Comparison of ray tracing and environment mapping

02561 Computer Graphics 16F
02562 Rendering - Introduction 16F

Technical University of Denmark

Bjørn Hasager Vinther
19-05-1992
s152423

Peter Scheel Nielsen
14-11-1991
s152424

Contents

1 Introduction

1.1 Scope

The scope of this project is to compare the render technique ray tracing with the real-time technique environment mapping in regards to reflections. The ray tracing part has been developed in the optix framework given to us in the course and the real-time part has been developed in WebGL.

We will use the Cornell box containing a few mirror balls as a test scene and compare the render time and quality of both methods. Since most of our work has been put into the real-time part, the report will mainly focus on this part. The ray tracing result will be used as a reference to the correct result. We will also look into various methods to improve the result of the real-time part.

The graphics weekly assignments and project can be found here: `http://www.student.dtu.dk/~s152423/Graphics/`.

1.2 Work distribution

We have both done an equal amount of code in this project.

For the report Peter has written section 2.2, 3.1, 4.1, and 5.

Bjørn has written section 1, 2.1, 3.2, and 4.2.

2 Theory

2.1 Raytracing

Ray tracing works by tracing rays from the eye (camera) through an image plane and out in the scene in order to calculate the color of every pixel in the image plane. Whenever the ray intersects a reflective surface it will send a new ray in the reflected direction in order to determine the reflectance of the object. This ray could once again hit a new reflective surface and do the same thing for the new surface. Same method applies for refraction except it now uses the refracted direction. A trace depth can be set to control the number of times a ray is allowed to bounce, and thus the number of inter-reflections, see figure 1.

The number of rays per pixel and the trace depth determine the accuracy of this technique. Increasing the number of rays will give a more realistic result but the render time increases as well. Increasing the trace depth will allow more accurate reflections and refractions, however, the change will quickly become unnoticeable.

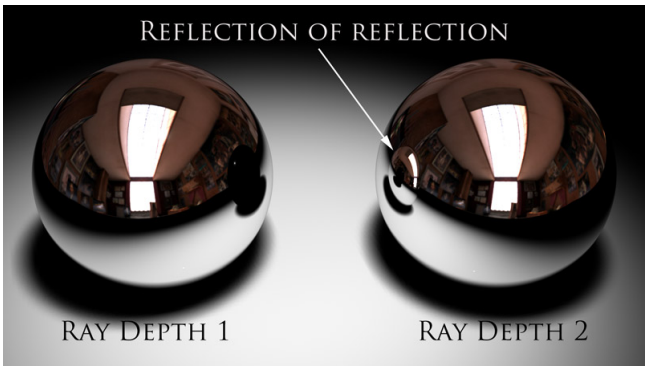


Figure 1: Comparison of render results with different trace depth. `http://www.ageofarmour.com/3d/tutorials/AoA_metalized_glass_shader_help.html`

Figure 2: Result of using ray tracing. Notice the highly detailed reflections. `https://en.wikipedia.org/wiki/Ray_tracing_(graphics)`

Ray tracing is a simple but expensive technique. The main benefit of this technique is that it can render highly photo-realistic results especially in regards to shadows and reflections. This, of cause, comes at the cost of render time, it easily becomes a very slow technique. The technique is ideal for rendering still images and for modelling highly reflective surfaces. Because it is a slow technique, it needs to be optimised for interactive programs where a fast render time is essential.

2.2 Environment mapping

A simple but faster alternative to ray tracing is using environment mapping.

Environment mapping uses a cube map to render the reflections of the surroundings. The cube map consists of six images taken from the center of the object in all directions (up, down, left, right, forward, backward).

For every visible point on the reflective surface the direction of the reflected ray can be calculated from the normal and the direction towards the camera. It then uses the reflected direction to look up in the cube map. This requires way less computation since the program can find the color from a look up in the cube map rather than having to trace a ray through the whole scene. Each triangle can now also be rendered independent of each other, which is the core idea behind pipeline rendering.

Creating a dynamic environment map is the same process as used in the exercise of week 10, but instead of using static images, the cube map is generated in each frame. Rendering the environment map is done by first moving the camera into the center of the reflective object that should use the environment map. The camera then renders the scene except the object itself in each of the six directions. The camera should have a 90 degrees field of view. The

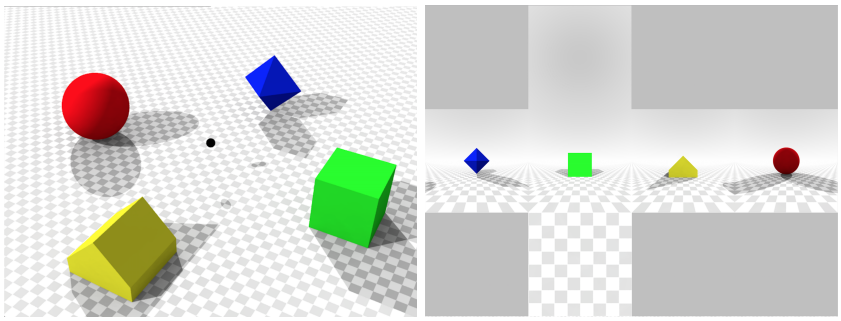


Figure 3: Illustration of a scene (left) and the corresponding environment map (right), `https://en.wikipedia.org/wiki/Cube_mapping`

positioning of the camera can be seen in figure 3, where the black dot in the center of the objects is illustrating the position.

Then the cube map is created and can be applied to the reflective object. This procedure has to be done for each reflective object.

There exists two issues with this approach.

The first is that it does not render inter-reflections. One simple but computational expensive solution would be to first render the environment, then apply it to the mirror balls and then repeat the process for each number of inter-reflections. The total number of times the scene has to be rendered can be calculated as:

$$R = N_{reflector} * i + 2$$

where R is the total number of times the scene is rendered, $N_{reflector}$ is the number of reflectors, and i is the number of inter-reflections. Besides for the cube maps the scene also has to be rendered once for the shadow map, and once for the final result, thus 2 is added.

An even better solution would be to use the environment map from the previous frame when generating the new environment map in the current frame. This will approximate a pretty accurate inter-reflection and only require the program to generate one environment map per reflective object in the scene.

$$R = N_{reflector} + 2$$

The second issue is that the environment map is generated from the center of the mirror balls but the reflection should ideally be from the surface point. This will result in incorrect reflections, especially if the distance from the center to the surface point is considerable compared to the distance of the surroundings.

One solution to this is called parallax correction. This method stores the distance to the surroundings in a depth map. For a reflected direction R it looks up the distance $|r|$ to the environment an finds a new point r. It assumes that the surface at r is perpendicular to R and calculate the distance d_p of the vector going from the surface point x to the new point p as $d_p = |r| - R \cdot x$, as shown

in figure 4. Using a linear function one can find the point p as $p = x + R * d_p$. Using the direction towards towards p when looking up in the environment map will give a more accurate reflection.

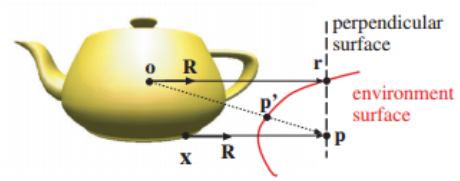


Figure 4: Parallax correction.

If the distance from the center of the reflector to the surface point is significant compared to the distance to the environment, parallax mapping will have adjust the look-up direction a lot. As the distance to the surface point becomes less significant the adjustment of parallax correction will become less noticeable.

3 Implementation

We have implemented the real-time part of the project in three different parts.

Part one will use a naive method to render all inter-reflections in each frame. In part two we will reuse the cube map from the previous frame when rendering inter-reflections. Finally, in part three we will use parallax correction to obtain more accurate reflections.

For the ray tracing part we have modified the Optix framework from the weekly exercises.

3.1 Ray tracing in optix

Our Optix implementation from the weekly hand ins almost had all the implemention we needed. In order to load the Cornell box we needed functions to perform intersection tests on tri-meshes and implement the area light, see listing 1 and 2.

For the intersect function we find the three vertices of the triangle with the given index. We then do a simple triangle intersection.

We implemented a simplified area light where the light position and normal was set to the mean of all vertex positions and normals. We then calculate the form factor used to determine the light contribution of the light in a given point.

Listing 1: Snippets from TriMesh.cpp

```
function render(){
    bool TriMesh::intersect(const Ray& r, HitInfo& hit, unsigned
        int prim_idx) const
{
  uint3 face = geometry.face(prim_idx);
  // positions of the vertices
  float3 v0 = geometry.vertex(face.x);
  float3 v1 = geometry.vertex(face.y);
  float3 v2 = geometry.vertex(face.z);

  float3 n;
  float t, w, v;

  // check if ray intersects triangle
  if (::intersect_triangle(r, v0, v1, v2, n, t, v, w)) {
    hit.has_hit = true;
    hit.dist = t;
    hit.geometric_normal = n;
    hit.material = &materials[mat_idx[prim_idx]];

    // calculate interpolated normal
    if (has_normals()) {
      float3 n0 = normals.vertex(face.x);
      float3 n1 = normals.vertex(face.y);
      float3 n2 = normals.vertex(face.z);
      n = normalize(n0 * (1 - v - w) + n1 * v + n2 * w);
    }
    hit.shading_normal = n;
```

```
28
29      // calculate interpolated texcoord
30      if (hit.material->has_texture) {
31        float3 tex0 = texcoords.vertex(face.x);
32        float3 tex1 = texcoords.vertex(face.y);
33        float3 tex2 = texcoords.vertex(face.z);
34        hit.texcoord = tex0 * (1 - v - w) + tex1 * v + tex2 * w;
35      }
36
37      return true;
38    }
39    return false;
40  }
```

Listing 2: Snippets from AreaLight.cpp

```
1   bool AreaLight::sample(const float3& pos, float3& dir, float3& L
        ) const
2
3   {
4     // shadows disabled
5     if (!shadows) {
6       return true;
7     }
8
9     // Get geometry info
10    const IndexedFaceSet& geometry = mesh->geometry;
11    const IndexedFaceSet& normals = mesh->normals;
12    const float no_of_faces = static_cast<float>(geometry.no_faces
          ());
13    const float no_of_vertices = static_cast<float>(geometry.
          no_vertices());
14
15    // set light position as mean of all vertex positions
16    // set light normal as mean of all vertex normals
17    float3 light_pos = make_float3(0.0f);
18    float3 light_n = make_float3(0.0f);
19
20    for (int i = 0; i < no_of_vertices; i++) {
21      light_pos += geometry.vertex(i);
22      light_n += normals.vertex(i);
23    }
24    light_pos /= no_of_vertices;
25    light_n = normalize(light_n);
26
27    // calculate total emission
28    /*for (int i = 0; i < no_of_faces; i++) {
29      light_pos += geometry.vertex(i);
30      light_n += normals.vertex(i);
31    }*/
32
33    // calculate distance and direction
34    float t = length(light_pos - pos);
35    dir = normalize(light_pos - pos);
36
37    // calculate form factor
38    float ff = max(dot(light_n, -dir), 0.0f) / (t * t);
```

```
39    // total emission (assumes emission is constant across surface
          )
40    float3 emit = get_emission(0) * mesh->surface_area;
41    L = emit * ff;
42
43    Ray r = Ray(pos, dir, 0, 10e-4, t);
44    HitInfo hit;
45    return tracer->trace_to_any(r, hit);
46  }
```

3.2 Environment mapping in WebGL

A single render pass in our program has the following structure:

1. apply animations
2. create the shadow map
3. generate cube maps
4. render scene
5. track render time

Successfully implementing environment mapping proved to be a big challenge. The following sections explains some of the challenges we encountered when implementing it.

3.2.1 Loading the models

For the environment map implementation, we had issues when trying to load the models from the CornellBox.obj.

We had to fix the Objparser to allow for the materials to load. We discovered that the issue was that the .mtl file was not loaded until the model was already created, resulting all of the faces to have the same default color. We fixed this by first loading the material file and parsing it as a string when loading the model.

When the MTLDoc was initialised, another issue caused the reading of the material name to be an empty string, still causing every face to have the default color. This bug was fixed in the parseUsemtl() and parseNewmtl(), to return the actual name of the material so the materials was correctly parsed into the MTLDoc.

Listing 3: Snippets from OBJParser.js

```
1  OBJDoc.prototype.parse = function (fileString, scale, reverse,
       mtllibString) {
2      var lines = fileString.split('\n');  // Break up into lines
           and store them as array
3      lines.push(null); // Append null
4      var index = 0;    // Initialize index of line
```

```
5
6       var mtl = new MTLDoc();   // Create MTL instance
7       this.mtls.push(mtl);
8       onReadMTLFile(mtllibString, mtl);
9
10      var currentObject = new OBJObject("");
11      this.objects.push(currentObject);
12      var currentMaterialName = "";
13
14      // Parse line by line
15      var line;         // A string in the line to be parsed
16      var sp = new StringParser();  // Create StringParser
17      while ((line = lines[index++]) != null) {....}
18  }
19
20  OBJDoc.prototype.parseNewmtl = function (sp) {
21      var newStr = sp.str.substring(7).replace('\r','');
22      return newStr;
23  }
24
25  OBJDoc.prototype.parseUsemtl = function (sp) {
26      var newStr = sp.str.substring(7).replace('\r','');
27      return newStr;
28  }
```

3.2.2 Rendering the Cornell box

We chose to render the Cornell box using a diffuse shader using Phong shading. We positioned the camera at [278, 273, -800] facing in the z-direction using 37.5 degrees as field of view.

The area light was rendered as a simple point light since this was not a focus of this report. The shadows were rendered using shadow mapping as in the exercise of week 8.

3.2.3 Rendering the mirror balls

We created a shader for the mirror balls which just made a look-up in the cube map. Each mirror ball was constructed from a tetra-hexagon with a subdivision level of five.

The mirror balls have the following properties:

id	position [x,y,z]	radius	animated
1	[445, 73, 320]	72	no
2	[90, 100, 110]	72	no
3	[190, 113, 410]	110	no
4	[500, 80, 210]	36	yes

Generating the cube maps was the main challenge for the mirror balls. To

create the reflection, the technique explained in the theory section was applied. The camera was moved into the center of the reflector and the scene was rendered in the six general directions.

In part one we used a ping-pong technique to switch between two cube maps (old and new) so that only two cube maps were needed. The code example below are taken from part one. In part two and three the cube maps are only generated once per frame.

Listing 4: Snippets from Part1.js

```
function generateCubeMaps() {
    // clear all cube maps initially
    for (var i = 0; i < data.mirrorBalls.length; i++) {
        resetCubemap(data.mirrorBalls[i]);
    }

    for (var j = 0; j < data.interreflections; j++) {
        // first we render all cube maps
        for (var i = 0; i < data.mirrorBalls.length; i++) {
            generateCubemap(data.mirrorBalls[i], renderScene);
        }
        // once all maps are rendered we update them all
        for (var i = 0; i < data.mirrorBalls.length; i++) {
            updateCubeMap(data.mirrorBalls[i]);
        }
    }
}
```

To render the cube map we used a frame buffer object to render to a texture. The code can be seen below. The method sets the view-projection matrix and calls renderScene() with the id of the mirror ball.

Listing 5: Snippets from MirrorBall.js

```
// generates a cubemap for a given mirror ball
function generateCubemap(mirrorBall, renderFunction) {
    // create projection matrix
    var fov = 90;
    var aspect = 1;
    var P = perspective(fov, aspect, 1, 1000);

    // used for view matrix
    var eye = mirrorBall.position;

    // use the mirror ball framebuffer
    gl.bindFramebuffer(gl.FRAMEBUFFER, mirrorBall.framebuffer);
    gl.viewport(0, 0, size, size);

    // render positive x
    var at = vec3(eye[0] + 1, eye[1], eye[2]);
    var up = vec3(0, -1, 0);
    var V = lookAt(eye, at, up);
    gl.framebufferTexture2D(gl.FRAMEBUFFER,gl.COLOR_ATTACHMENT0,
         gl.TEXTURE_CUBE_MAP_POSITIVE_X, mirrorBall.writing, 0);
```

```
    renderFunction(V, P, mirrorBall.id, 1);
    // update V and render negative x
    ...
    // update V and render positive z
    ...
    // update V and render negative z
    ...
    // update V and render positive y
    ...
    // update V and render negative y
    ...

    // unbind framebuffer
    gl.bindFramebuffer(gl.FRAMEBUFFER, null);
    gl.viewport(0, 0, canvas.width, canvas.height);
}
}
```

Listing 6: Snippets from Part1.js

```
// render the whole scene except the MirrorBall with id <
    ignoreId> using the view and projection matrix
function renderScene(V, P, ignoreId) {
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

    renderCornellBox(V, P);

    // render the mirror balls
    switchProgram("MirrorBall");
    for (var i = 0; i < data.numOfBalls; i++) {
        if (ignoreId != i)
            renderMirrorBall(data.mirrorBalls[i], V, P);
    }
}
```

3.2.4 Implementing parallax correction

In order to implement parallax correction we first needed to create a depth map. Since we already had to render the scene when generating the cube map, we decided to calculate the depth in world space and store it in the alpha component of the texture. This, however, gave us some problems. The alpha component was clamped between zero and one.

Because we were runnning out of time we chose a less desirable quick-fix solution and just divided the depth with a fixed number to ensure it was less than one. When we used the depth for parallax correction we would then multiply with the same number again to transform it back to world coordinates.

We modified the shader so it could be used across all three parts of the project by introducing two new uniforms in the render pass for the spheres as well as the Cornell Box.

Listing 7: Snippets from fragment.txt (Lambertian)

```
...

uniform float storeDepth; // whether or not to store the depth
    in the alpha component

...

void main()
{
    // stores the depth in the alpha value
    float alpha = storeDepth > 0.5 ? length(fPosition) / 200.0 :
         1.0;

    ...

    gl_FragColor = vec4(visibility * Id + ambient, alpha);
}
```

Listing 8: Snippets from fragment.txt (MirrorBall)

```
...

uniform float useParallax; // whether or not to use parallax
    correction
uniform float storeDepth; // whether or not to store the depth
    in the alpha component

...

// parallax correct a reflected direction
vec3 correct(vec3 r) {
    // distance in given direction seen from the center of the
        sphere
    float rl = textureCube(cubemap, mat3(inv) * r).a * 200.0;
    // vector from center to surface point in view space
    vec3 x = fPosition - center;
    float dp = rl - dot(x, r);
    vec3 p = x + r * dp;
    return normalize(p);
}

void
main()
{
    // stores the depth in the alpha value
    float alpha = storeDepth > 0.5 ? length(fPosition) / 200.0 :
         1.0;

    // find reflected direction
    vec3 incident = normalize(fPosition);
    vec3 n = normalize(fNormal);
    vec3 reflect = reflect(incident, n); // in view space

    if (useParallax > 0.5) {
        reflect = correct(reflect);
    }

```

```
34      reflect = mat3(inv) * reflect; // now in world space
35
36      // find value in cubemap
37      vec3 color = textureCube(cubemap, reflect).rgb;
38      gl_FragColor = vec4(color, alpha);
39  }
```

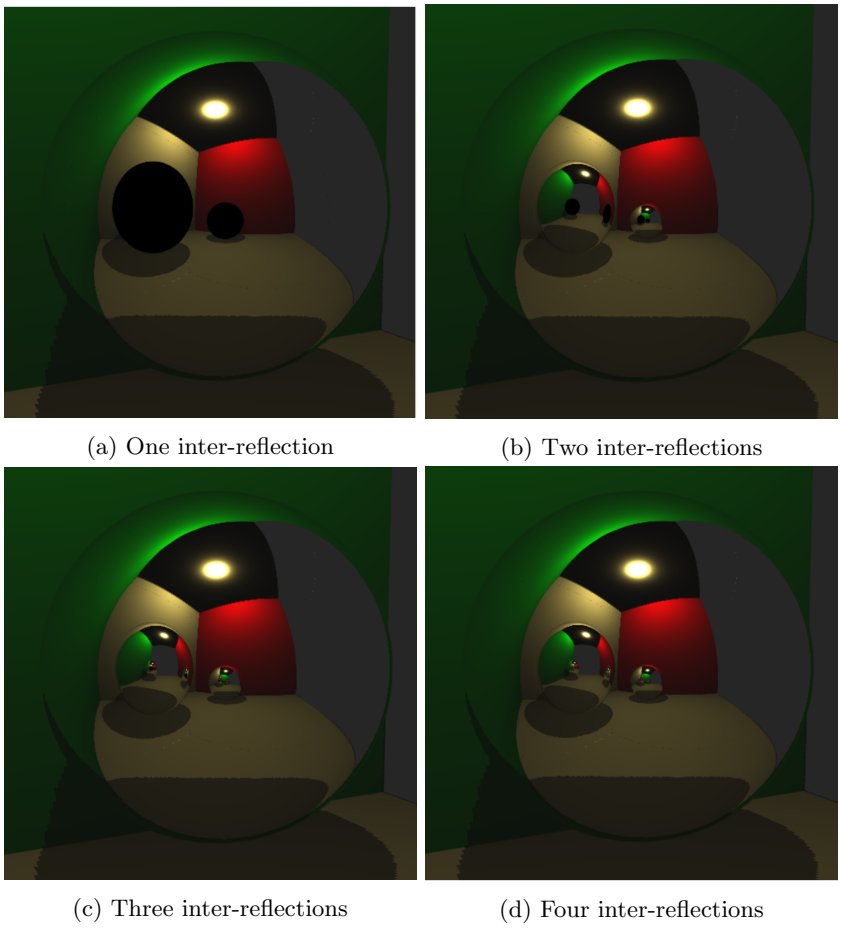
4 Comparison

In this section we will compare both the quality and performance of the different rendering techniques implemented in the previous section.

4.1 Quality

We will compare the results of environment mapping and use the ray tracing result as a reference.

For part one of the environment map implementation, we use a fixed amount of inter-reflections, and reset the environment map each time. Figure 5 shows the result of different number of inter-reflections. After three inter-reflections the difference becomes unnoticeable.



(a) One inter-reflection

(b) Two inter-reflections

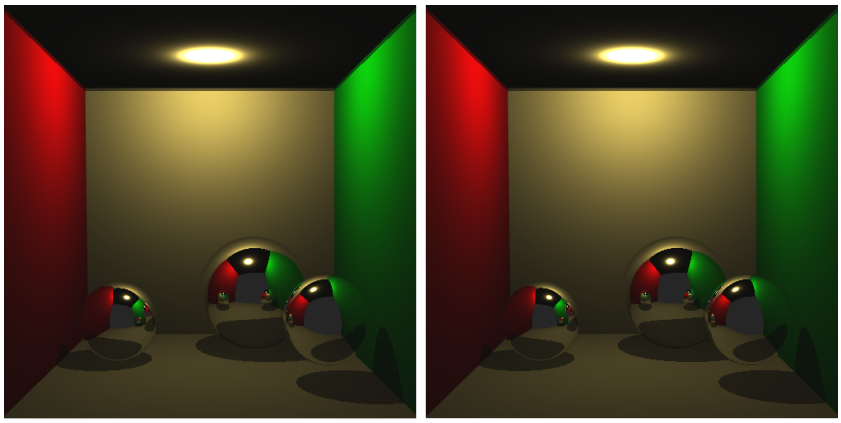
(c) Three inter-reflections

(d) Four inter-reflections

Figure 5: Comparison of inter-reflections

In theory, part two also has this problem, but since we do not reset the environment map each frame but instead use the cube map from the last frame for the inter-reflections, the problem will only occur the first couple of frames, making it impossible to spot unless the program is running at an extremely low frame rate.

Figure 6 shows that there is no visible difference between the reflections of part one and two.

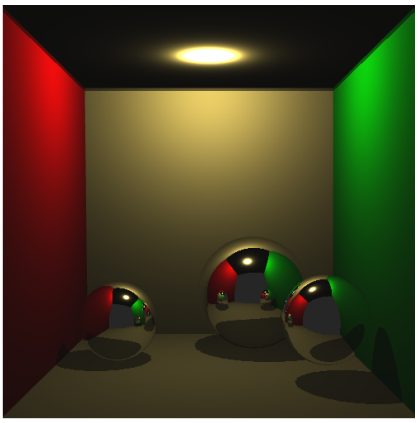


(a) Part one with four inter-reflections. (b) Part two

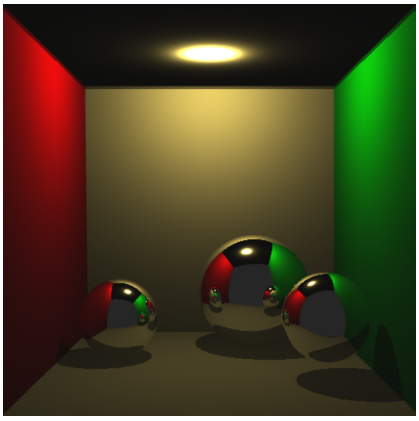
Figure 6: Comparison of the render result of part one and two.

For the final part of the project, we introduced parallax correction when looking up in the environment map. The most clear effect is when the spheres are reflecting shadows, where the shadows in the reflections simply does not look right before implementing parallax correction.

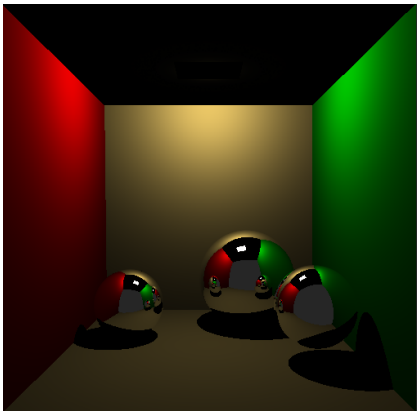
In figure 7 one can see that the final image looks very close to the ray trace result, and the reflected shadows look much more realistic after applying parallax correction.



(a) Standard environment map



(b) With parallax correction



(c) Ray trace

Figure 7: Comparison of using parallax correction or not.

The result, however, introduced some artifacts in some of the inter-reflection as shown in figure 8. This is due to an error in the implementation and not the technique itself. We think the error is related to the issue of storing the depth in the alpha component of the cube map, but we do not know for sure.

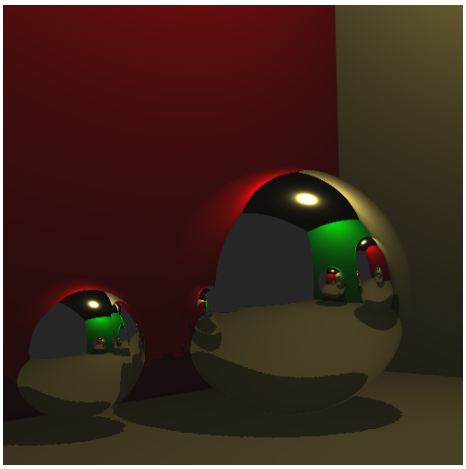


Figure 8: Artifacts in the inter-reflections in part three.

4.2 Performance

We will compare the render time of ray tracing with the methods in environment mapping. Since there is no easy and accurate way of measuring the performance on the GPU, we will instead measure the frames per second (FPS) by disabling vertical synchronization in the browser. All tests has been performed offline using the following specs:

CPU Intel(R) Core(TM) i5-4570S CPU @ 2.90GHz

GPU GeForce GTX 770

RAM 8 GB RAM

Table 1 clearly shows that the performance improved by reusing the cube maps of the previous frame. One thing that is unexpected is that the performance is even better than when having a inter-reflection of zero. We think the reason for this is because we clear the whole cube map at the beginning of each frame in part one and not in part two and three. This is of cause not needed when the inter-reflections are set to zero but we do not check for that.

Another interesting thing is that the render time of part two and three is almost identical. This means that the only real drawback of parallax correction is the added complexity of the code.

Even the naive implementation is multiple times faster than the ray tracing technique. Even though the ray tracing part has not been fully optimised we will still argue that it is too slow for real-time applications, since most devices uses around 60 FPS. We could maybe get it a few times faster but not 15 times which is the case with part three.

implementation	inter-reflections	FPS	MSPF
Part one	0	93	10.8
Part one	1	88	11.4
Part one	2	68	14.7
Part one	3	48	20.8
Part one	4	37	27.0
Part two	-	136	7.4
Part three	-	134	7.5
ray trace	-	10.7	93.0

Table 1: Render times in frames per second (FPS) and milliseconds per frame (MSPF).

5 Conclusion

Our results clearly shows that it is indeed possible to use environment mapping to create believable reflection, and we do believe that real-time reflections can be rendered sufficiently accurate for most real-time applications and therefore become a good technique for rendering reflections. Different real-time techniques and optimizations exists to give results that looks realistic to the user. Only minor artifacts may occur and therefore this rendering technique is great for applications where the result does not have to be photo realistic but instead have a good performance, e.g. computer games.

Ray tracing still produces the most accurate and realistic results but the method is currently too slow on consumer graphics hardware for real time applications. If the power of the consumer hardware becomes good enough, we think this method can become good due to its simplicity and high quality results.