Raytracer Report

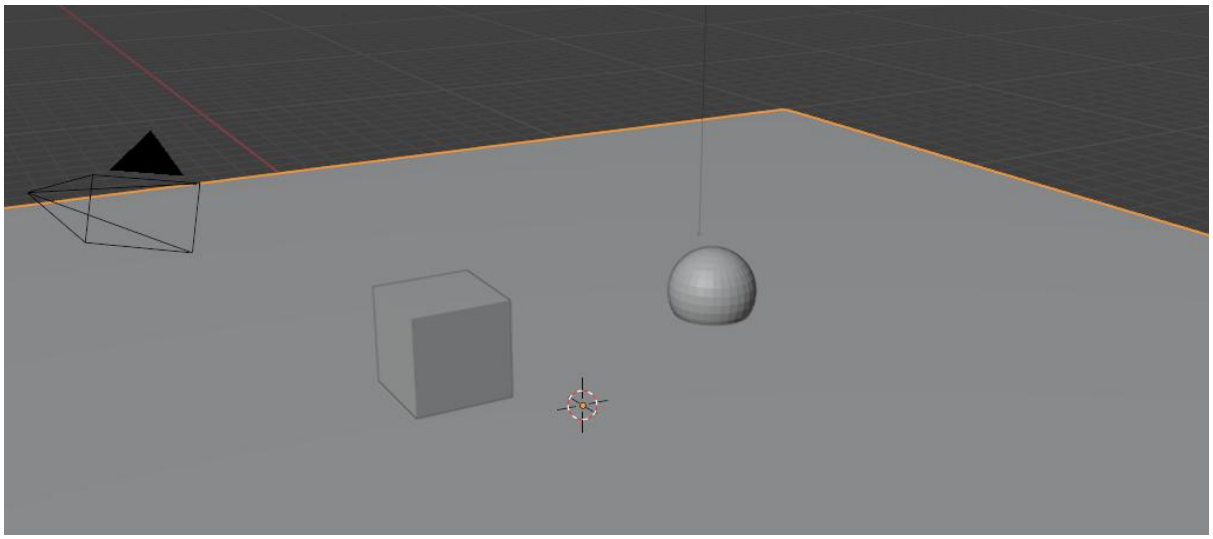
s2281597

Module 1	Completed?
Blender exporter	
Camera space	
Image R/W	
Module 2	
Ray intersection	
Acceleration	
Module 3	
Whitted-style RT	
Anti-aliasing	
Texture mapping	
Final	
System Integration	
Distributed RT	
Lens Effects	

Module 1

Blender Exporter

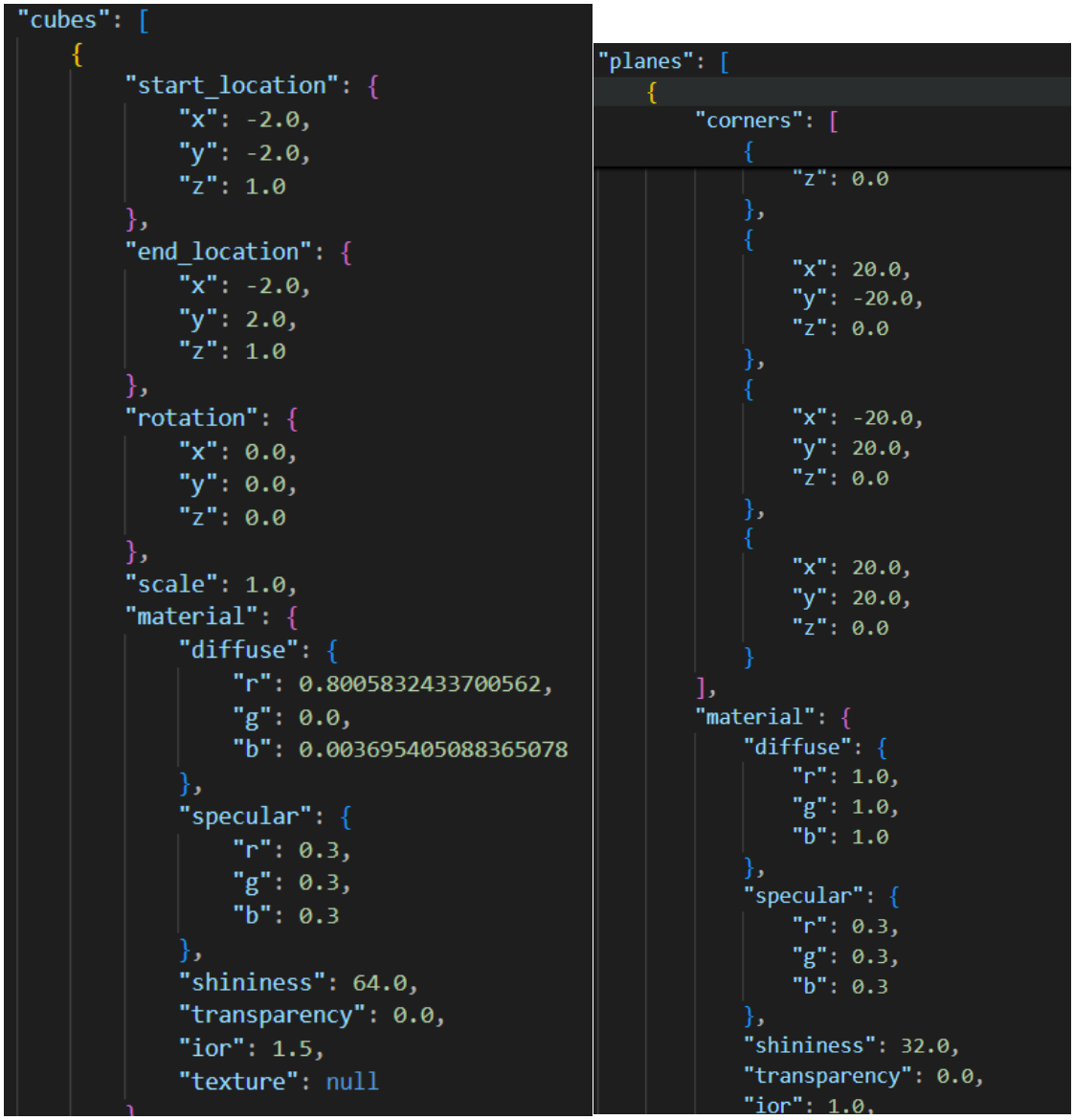
This functions via a Python script that extracts data from a given Blender scene e.g. camera direction vectors, focal distance, resolution, shape translation, rotation and scale, and material & texture information. This is exported into a JSON which is parsed by functions in each object type.



Exported Blender scene (exported.blend)

```
{
    "properties": {
        "cameras": [
            {
                "location": {
                    "x": 10.0,
                    "y": -10.0,
                    "z": 8.0
                },
                "gaze_vector": {
                    "x": -0.6556180119514465,
                    "y": 0.6556179523468018,
                    "z": -0.3746066391468048
                },
                "up_vector": {
                    "x": -0.26488688588142395,
                    "y": 0.26488688588142395,
                    "z": 0.9271838665008545
                },
                "aperture": 0.5,
                "focal_distance": 10.75083065032959,
                "focal_length": 50.0,
                "sensor": {
                    "width": 36.0,
                    "height": 24.0
                },
                "film_resolution": {
                    "width": 1920,
                    "height": 1080
                }
            }
        ],
```

```
"point_lights": [
    {
        "location": {
            "x": -8.0,
            "y": 8.0,
            "z": 8.0
        },
        "radiant_intensity": 1000.0
    }
],
"spheres": [
    {
        "start_location": {
            "x": -3.0,
            "y": 5.0,
            "z": 0.5
        },
        "end_location": {
            "x": -10.0,
            "y": -5.0,
            "z": 0.5
        },
        "radius": 1.0,
        "material": {
            "diffuse": {
                "r": 0.0,
                "g": 0.0014553972287103534,
                "b": 0.8002356290817261
            },
            "specular": {
                "r": 0.3,
                "g": 0.3,
                "b": 0.3
            },
            "shininess": 115.19999980926514,
            "transparency": 0.0,
```



Extracted JSON

Example AI Coding Assistant Prompts

"I would like to extract camera, point light, sphere, cube and plane data from a Blender scene into a JSON file, using a Python script. Could you write this for me?

Changes: This was improved incrementally to include other data such as material data and lens aperture. This was done mostly manually, with coding assistant help to find specific property names for Blender objects.

Camera Space Transformations

The Camera object parses data for camera in the scene into the object's properties. Given a pixel, a method has been implemented that will create a ray in world space corresponding to that pixel (origin and direction).

```
vector<Camera> Camera::parseCameraDataFromJson(Config config) {
    for (int i = 0; i < cameraObjects.size(); i++){

        //Parse location
        string locationStr = getJSONObject(cameraDataStr, "\"location\"");
        newCam.location = {getFloat(locationStr, "\"x\""),
        getFloat(locationStr, "\"y\""),
        getFloat(locationStr, "\"z\"")};

        //Parse gaze vector
        string gazeVectorStr = getJSONObject(cameraDataStr, "\"gaze_vector\"");
        newCam.gaze_vector = {getFloat(gazeVectorStr, "\"x\""),
        getFloat(gazeVectorStr, "\"y\""),
        getFloat(gazeVectorStr, "\"z\"")};

        //Parse camera aperture
        if (config.dof){
            newCam.aperture = getFloat(cameraDataStr, "\"aperture\"");
        }
        else{
            newCam.aperture = 0.0f;
        }

        //Parse focal length
        newCam.focal_length = getFloat(cameraDataStr, "\"focal_length\"") / 1000.0f;
        newCam.focal_distance = getFloat(cameraDataStr, "\"focal_distance\"");

        //Parse sensor width and height
        string sensorStr = getJSONObject(cameraDataStr, "\"sensor\"");
        newCam.sensor_width = getFloat(sensorStr, "\"width\"") / 1000.0f;
        newCam.sensor_height = getFloat(sensorStr, "\"height\"") / 1000.0f;

        //Parse resolution
        string filmResolutionStr = getJSONObject(cameraDataStr, "\"film_resolution\"");
        newCam.res_x = getFloat(filmResolutionStr, "\"width\"");
        newCam.res_y = getFloat(filmResolutionStr, "\"height\"");
```

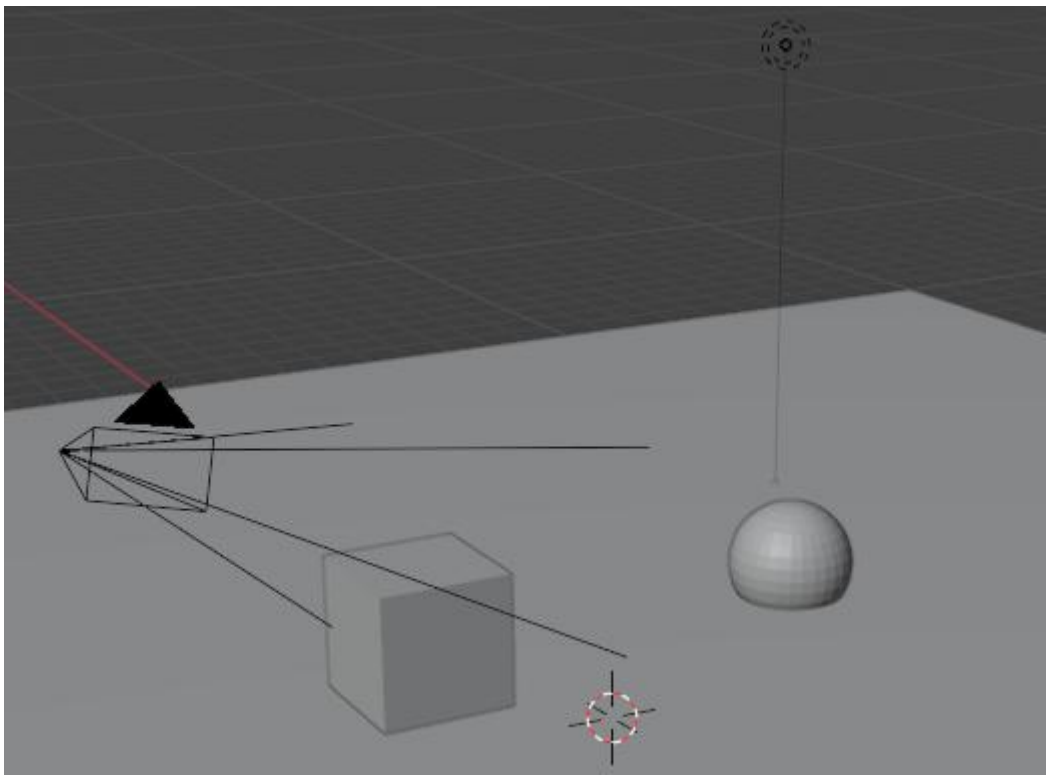
Camera data parsing code snippet

```
Camera location: (10, -10, 8)
Gaze vector: [-0.655618, 0.655618, -0.374607]
Lens aperture: 0
Focal length: 10.7508
Sensor width: 0.036
Sensor height: 0.024
Resolution: (1920, 1080)
```

Extracted camera data (printed from object properties)

```
Ray 1 origin: (10, -10, 8)
Ray 1 direction: [-0.796123, 0.547565, -0.257607]
Ray 2 origin: (10, -10, 8)
Ray 2 direction: [-0.54727, 0.796331, -0.25759]
Ray 3 origin: (10, -10, 8)
Ray 3 direction: [-0.48511, 0.734158, -0.475057]
Ray 4 origin: (10, -10, 8)
Ray 4 direction: [-0.733946, 0.485401, -0.475088]
```

Rays created for each quadrant of a 1920x1080 camera's image plane



Rays visualised in scene

Image R/W

The Image class has a constructor taking a PPM file from the "Textures" folder of the project structure, with methods to read and edit individual pixel colours given the pixel and RGB values, and a method to output it as a PPM file with a chosen name

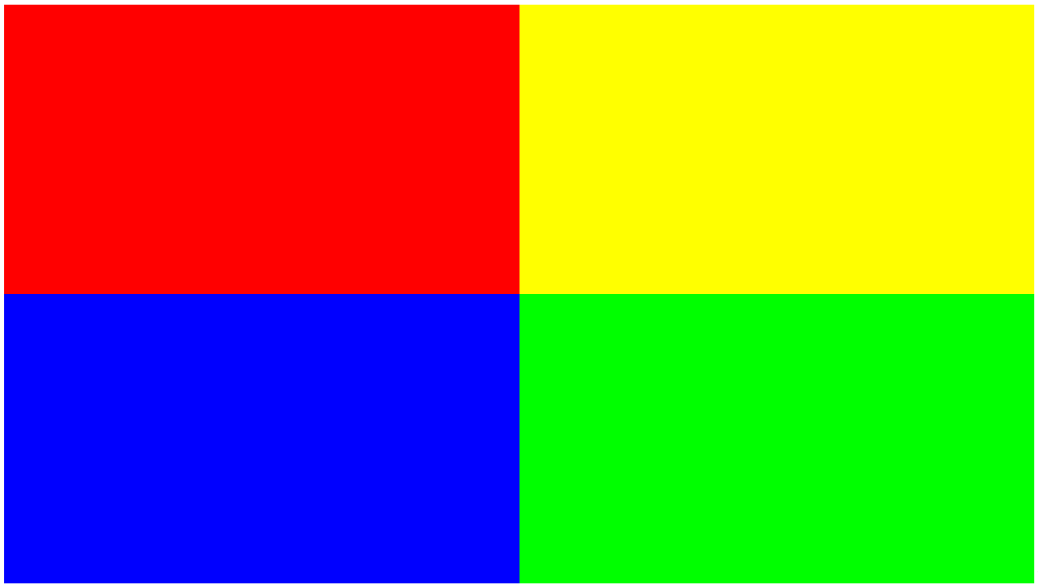
```
Image::Image(const string& fileName){
    file = fileName;
}
```

Image constructor

Pre-edit PPM image (blank_1920x1080.ppm)

```
for (int x = 0; x < 960; x++){
    for (int y = 0; y < 540; y++){
        img.modifyPixel(x, y, 255, 0, 0);
    }
}

for (int x = 960; x < 1920; x++){
    for (int y = 540; y < 1080; y++){
        img.modifyPixel(x, y, 0, 255, 0);
    }
}

for (int x = 0; x < 960; x++){
    for (int y = 540; y < 1080; y++){
        img.modifyPixel(x, y, 0, 0, 255);
    }
}

for (int x = 960; x < 1920; x++){
    for (int y = 0; y < 540; y++){
        img.modifyPixel(x, y, 255, 255, 0);
    }
}

img.writeImage("colours.ppm");
```

Sample pixel edit code



Resulting image (colours.ppm)

Module 2

Ray intersection

A Shape superclass was implemented, containing data consistent across planes, spheres and cubes. Subclasses for each shape were implemented to include shape-specific data e.g. corners for planes, along with specific JSON parsing, intersection and transform methods.

A hit structure class was implemented to store point hit in 3D space, hit distance along ray, the material data of the object hit, the position on the object's face hit (for texture mapping) and the time the ray occurred at (for motion blur)

```
float thc = sqrt(radius*radius - d2);
float t0 = tca - thc;
float t1 = tca + thc;

float t = (t0 > 0) ? t0 : t1;
if (t < 0) return false;

vector<float> intersectPoint = {ray.origin[0] + t * ray.direction[0], ray.origin[1] + t * ray.d

if (hasTex){
    //Get hit point on texture for texture mapping
    vector<float> normal = Raytracer::normalise(Raytracer::sub_vec(intersectPoint, location));
    float u = 0.5f + atan2(normal[2], normal[0]) / (2 * M_PI);
    float v = 0.5f - asin(normal[1]) / M_PI;

    hs.u = u;
    hs.v = v;
    hs.textureFile = texture;
    hs.hasTex = true;
}

hs.hitPoint = intersectPoint;
hs.normal = {(intersectPoint[0] - location[0]) / radius, (intersectPoint[1] - location[1]) / ra
hs.rayDistance = sqrt(pow(ray.origin[0] - intersectPoint[0], 2) + pow(ray.origin[1] - intersect
hs.diffuse = {diffuse[0], diffuse[1], diffuse[2]};
hs.specular = {specular[0], specular[1], specular[2]};
hs.shininess = shininess;
hs.transparency = transparency;
hs.ior = ior;
if (ray.time){
    hs.time = ray.time;
}
return true;
```

Storing of ray data in hit structure, and intersection true/false (sphere.cpp)

```
//Transform ray into local space
Ray local = ray;

//Translate
local.origin = Raytracer::sub_vec(local.origin, location);

//Inverse rotate
vector<float> o = {local.origin[0], local.origin[1], local.origin[2]};
vector<float> d = {local.direction[0], local.direction[1], local.direction[2]};

rotateXYZInverse(o, rotation);
rotateXYZInverse(d, rotation);

local.origin = Raytracer::mul_vec(o, 1.0f / scale);
local.direction = Raytracer::mul_vec(d, 1.0f / scale);

local.direction = Raytracer::normalise(local.direction);
```

Transforming ray into local space before intersection checking – to account for shape transformations (cube.cpp)

Acceleration hierarchy

A bounded-volume hierarchy (BVH) was created using axis-aligned bounded boxes (AABBs) to accelerate intersection checking. Previously, for each ray, every shape was checked for intersection.

```cpp
bool BVHNode::intersect(Ray& ray, float& tMin, float& tMax, Config config){
    //Check if ray intersects this node's bounding box
    if (!aabb.intersect(ray, tMin, tMax)) {
        return false;
    }

    bool hit = false;
    float tLeftMin = tMin, tLeftMax = tMax;
    float tRightMin = tMin, tRightMax = tMax;

    //Recursively check left child
    if (left && left->intersect(ray, tLeftMin, tLeftMax, config)) {
        tMin = tLeftMin;
        tMax = tLeftMax;
        hit = true;
    }

    //Recursively check right child
    if (right && right->intersect(ray, tRightMin, tRightMax, config)) {
        if (!hit || tRightMin < tMin) {
            tMin = tRightMin;
            tMax = tRightMax;
        }
        hit = true;
    }

    //If hit an internal child, return true
    if (hit) return true;
```

Recursive intersection checking in BVH (bvh.cpp)

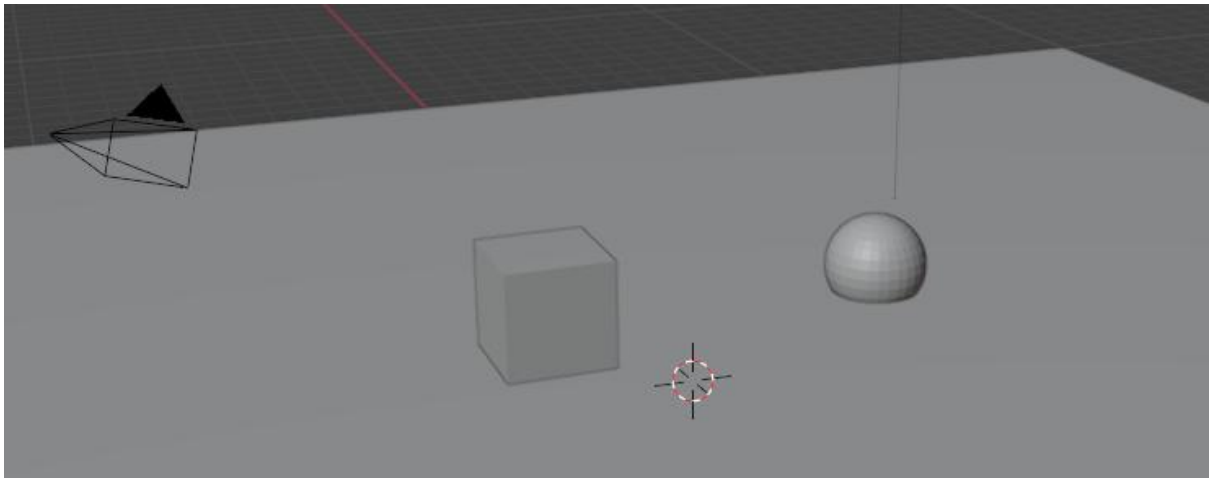
```
//Check intersection with spheres
for (int s = 0; s < spheres.size(); s++){

    HitStructure temp;
    intersect = spheres[s].intersect(ray, temp, config);

    if (intersect && temp.rayDistance < closestDist){
        closestDist = temp.rayDistance;
        hs = temp;
        hit = true;
    }
}

if (hit) {
    ray.hs.push_back(hs);
    return true;
}
else{
    return false;
}
```

```cpp
//Unaccelerated object-by-object intersection testing
bool Raytracer::unacceleratedIntersection(Ray& ray, vector<Plane>& pl

    HitStructure hs;
    bool hit = false;
    float closestDist = FLT_MAX;
    bool intersect;

    //Check intersection with planes
    for (int p = 0; p < planes.size(); p++){

        HitStructure temp;
        intersect = planes[p].intersect(ray, temp, config);

        if (intersect && temp.rayDistance < closestDist){

            closestDist = temp.rayDistance;
            hs = temp;
            hit = true;
        }
    }

    //Check intersection with cubes
    for (int c = 0; c < cubes.size(); c++){

        HitStructure temp;
        intersect = cubes[c].intersect(ray, temp, config);

        if (intersect && temp.rayDistance < closestDist){
            closestDist = temp.rayDistance;
            hs = temp;
            hit = true;
        }
```

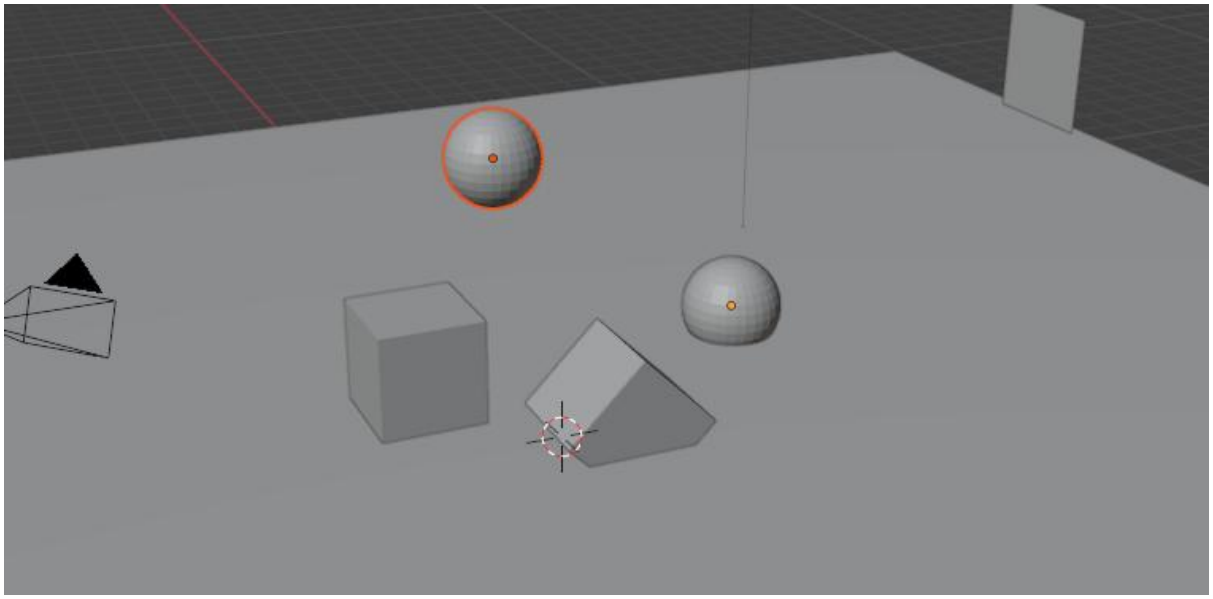
```cpp
//Check intersection with spheres
for (int s = 0; s < spheres.size(); s++){

    HitStructure temp;
    intersect = spheres[s].intersect(ray, temp, config);

    if (intersect && temp.rayDistance < closestDist){
        closestDist = temp.rayDistance;
        hs = temp;
        hit = true;
    }
}

if (hit) {
    ray.hs.push_back(hs);
    return true;
}
else{
    return false;
}
```

Naïve unaccelerated intersection checking (raytracer.cpp)

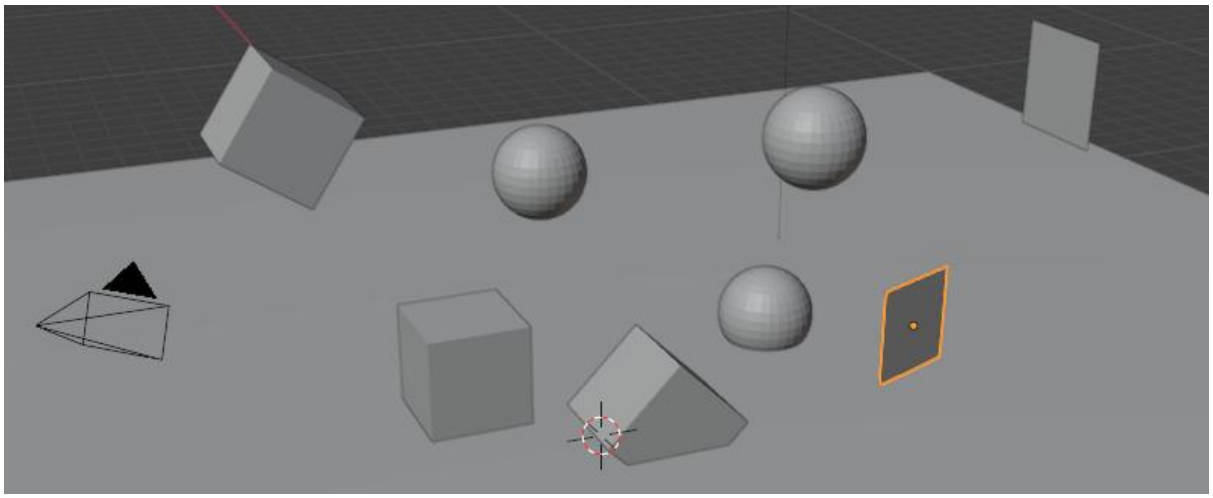
Performance comparison:



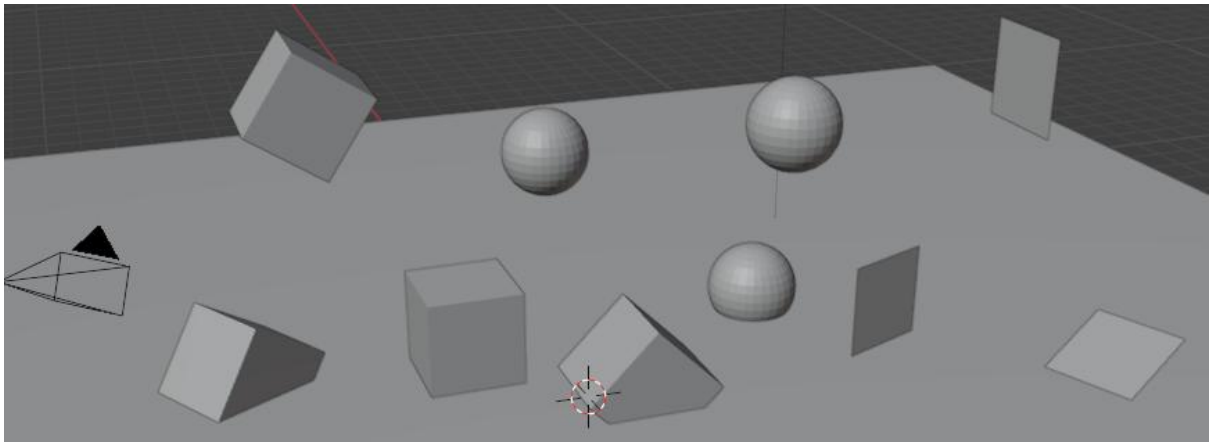
3 object scene (bvh_vs_unaccel_1.blend)



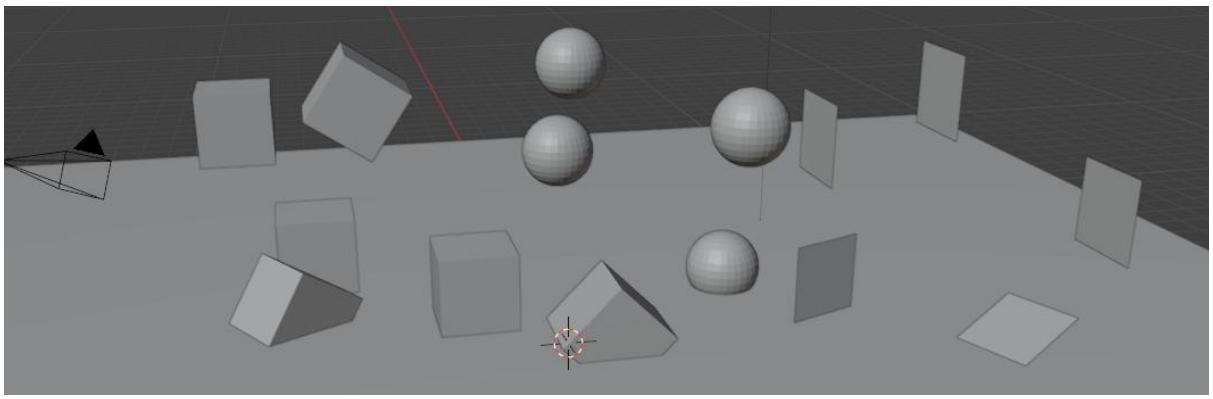
6 object scene (bvh_vs_unaccel_2.blend)



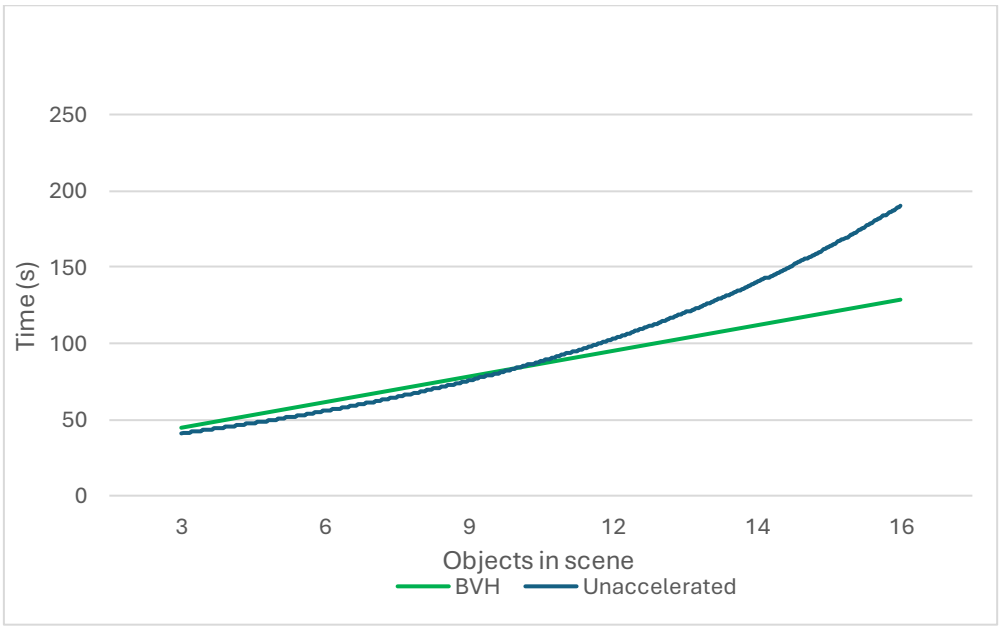
9 object scene (bvh_vs_unaccel_3.blend)



12 object scene (bvh_vs_unaccel_4.blend)



16 object scene (bvh_vs_unaccel_5.blend)

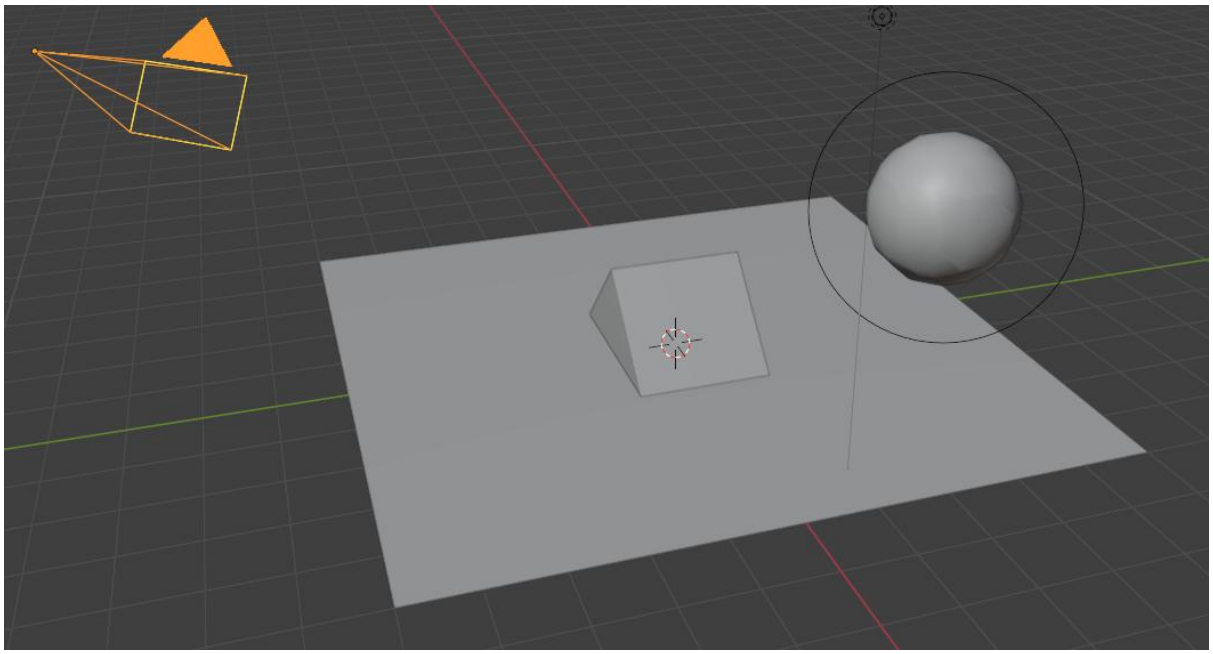


Total raytracing time

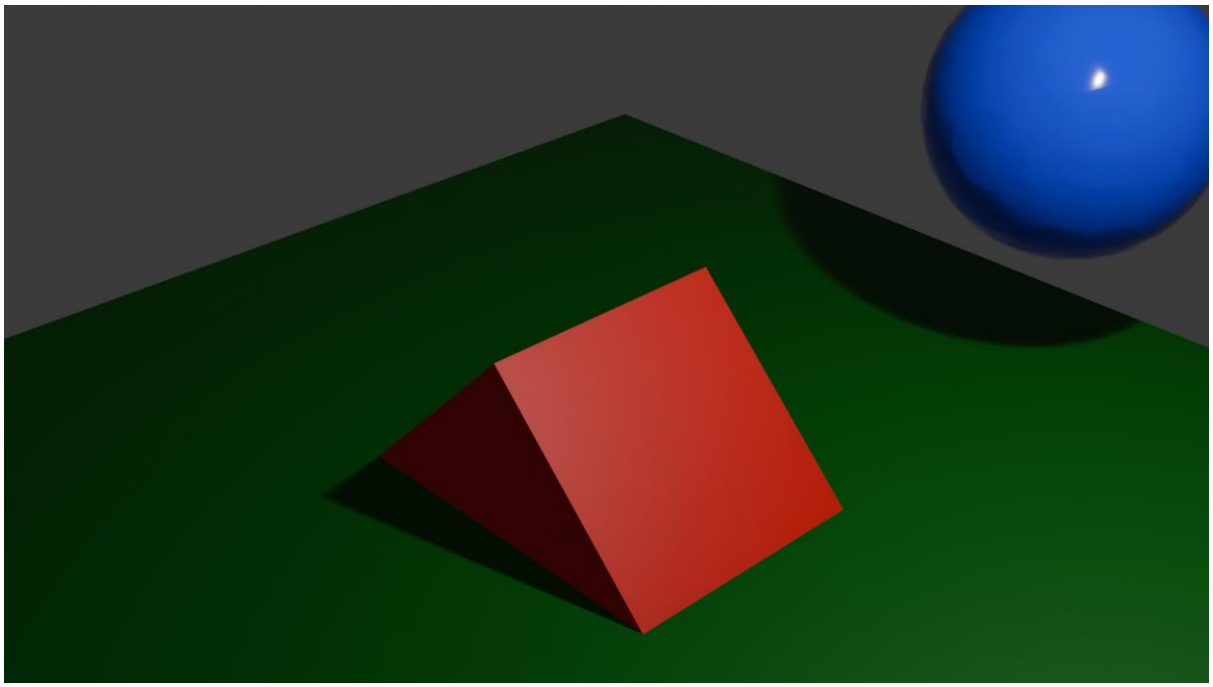
Module 3

Whitted-style raytracing

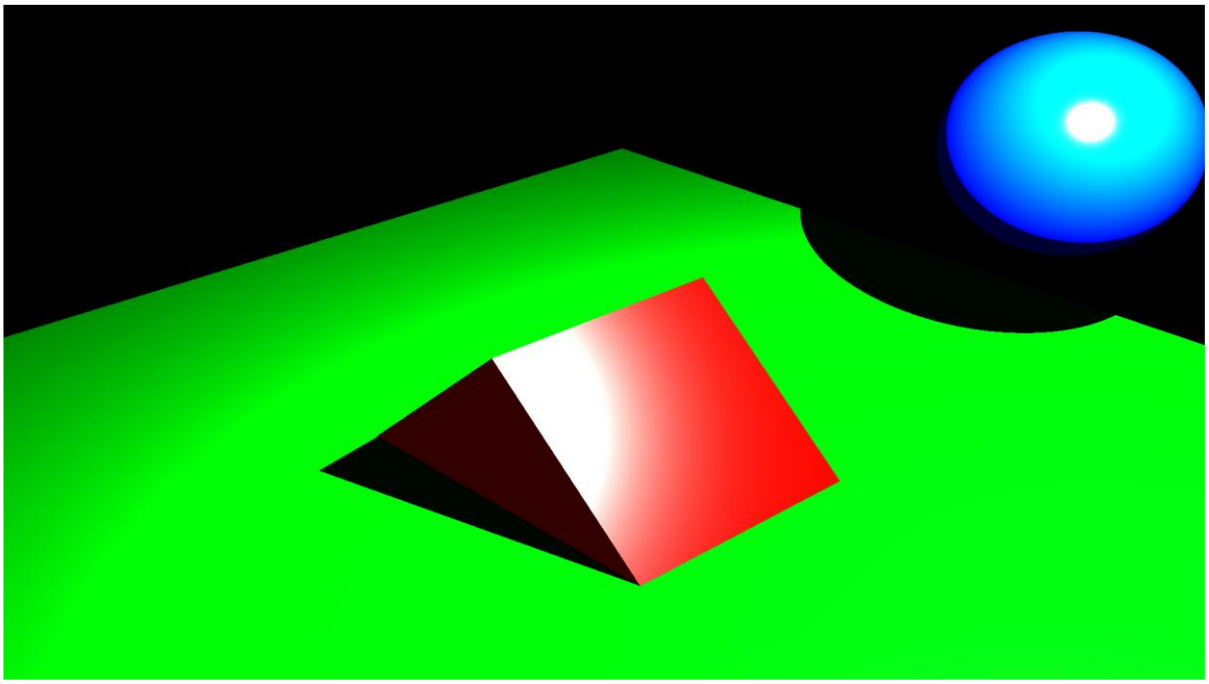
Previous features have been combined to trace rays into a scene and check for shape intersections. Intersections are shaded according to Blinn-Phong, and reflection and refraction rays are cast recursively (based on max depth and object material)



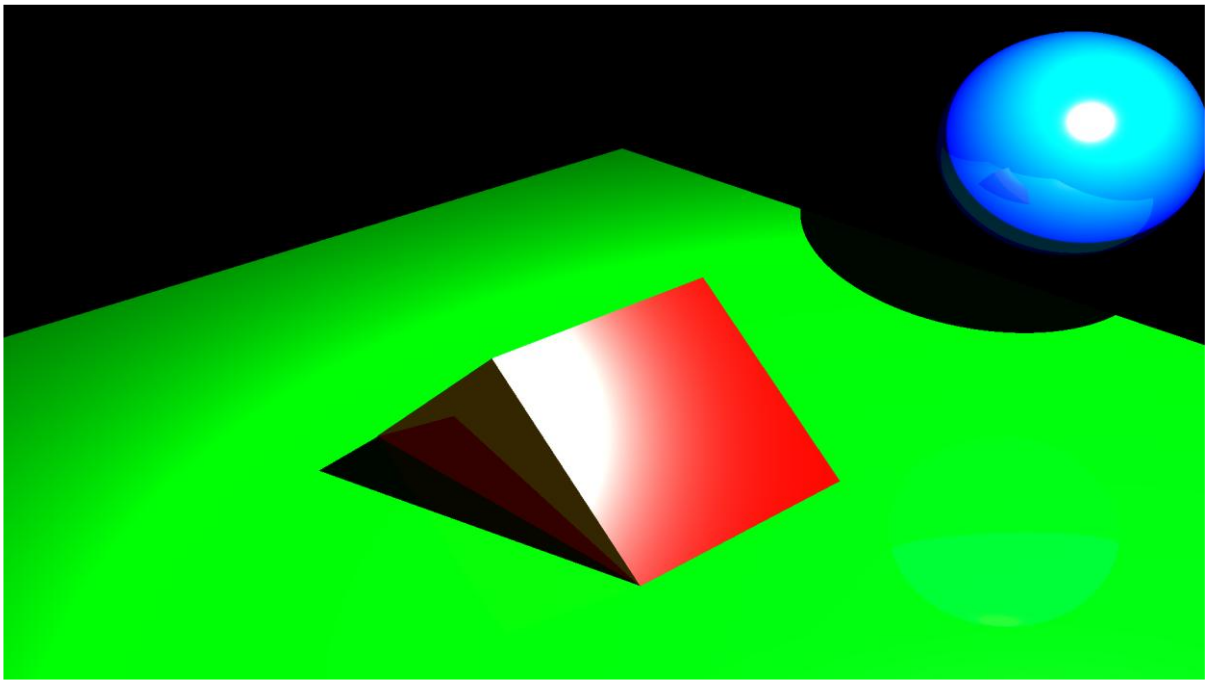
Blender scene (test_personal_1.blend)



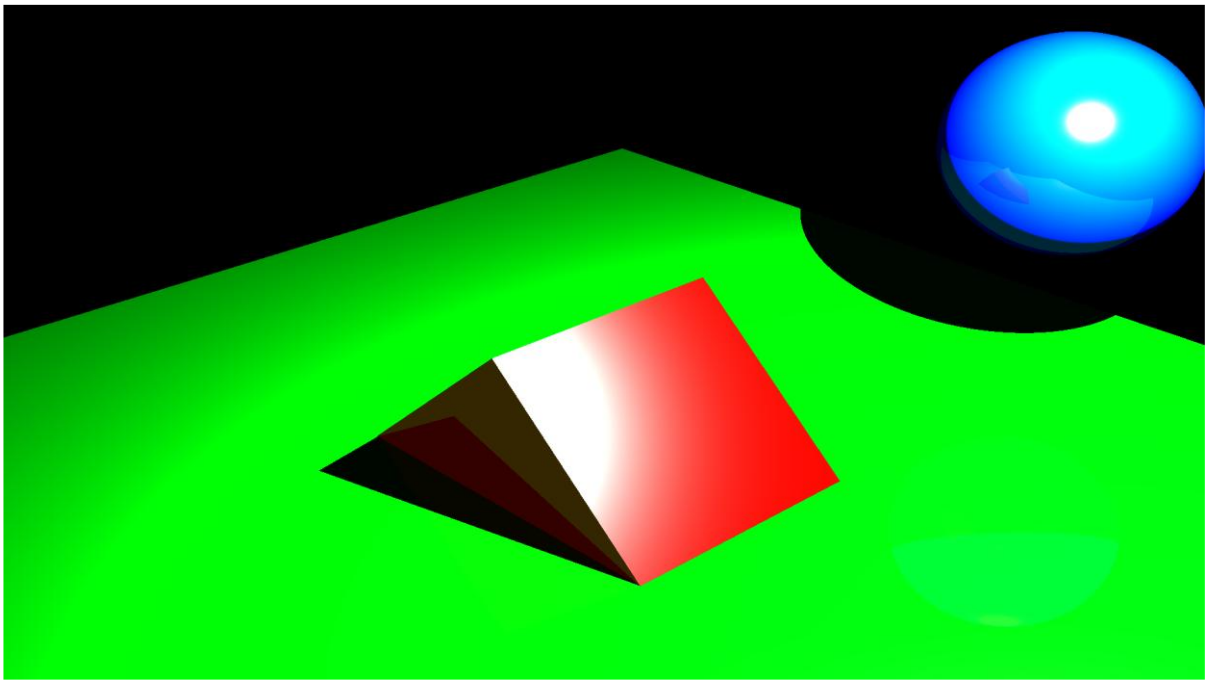
Blender render



Whitted-style render with no reflection/refraction (whitted_style.ppm)



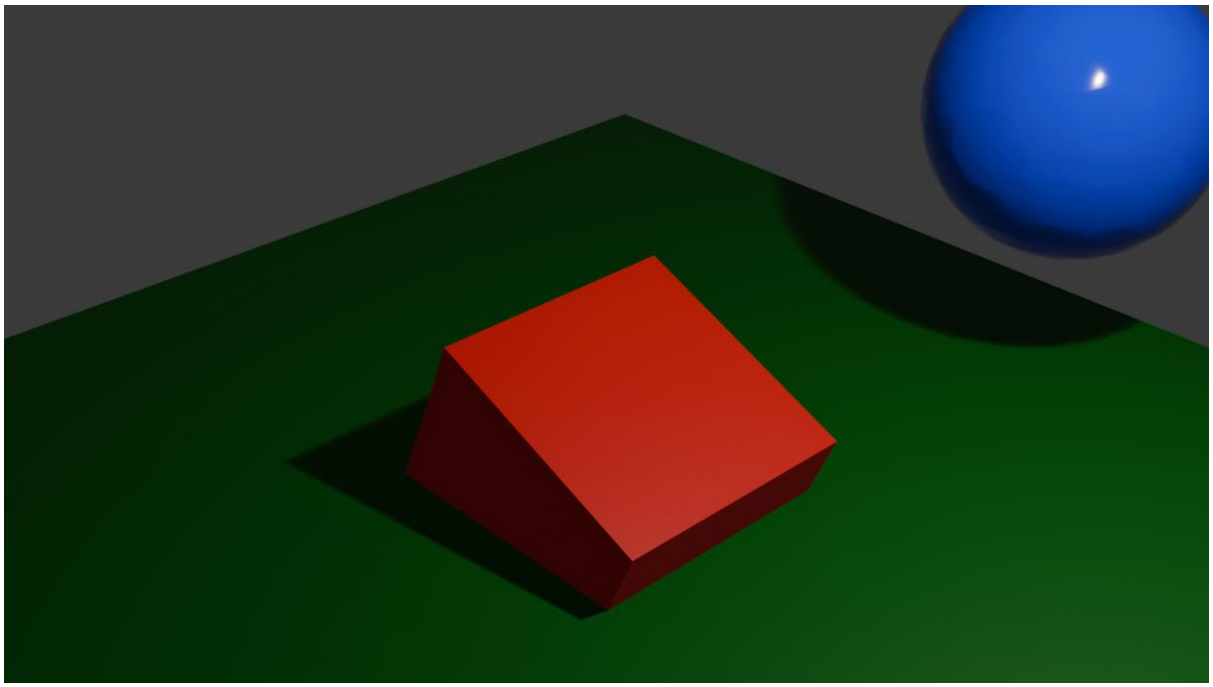
Whitted-style render with 1 reflection/refraction depth (whitted_style_1_ref.ppm)



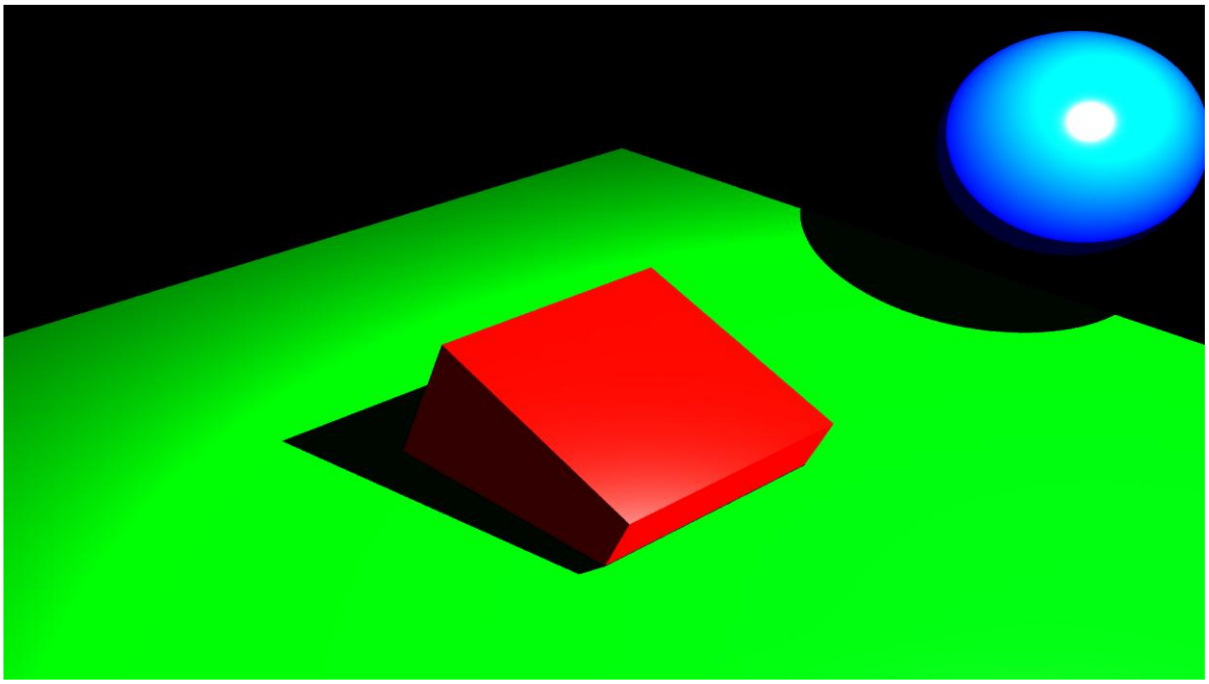
Whitted-style render with 2 reflection/refraction depth (whitted_style_2_ref.ppm)

Anti-aliasing

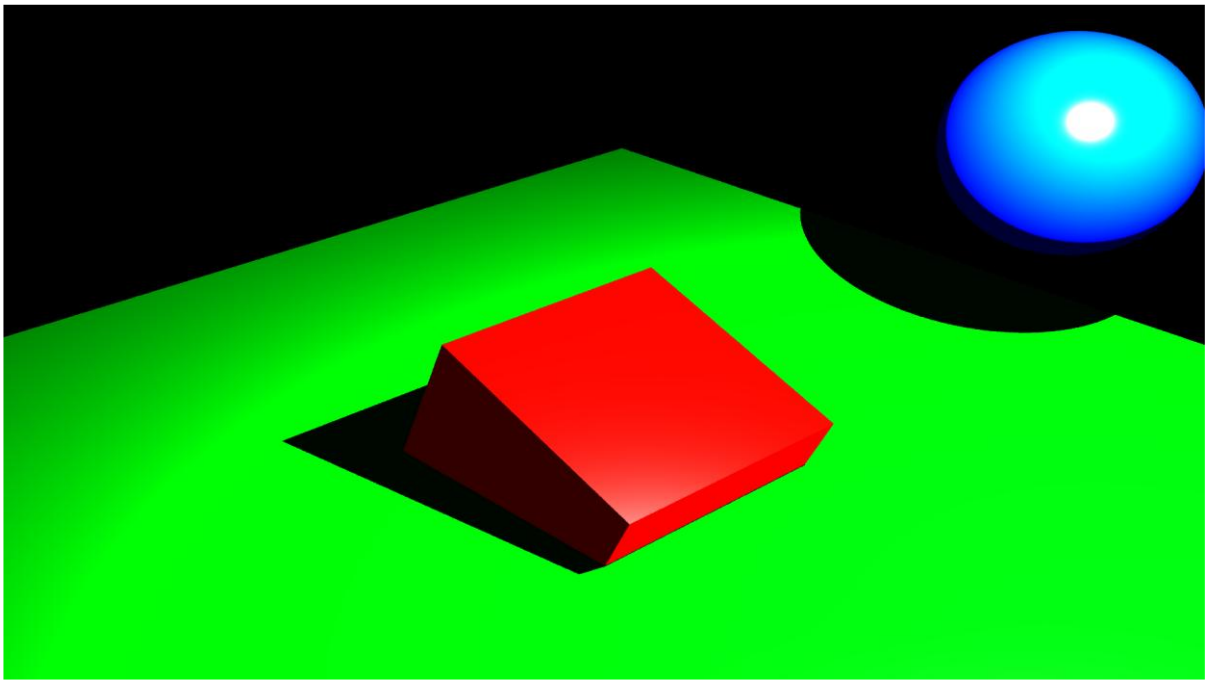
Implemented by taking multiple rays per pixel, each directed at a slightly jittered position within it (not centre). The colour across each sample is averaged.



Blender render (test_personal_2.blend)



Whitted-style render with 4 AA samples (antialiasing_4.ppm)

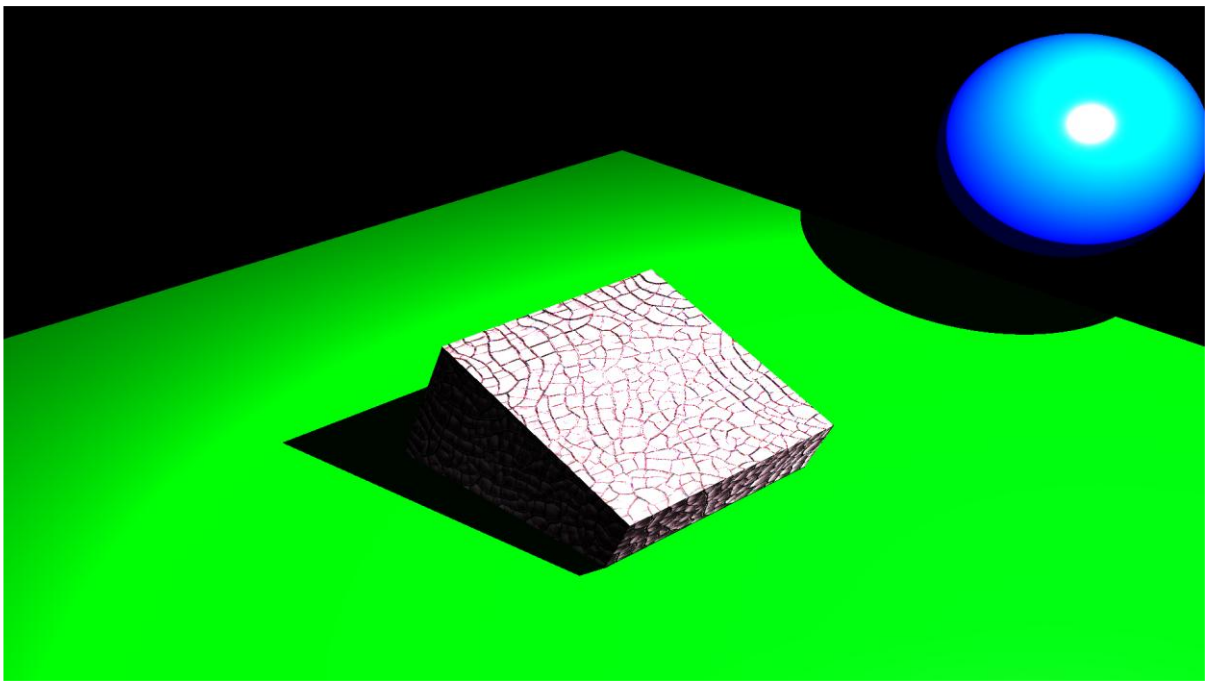


Whitted-style render with 8 AA samples (antialiasing_8.ppm)

Textures

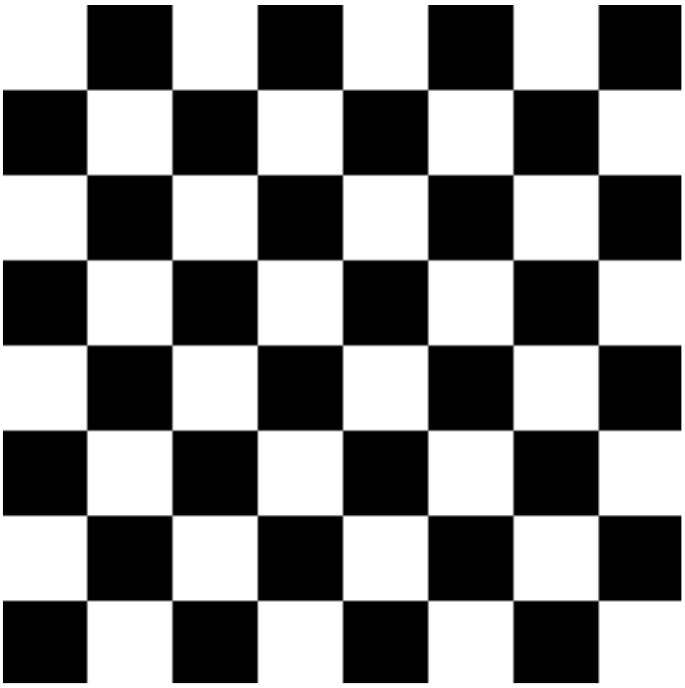
Texture file name is read from the Blender scene. If texture mapping is enabled on the raytracer (by command line arguments), the texture will be read from the “Textures” folder and the hit point (u, v) will be the point on the texture where colour is sampled and mapped onto the intersection

Texture mapped (tex2.ppm)

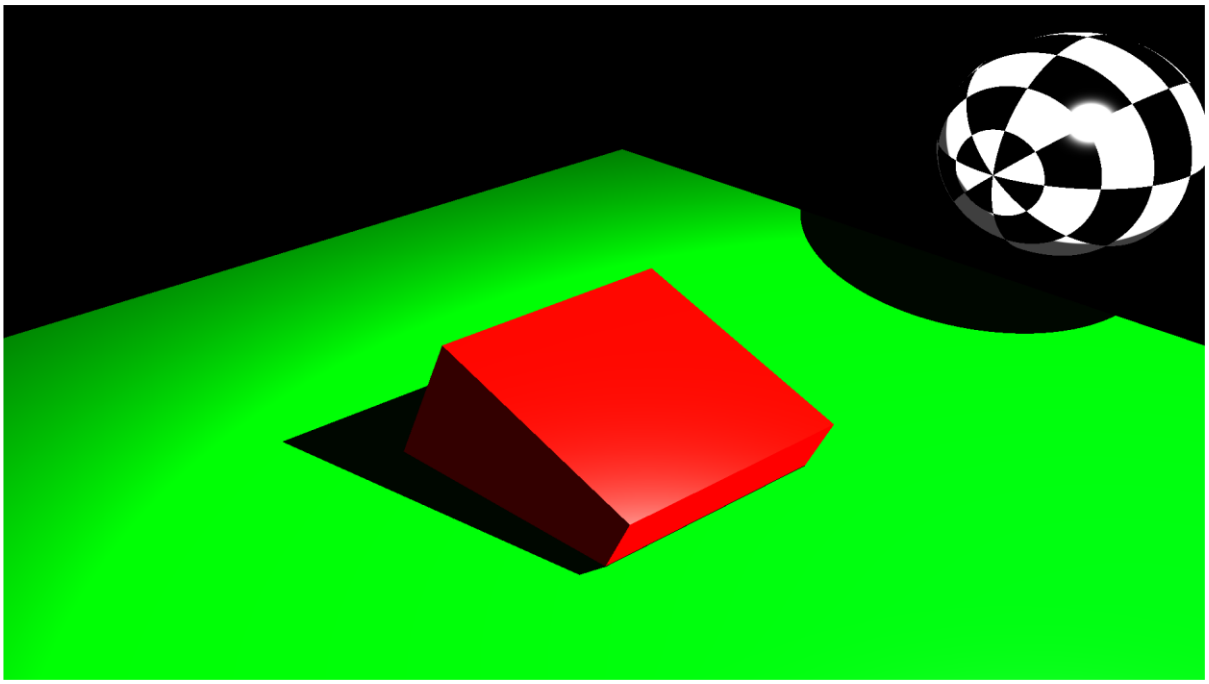


Texture mapped to cube (tex_map1.ppm)

(texture_mapping2.blend)



Texture mapped (checkerboard.ppm)



Texture mapped to sphere (tex_map2.ppm)

(texture_mapping2.blend)

System Integration

Functionality has been divided into modules and classes to support extensibility and modularity. Command line arguments have been added to allow activation/deactivation of features such as soft shadows, glossy reflections, anti-aliasing, BVH acceleration, reflections/refractions, depth of field and motion blur.

```
struct Config {
    bool softShadows = false;
    int SSsamples = 4;
    bool glossyReflect = false;
    int GRsamples = 16;
    float lightRadius = 0.25f;

    bool antiAliasing = false;
    int AAsamples = 4;

    bool textures = false;

    bool bvh = true;

    bool reflections = false;
    int reflectRefractDepth = 0;

    bool dof = false;
    int dofSamples = 16;
    bool motionBlur = false;
    int mbSamples = 16;
    string outputFile = "output.ppm";

    static Config parseArgs(int argc, char* argv[]);
};

#endif
```

Configuration options (config.h)

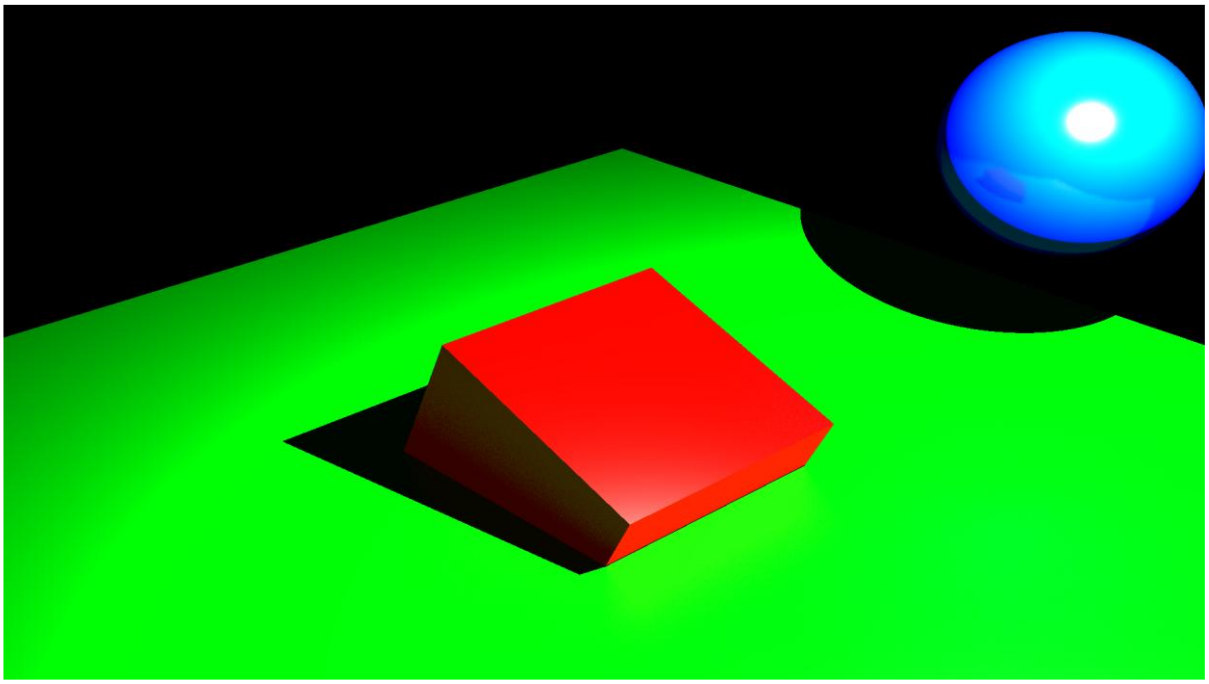
```
class Raytracer{
    public:

        //Main functions
        static vector<float> reflectRefract(Config, BVHNode*, Ray&, const Camera&, vector<PointLight>&
        static vector<float> blinnPhong(Config, BVHNode*, HitStructure&, const Camera&, vector<PointLi
        static float computeHardShadows(const vector<float>&, const float, const vector<float>&, const
        static bool unacceleratedIntersection(Ray&, vector<Plane>&, vector<Sphere>&, vector<Cube>&, Co

        //Distributed RT
        static float computeSoftShadows(const vector<float>&, const float, const vector<float>&, const

        //Helper functions
        //Vectors
        static vector<float> normalise(vector<float>);
        static float dotProd(vector<float>, vector<float>);
        static vector<float> crossProd(vector<float>, vector<float>);
        static vector<float> add_vec(vector<float>, vector<float>);
        static vector<float> sub_vec(vector<float>, vector<float>);
        static vector<float> mul_vec(vector<float>, float);

        //Random sampling
        static vector<float> rndUnitSphere();
        static vector<float> rndConeDirection(const vector<float>&, float);

        //Other
        static float lerp(float, float, float);
};
#endif
```

Raytracer module separation (raytracer.h)

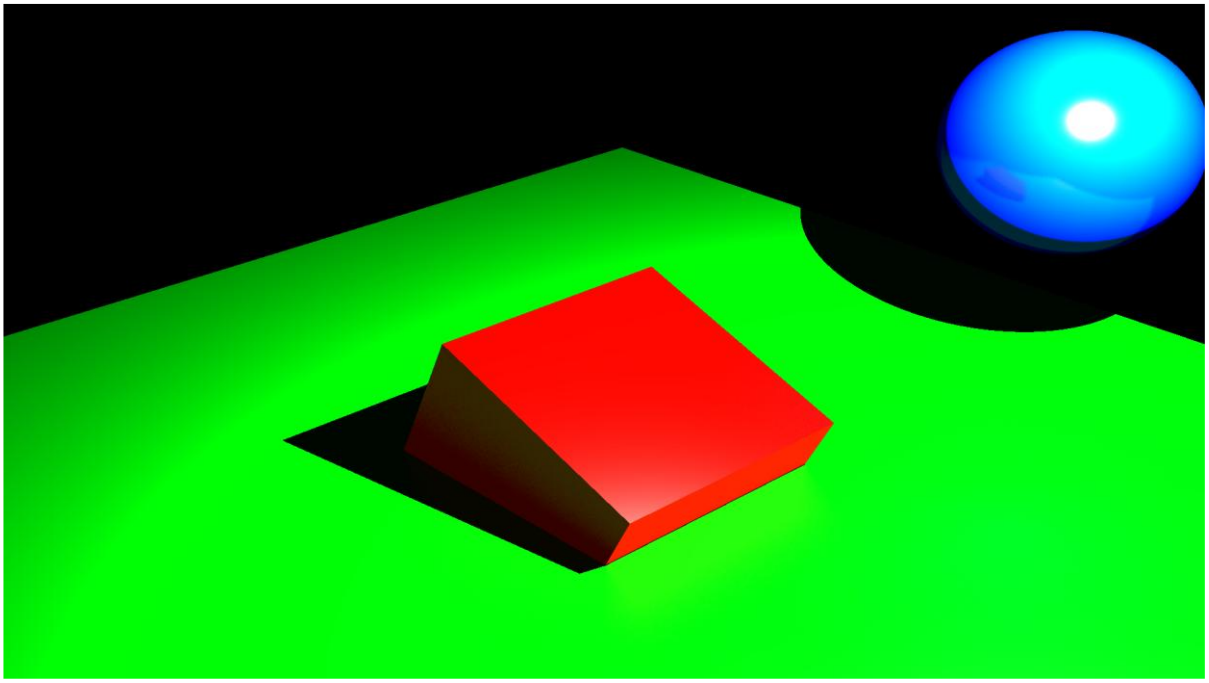
Distributed raytracing

For glossy reflections, reflected rays are sampled over cone direction (instead of ideal reflection). The colour component from that depth of reflection is averaged over the samples.

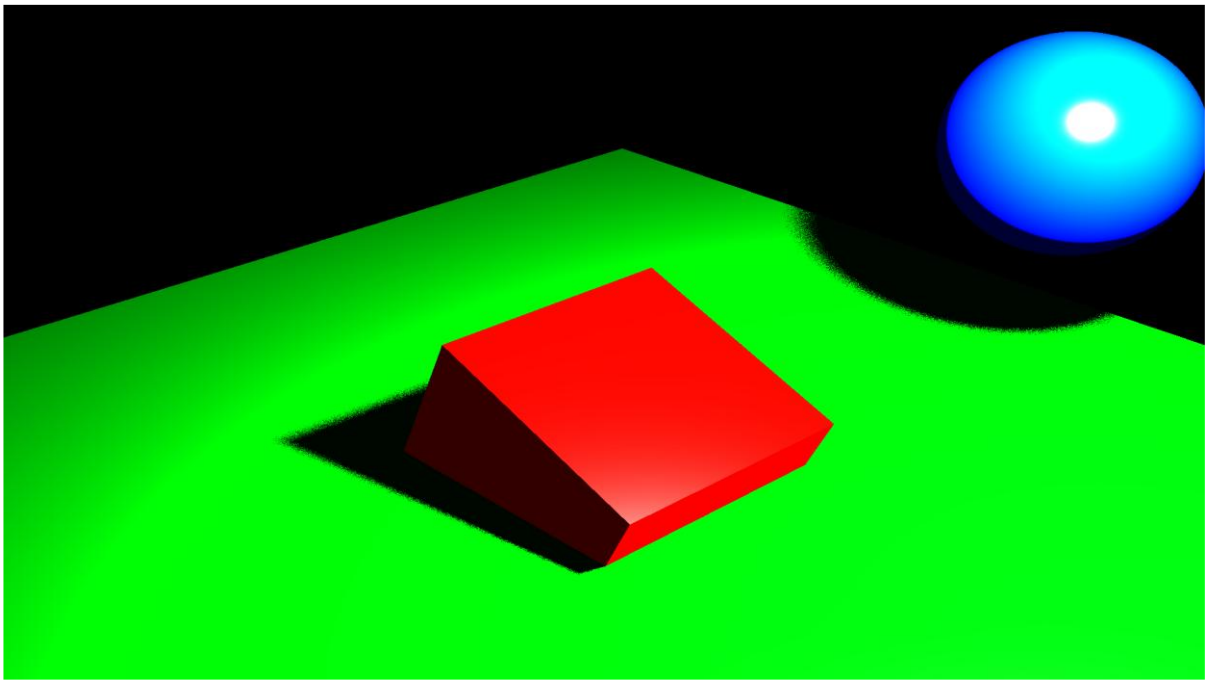
For soft shadows, shadow ray origins are jittered over the light's radius for each light, then their blocked/unblocked ratio is averaged over rays to give the softer effect.



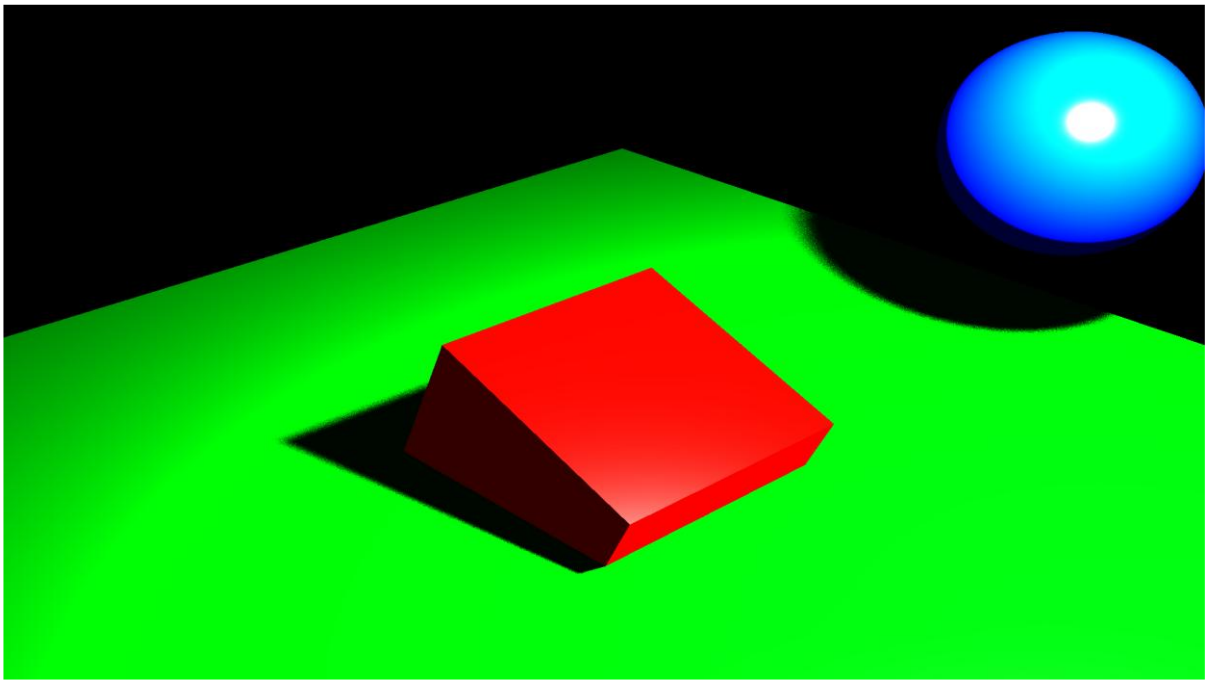
Whitted-style render with 4 glossy reflection samples (reflection depth = 1) (glossy_4.ppm)



Whitted-style render with 8 glossy reflection samples (reflection depth = 1) (glossy_8.ppm)



Soft shadows render with 4 shadow samples (softshadows_4.ppm)

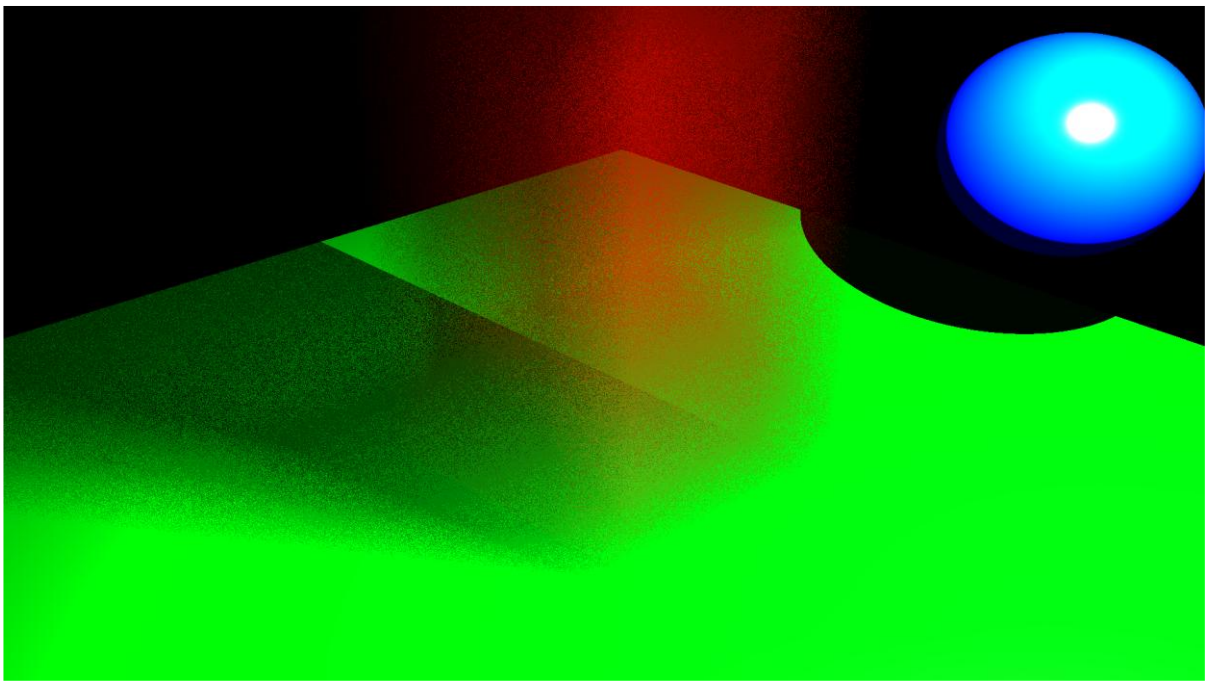


Soft shadows render with 16 shadow samples (softshadows_16.ppm)

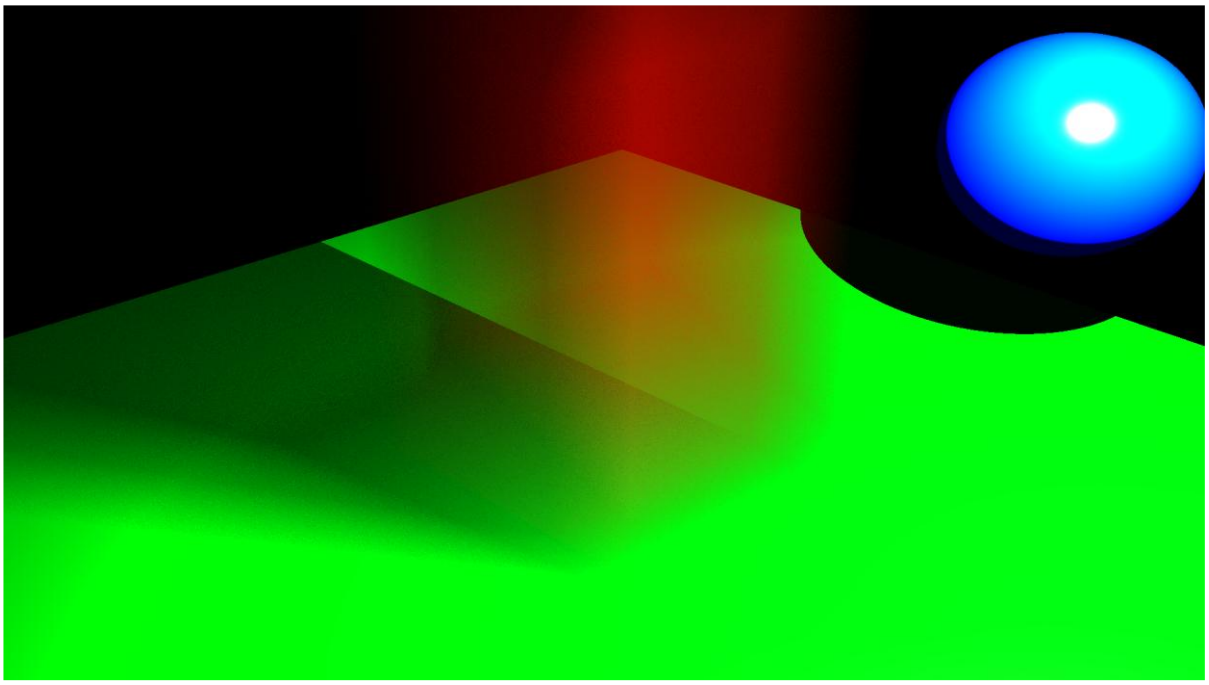
Lens effects

Motion blur uses the object's start and end positions and assumes a constant movement between those points between times 0 and 1. For each pixel, each sample ray is given a random time, and it returns the intersection of each object at that time. These contributions are averaged over to give the blur effect for a single pixel.

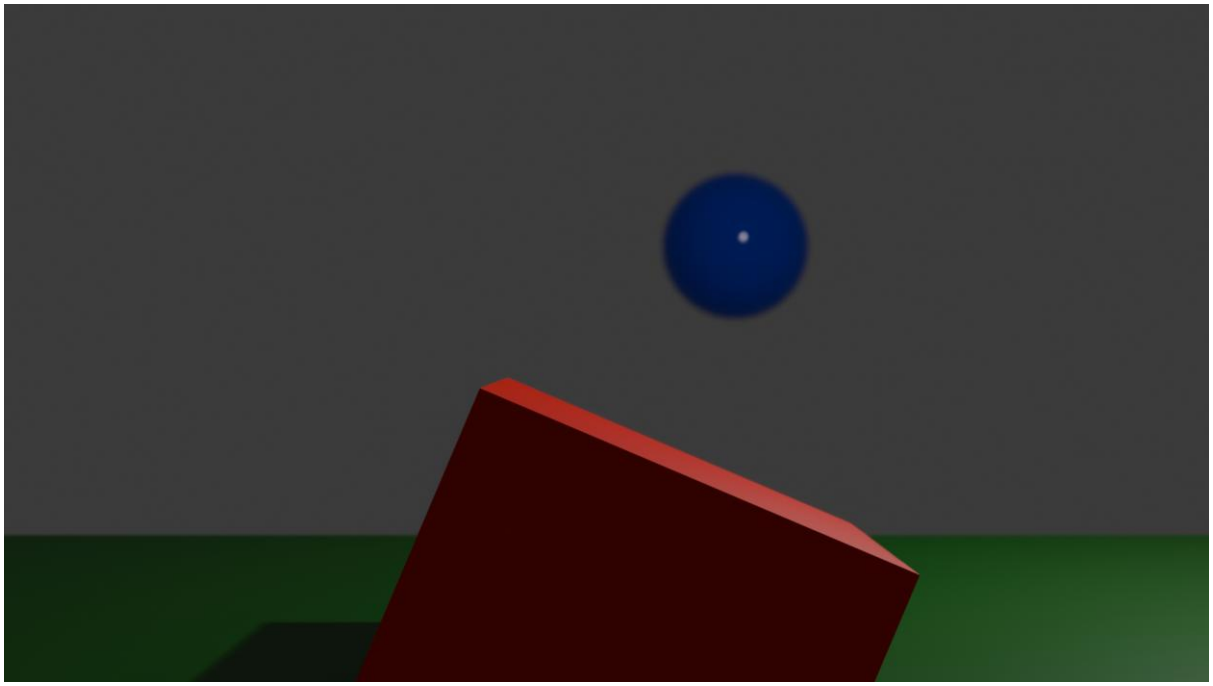
Depth of field adjusts ray direction by focal length and lens aperture. This causes more rays to be focussed on the focus point, so it looks clearer, and noisier further away.



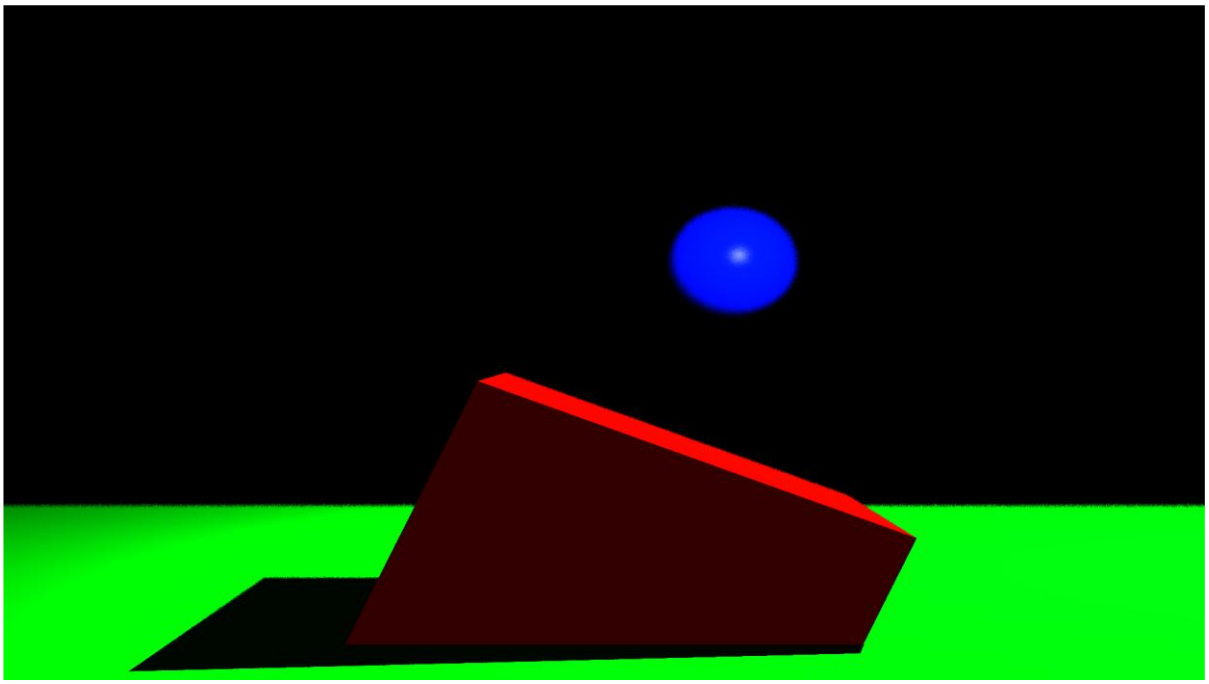
Render of cube moving between [0, 0, 0] and [0, 0, 5] with 4 motion blur samples (cube_mb.blend) (cube_mb.ppm)



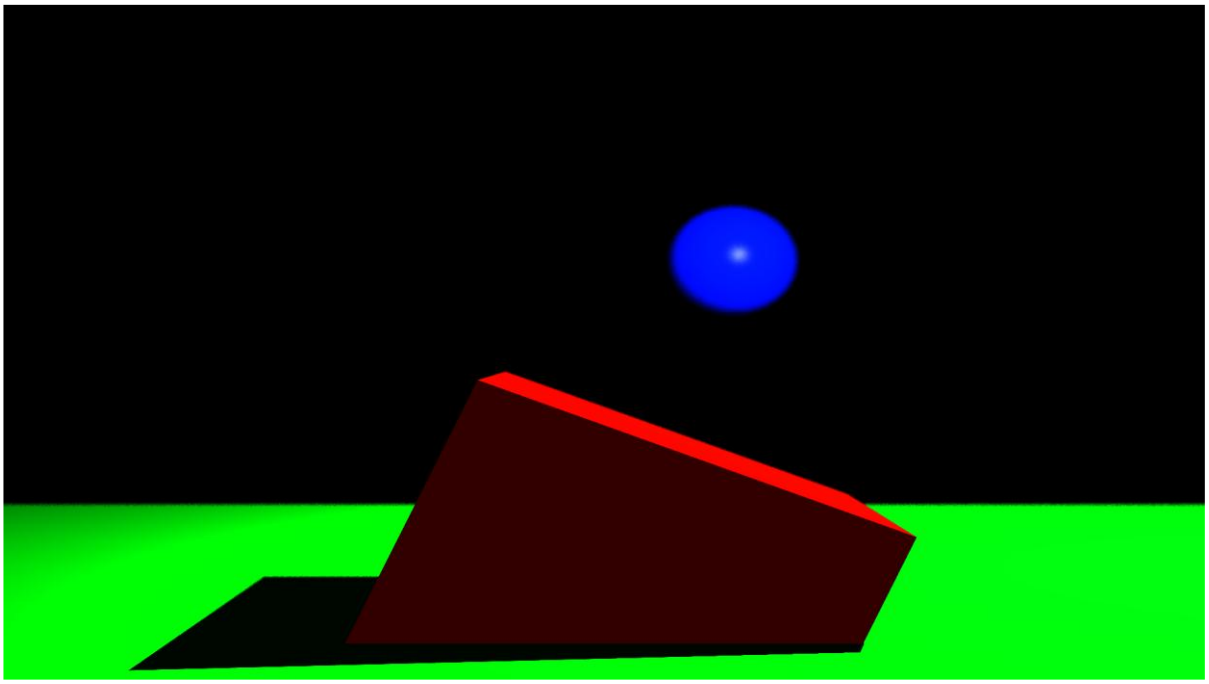
Render of cube moving between [0, 0, 0] and [0, 0, 5] with 16 motion blur samples (cube_mb.blend) (cube_mb_16.ppm)



Blender render of dof.blend



Render with depth of field, with 4 DoF samples (dof.blend) (dof_4.ppm)



Render with depth of field, with 16 DoF samples (dof.blend) (dof_16.ppm)

Timeliness Bonus

Module 1

The changes made to the export script were to facilitate further functionality, and the version submitted by the deadline was sufficient for the functionality at the time.

For camera data, slight adjustments were made to right and up vectors, due to image orientation being mirrored.

Pixel-to-ray conversion has stayed the same, only being affected by camera calculations.

Image read and write has only been changed to support user-decided file names, and to fully support P6 PPMs files.

While small bug fixes and improvements have been made, I believe this module met the functionality required at the time.

Module 2

Intersection checking for improvements in recognising transformed shapes, but the fundamental functionality is the same

The Shape hierarchy is consistent, with extra functions and properties being added to support new functionality.

The hit structure is also consistent, only including extra properties to support further functionality.

The acceleration hierarchy has been adjusted to account for moving objects (for motion blur) but with stationary object is the same.

Overall, I believe there has only been improvements to this module, with the submitted functionality meeting the requirements.

Module 3

Whitted-style raytracing has been improved by using hitpoint-to-light shadow checking instead of just Blinn-Phong and reflections/refractions and adjustments to material extraction have been made to help with lighting.

Anti-aliasing has stayed the same.

Texture mapping has been improved to ensure textures are extracted from Blender scene, but overall are the same.

I believe this module has only been improved, with the functionality submitted meeting the requirements, although not perfectly.