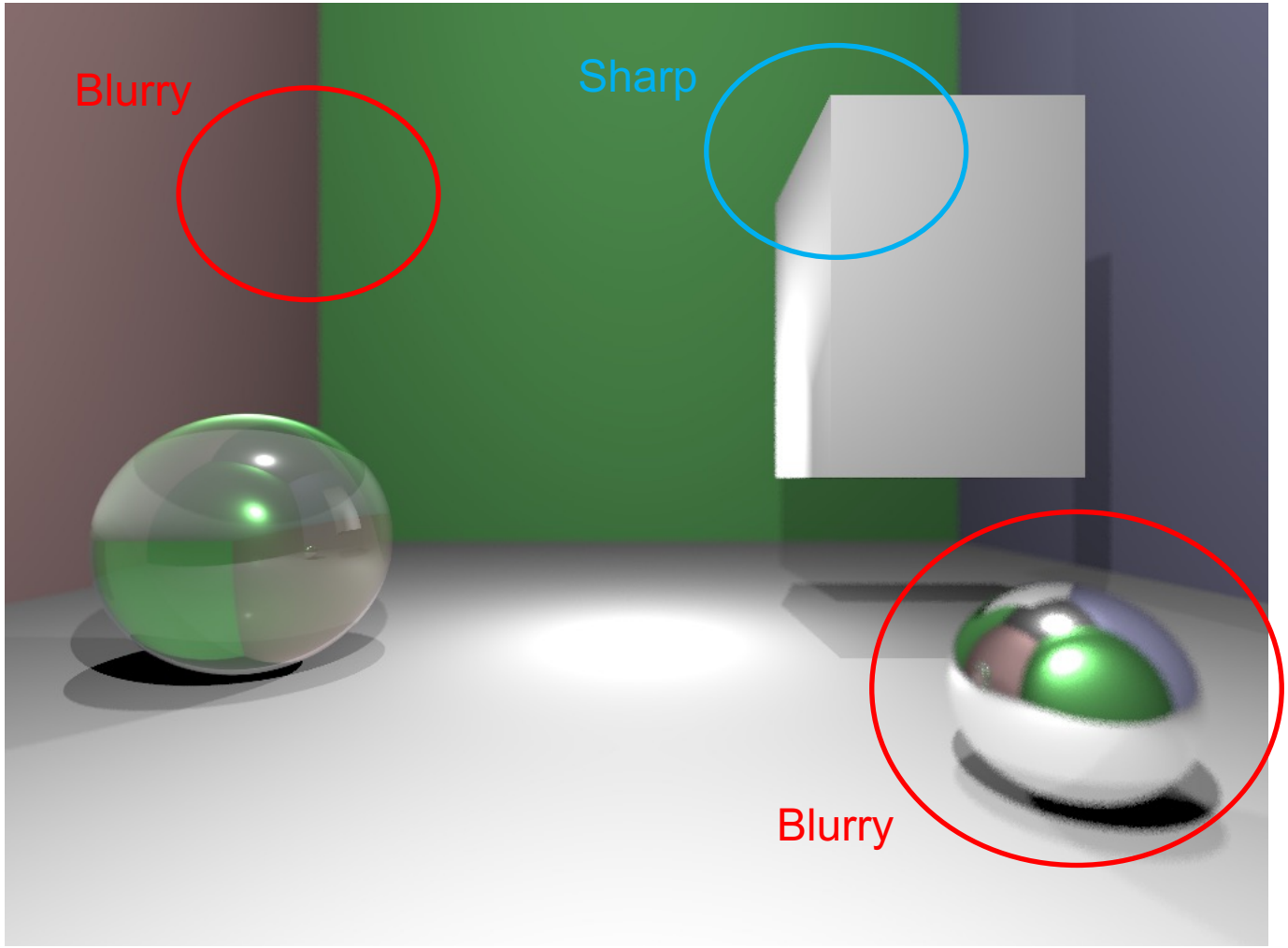


Depth of field results



Exaggeration
(too much aperture)

Depth of field results



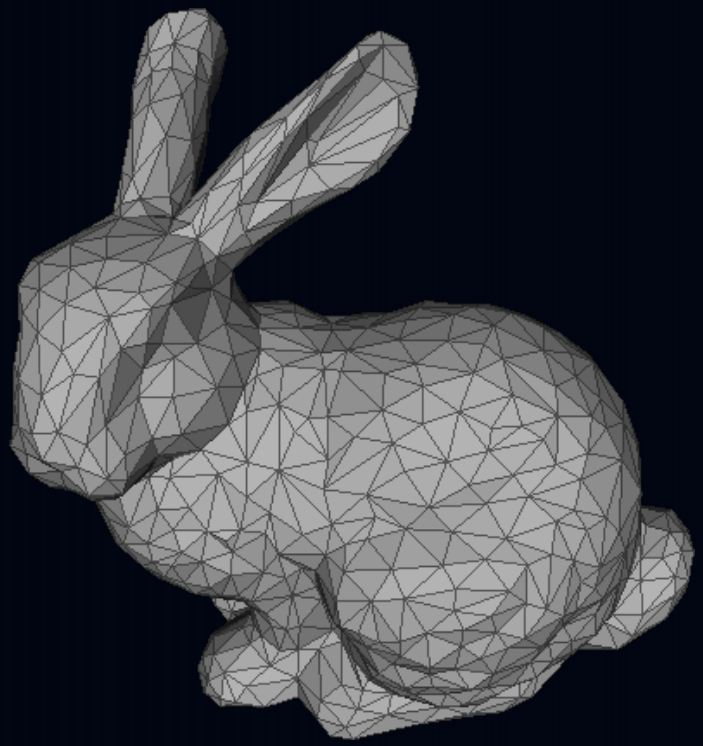
20 rays

Antialiasing + DOF

```
for(i = 0; i < width ; i++){
    for(j = 0; j < height ; j++){
        color = glm::vec3(0, 0, 0);
        for(k = 0; k < T; k++){         // T is the number of antialiasing rays
            dx = X + i*s + s/2;
            dy = Y - j*s - s/2;
            dz = 1;

            // Stochastic anti-aliasing - we shoot multiple rays per pixel randomly
            dx += random(-s/2, s/2);
            dy += random(-s/2, s/2);

            origin = glm::vec3(0, 0, 0);
            direction = glm::vec3(dx, dy, dz);
            direction = glm::normalize(direction);

            focal_point = glm::vec3(focal_dist * direction / direction.z);
            dof_color = glm::vec3(0, 0, 0);

            // Depth of field
            for (l = 0; l < N; l++){            // N is the number of rays for depth of field
                offset = glm::vec3(random(-aperture, aperture), random(-aperture, aperture), 0);
                new_o = origin + offset;
                new_d = glm::normalize(focal_point - new_o);
                Ray ray(new_o, new_d);
                dof_color += trace_ray(ray);
            }
            dof_color /= (float) N;
            color += dof_color;
        }
        image.setPixel(i, j, toneMapping(color / (float) T));
    }
}
```

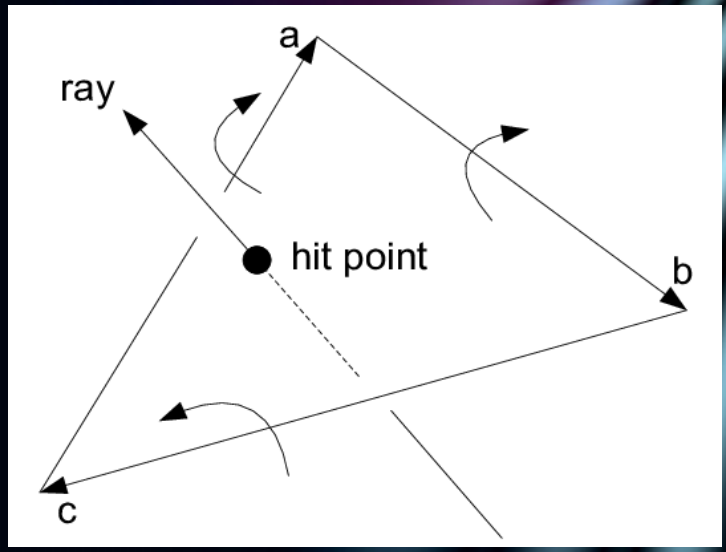
Meshes

- Object countour approximated with triangles.

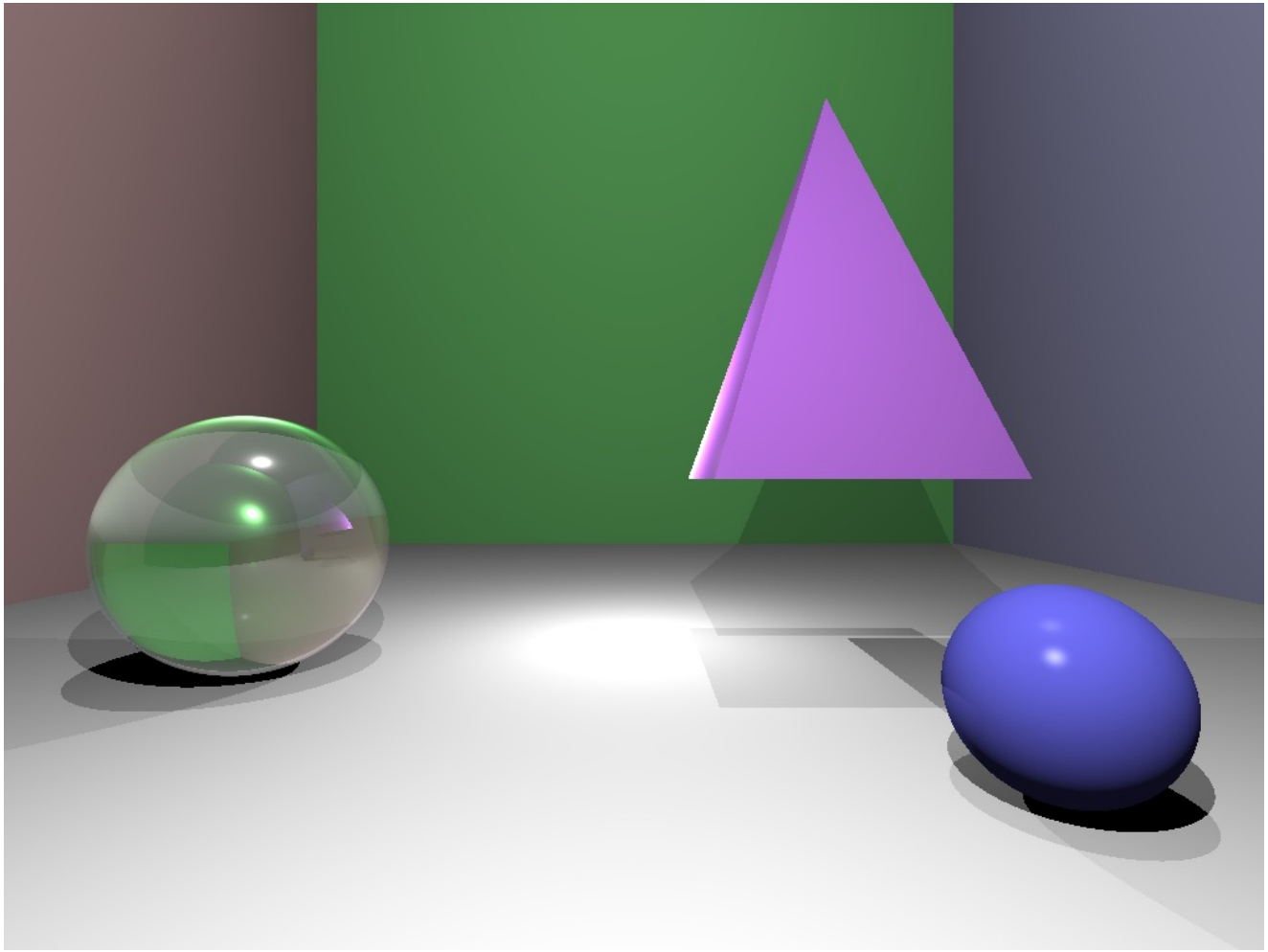


Ray triangle intersection

1. Intersect with plane
2. Check if intersection point is inside triangle (barycentric coordinates)



Ray triangle intersection



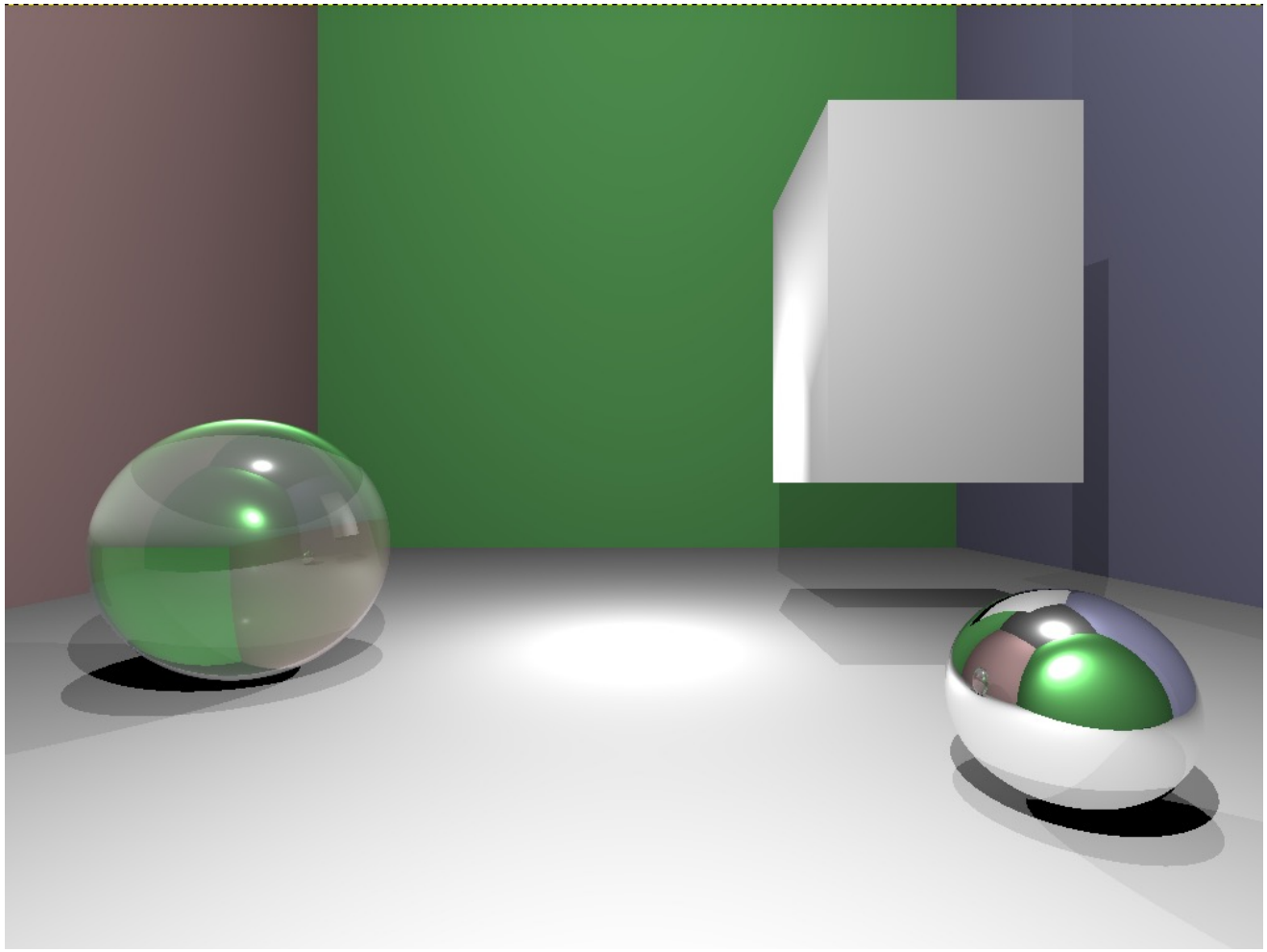
Mesh reader

```
# Blender v2.78 (sub 0) OBJ File: ''
# www.blender.org
mtllib TestObjCube_v2.mtl
o Cube
v 0.000000 0.000000 0.000000
v 0.000000 2.000000 0.000000
v 0.000000 0.000000 -2.000000
v 0.000000 2.000000 -2.000000
v 2.000000 0.000000 0.000000
v 2.000000 2.000000 0.000000
v 2.000000 0.000000 -2.000000
v 2.000000 2.000000 -2.000000
vn -1.0000 0.0000 0.0000
vn 0.0000 0.0000 -1.0000
vn 1.0000 0.0000 0.0000
vn 0.0000 0.0000 1.0000
vn 0.0000 -1.0000 0.0000
vn 0.0000 1.0000 0.0000
usemtl None
s off
f 1//1 2//1 4//1 3//1
f 3//2 4//2 8//2 7//2
f 7//3 8//3 6//3 5//3
f 5//4 6//4 2//4 1//4
f 3//5 7//5 5//5 1//5
f 8//6 4//6 2//6 6//6
```

Mesh Reader

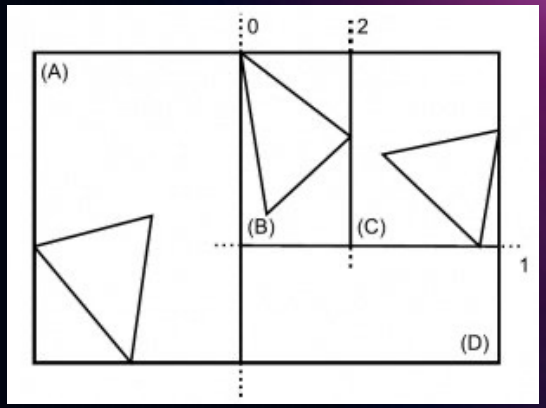
```
# cube.obj
#

g cube

v  0.0  0.0  0.0
v  0.0  0.0  1.0
v  0.0  1.0  0.0
v  0.0  1.0  1.0
v  1.0  0.0  0.0
v  1.0  0.0  1.0
v  1.0  1.0  0.0
v  1.0  1.0  1.0

vn  0.0  0.0  1.0
vn  0.0  0.0 -1.0
vn  0.0  1.0  0.0
vn  0.0 -1.0  0.0
vn  1.0  0.0  0.0
vn -1.0  0.0  0.0

f  1//2  7//2  5//2
f  1//2  3//2  7//2
f  1//6  4//6  3//6
f  1//6  2//6  4//6
f  3//3  8//3  7//3
f  3//3  4//3  8//3
f  5//5  7//5  8//5
f  5//5  8//5  6//5
f  1//4  5//4  6//4
f  1//4  6//4  2//4
f  2//1  6//1  8//1
f  2//1  8//1  4//1
```

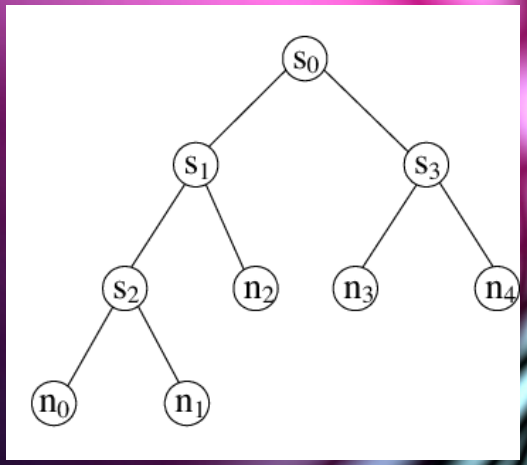
Cube with smooth normal interpolation

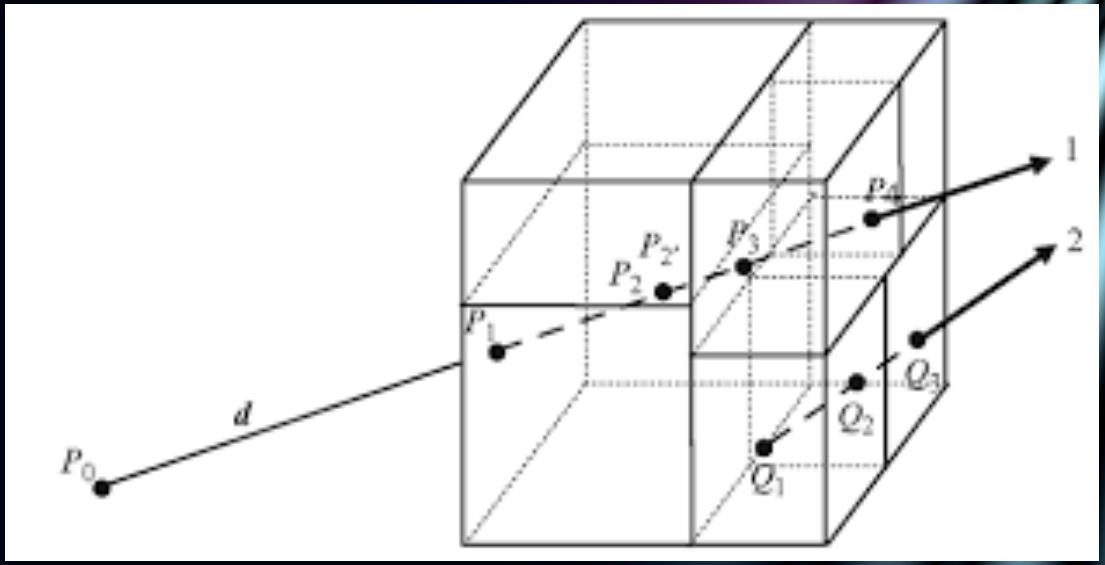


Acceleration structures: Kd-trees

1. When creating a mesh, store triangles using a Kd-tree.
2. Use midpoint as heuristic.
3. To find ray intersections, traverse the tree until the appropiate leaf is found and compute the intersection for each triangle in the leaf.



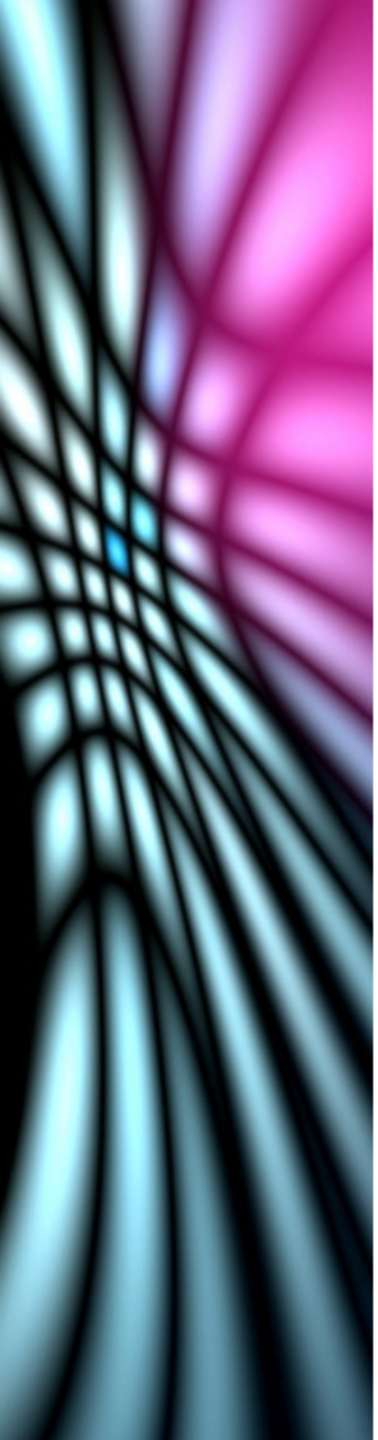




Threads

- Pragma (OpenMP)
  - **One line**
  - **Automatically selects optimal number of threads**
  - **Each thread computer the color of a certain number of pixels.**

```
int i, j, k, l;
glm::vec3 color, origin, direction, focal_point, dof_color, offset, new_o, new_d, ray;
float dx, dy, dz;
#pragma omp parallel for collapse(2) schedule(static) private(i, j, k, l, color, origin, direction, focal_point, dof_color, offset, new_o, new_d, ray, dx, dy, dz)
// schedule static -> divide the loop in chunks and assign them to the threads
for(i = 0; i < width ; i++){
    for(j = 0; j < height ; j++){

        if (i % 50 == 0 && j == 0){
```



Results

- Assignment images
  - **Originally: 10 seconds**
  - **Now: 0 seconds**
- >4000 triangles
  - **1/5 of the original time**

Advanced Reflectance Model: Ward

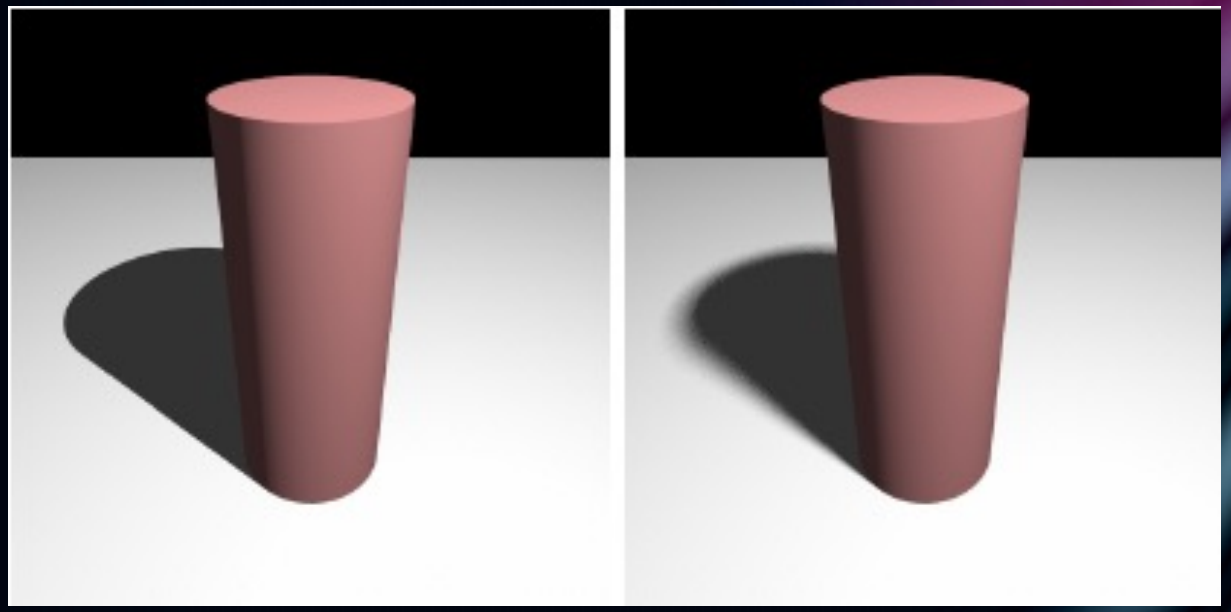
- Applies to specular component
- Diffuse, ambient components don’t change
- Controlled by 2 parameters: alpha_x and alpha_y

$$k_{\text{spec}} = \frac{\rho_s}{\sqrt{(N \cdot L)(N \cdot V)}} \frac{N \cdot L}{4\pi\alpha_x\alpha_y} \exp\left[-2\frac{\left(\frac{H \cdot X}{\alpha_x}\right)^2 + \left(\frac{H \cdot Y}{\alpha_y}\right)^2}{1 + (H \cdot N)}\right]$$

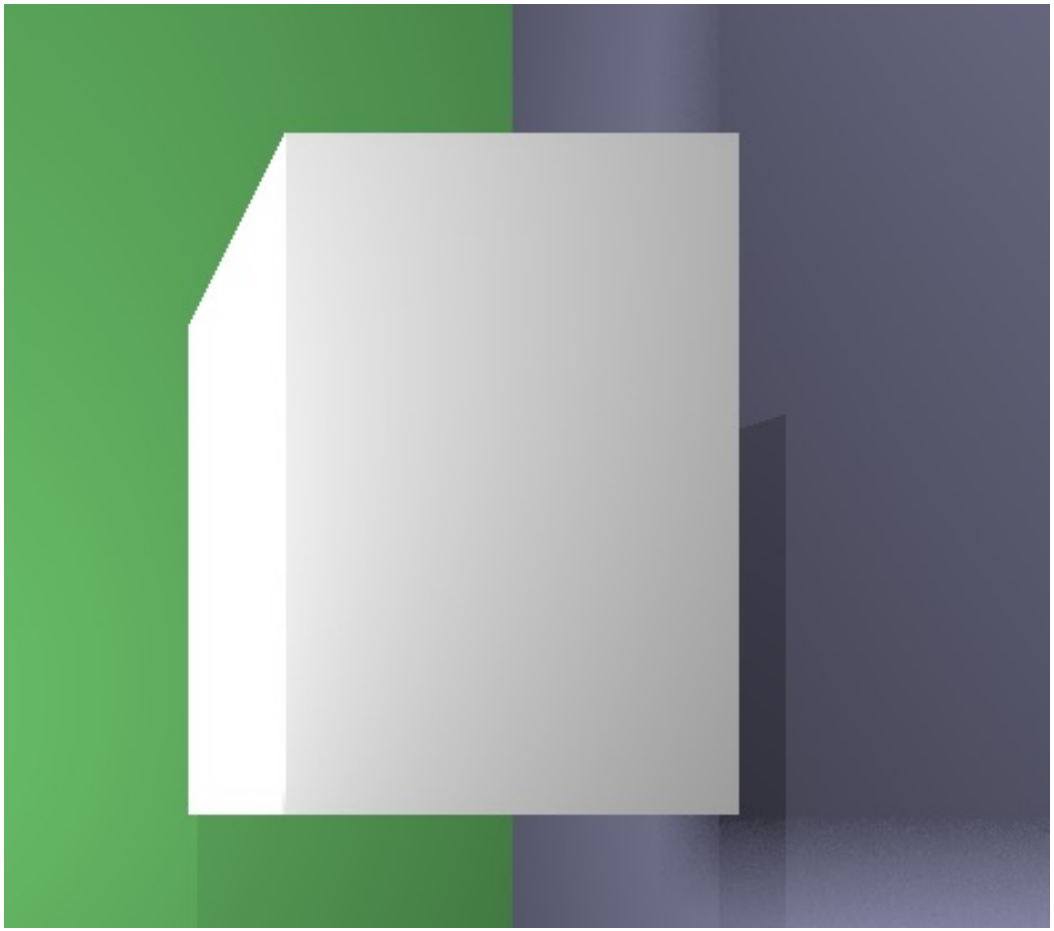
Ward model (blue ceramic)

Soft shadows

- Area Light
- Blurry borders, different shades of gray
- [0, 1] instead of {0, 1}



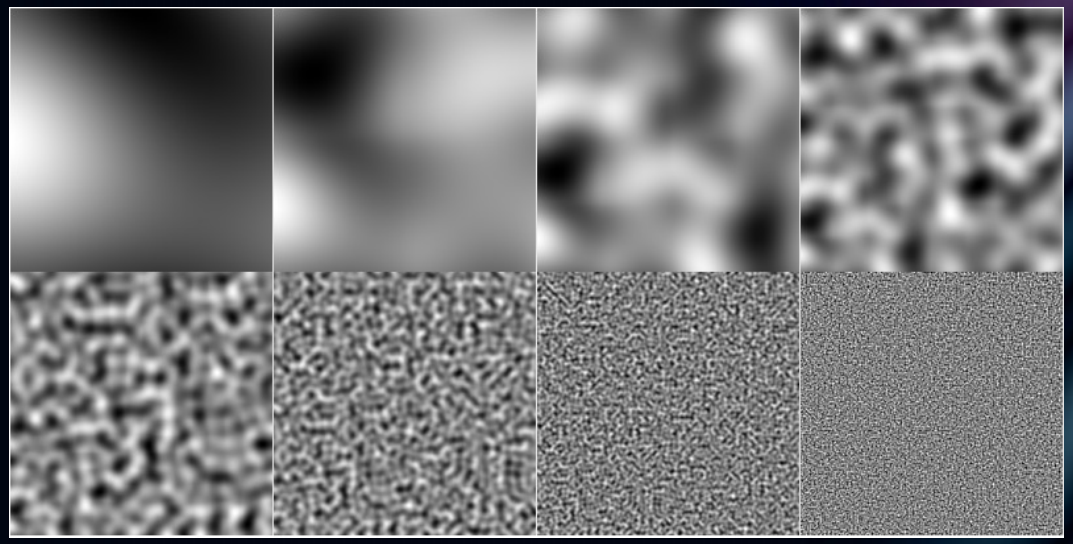
Results: too few samples



Nice results

Perlin Noise

- Create textures from the topology itself instead of from an external source.
- Generate noise in a semi-random way.
- Use the noise to create more advanced patterns.

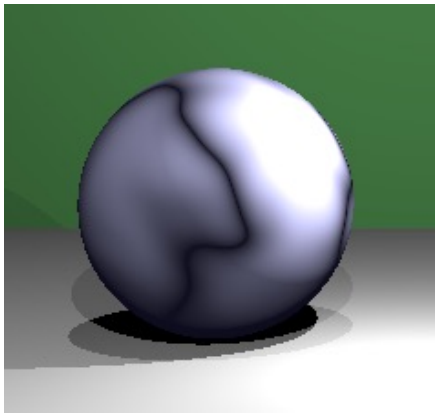


Implementation

```
float generateNoise(float x, float y, float z){
    int X = ((int) floor(x)) & 255;     // Equivalent to mod 256, but quicker than %
    int Y = ((int) floor(y)) & 255;
    int Z = ((int) floor(z)) & 255;

    x -=  ((int) floor(x));
    y -= ((int) floor(y));
    z -= ((int) floor(z));

    // Get the hash of the 8 corners of the cube
    int A = p[X] + Y;
    int AA = p[A] + Z;
    int AB = p[A + 1] + Z;
    int B = p[X + 1] + Y;
    int BA = p[B] + Z;
    int BB = p[B + 1] + Z;

    // Apply a sigmoid to the corners to get a smooth interpolation
    float u = fade(x);
    float v = fade(y);
    float w = fade(z);

    // Interpolate corners pair by pair, using u, v and w as weights
    return lerp(w,  lerp(v, lerp(u, grad(p[AA], glm::vec3(x, y, z)),
                                    grad(p[BA], glm::vec3(x - 1, y, z))),
                            lerp(u, grad(p[AB], glm::vec3(x, y - 1, z)),
                                    grad(p[BB], glm::vec3(x - 1, y - 1, z)))),
                    lerp(v, lerp(u, grad(p[AA + 1], glm::vec3(x, y, z - 1)),
                                    grad(p[BA + 1], glm::vec3(x - 1, y, z - 1))),
                            lerp(u, grad(p[AB + 1], glm::vec3(x, y - 1, z - 1)),
                                    grad(p[BB + 1], glm::vec3(x - 1, y - 1, z - 1))))
                );
}
```

Perlin Noise results

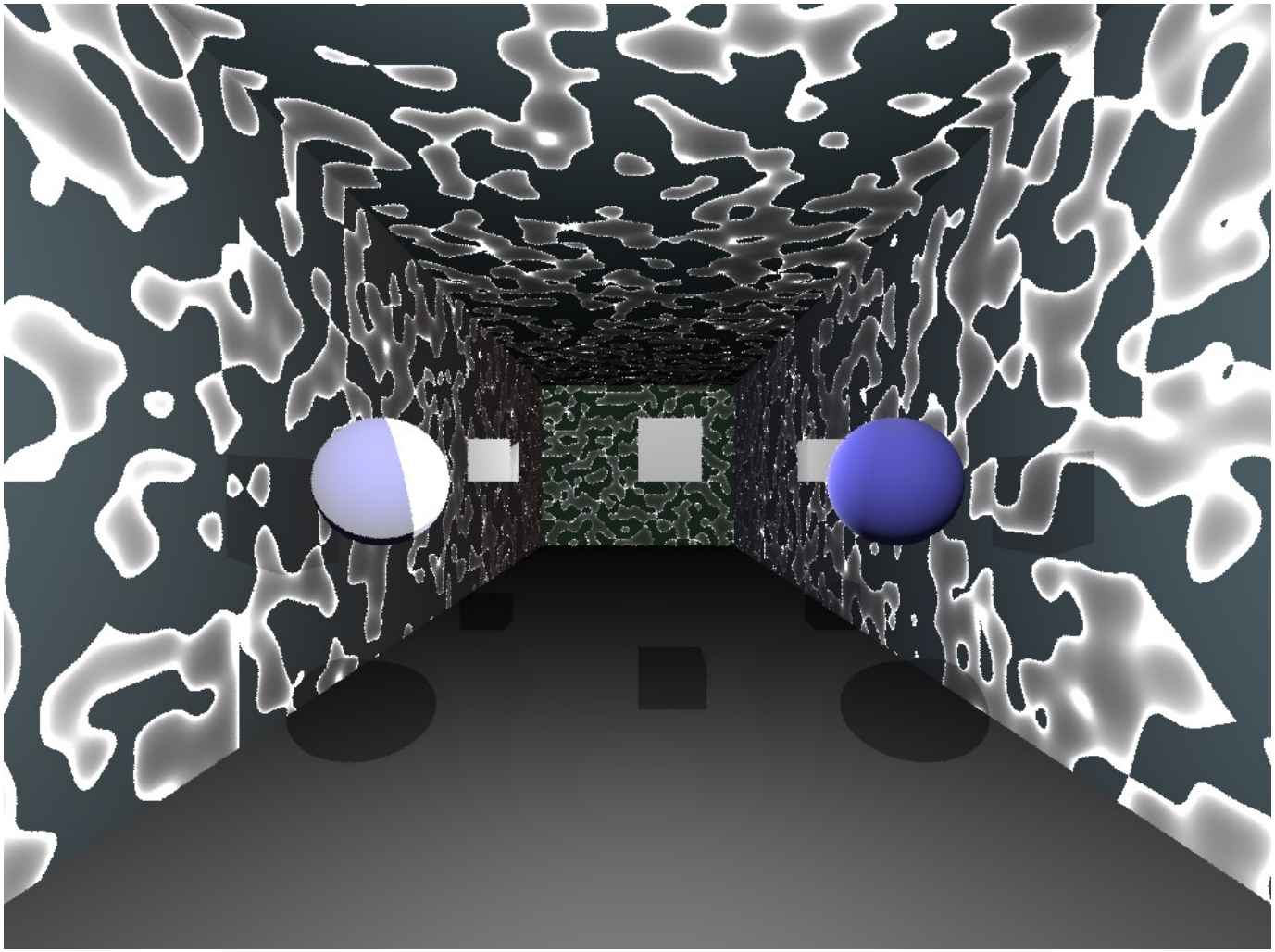
Spotted



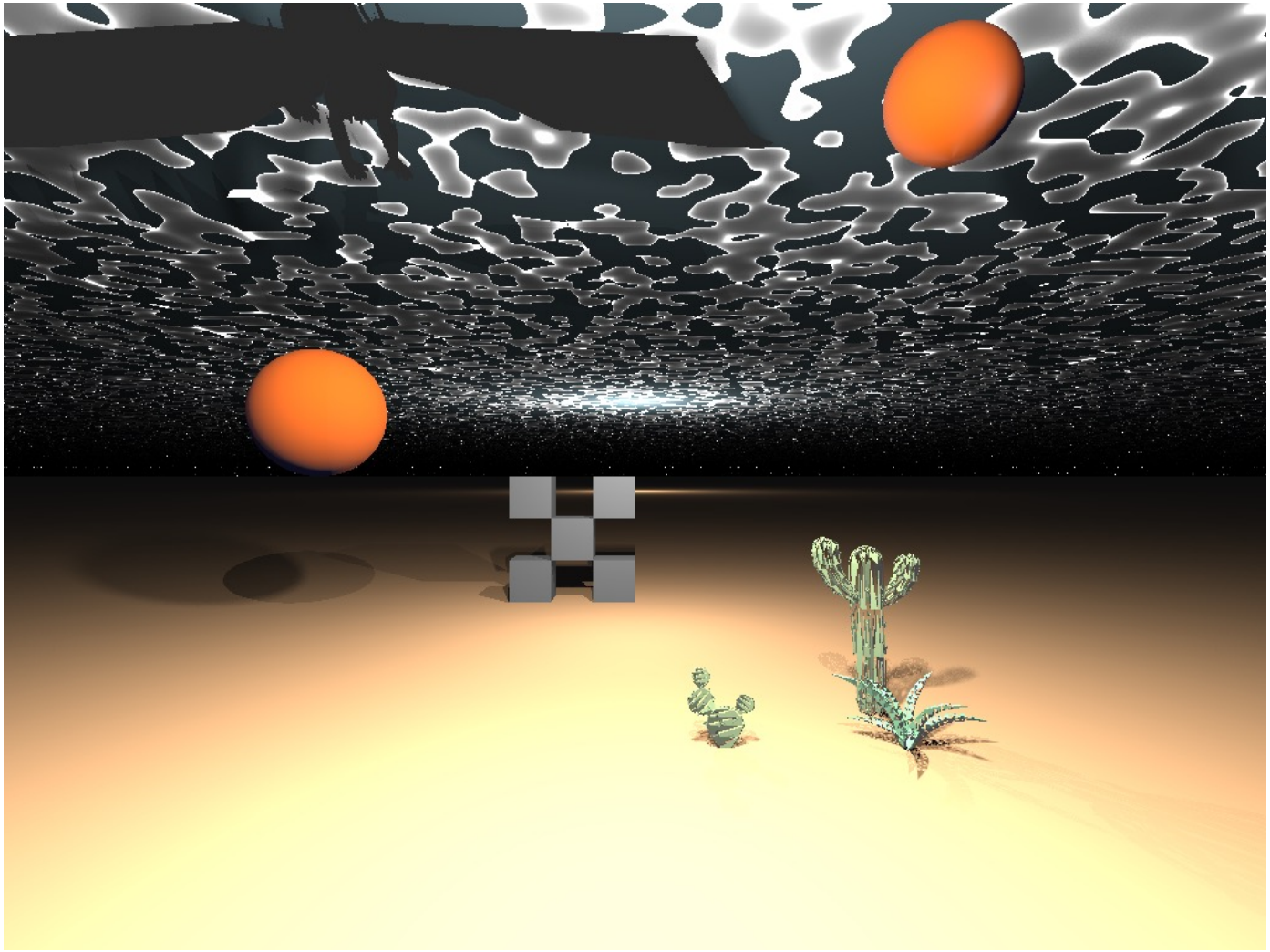
Marble

Sky with clouds

Some experiments



Final image



Thank you!