Weeks 10 Report - Texture Mapping

Jakub Dutkowski, s121937
Alexander Birke, s124044

December 19, 2012

1 INTRODUCTION

The purpose of these exercises was to add texture mapping to the raytracing framework.

2 LOADING OF IMAGE FILES

The loading of textures is first started when the plane is loaded with the following line in RenderEngine.cpp:

```
1  scene.add_plane(make_float3(0.0f, 0.0f, 0.0f), make_float3(0.0f, 1.0f, 0.0f), "../models
       /default_scene.mtl", 1, 0.02f);
```

The add_plane function in In Scene.cpp is passed the path to the default scene material desription file. The add_plane function looks like this.

```
1  void Scene::add_plane(const float3& position, const float3& normal, const string&
       mtl_file, unsigned int idx, float tex_scale)
2  {
3    vector<ObjMaterial> m;
4    if(!mtl_file.empty())
5      mtl_load(mtl_file, m);
6    if(m.size() == 0)
7      m.push_back(ObjMaterial());
8    Plane* plane = new Plane(position, normal, idx < m.size() ? m[idx] : m.back(),
         tex_scale);
9    planes.push_back(plane);
10 }
```

First a vector is created to store all the materials in the file. The file is then passed with the call to mtl_load and all the materials in the file is put into the vector. A plane is then created and assigned a copy of the created material that corresponds to its index. An improvement to the parsing would be to only parse the mtf file once before the scene is created. As it is right now the mtf file is parsed each time a primitive is loaded. For each material the following three variables are defined: In ObjMaterial.h:

```
1     bool has_texture;
2     std::string tex_path, tex_name;
3     int tex_id;
```

They are set by the material's constructor. After the path to the the texture has been stored in the material by the call to load_files the init_textures in the render_engine is called afterwards. This in turn calls the function load_texture for all the planes, spheres and other meshes in the scene.

```
1  void Scene::load_texture(const ObjMaterial& mat, bool is_sphere)
2  {
3    if(mat.has_texture && textures.find(mat.tex_name) == textures.end())
4    {
5      Texture*& tex = textures[mat.tex_name];
6      tex = is_sphere ? new InvSphereMap : new Texture;
7      string path_and_name = mat.tex_path + mat.tex_name;
8      tex->load(path_and_name.c_str());
9    }
10 }
```

If the material has a texture the Texture::load function gets called:

```
1  void Texture::load(const char* filename)
2  {
3    SOIL_free_image_data(data);
4    data = SOIL_load_image(filename, &width, &height, &channels, SOIL_LOAD_AUTO);
5    if(!data)
6    {
7      cerr << "Error: Could not load texture image file." << endl;
8      return;
9    }
10   int img_size = width*height;
11   delete [] fdata;
12   fdata = new float4[img_size];
13   for(int i = 0; i < img_size; ++i)
14     fdata[i] = look_up(i);
15   tex_handle = SOIL_create_OGL_texture(data, width, height, channels, tex_handle,
         SOIL_FLAG_INVERT_Y);
16   tex_target = GL_TEXTURE_2D;
17 }
```

This function utilizes the Simple OpenGL Image Library that does all the more low level work like reading the compressed data in the texture into the data variable that can be accessed by the rest of the program. In order to use the data of the texture in the rest of the framework we also need to convert it into floating point format. This is done in the next section.

3 TEXTURE COLOUR'S USE IN RENDERING

In the load function an array called fdata is declared and filled with normalized floating point values of each pixels. This is done with the functions look_up and convert:

```
1  float4 Texture::look_up(unsigned int idx) const
2  {
3    idx *= channels;
4    switch(channels)
5    {
6    case 1:
7    {
8      float v = convert(data[idx]);
9      return make_float4(v, v, v, 1.0f);
10   }
11   case 2:
12     return make_float4(convert(data[idx]), convert(data[idx]), convert(data[idx]),
           convert(data[idx + 1]));
13   case 3:
14     return make_float4(convert(data[idx]), convert(data[idx + 1]), convert(data[idx +
           2]), 1.0f);
15   case 4:
16     return make_float4(convert(data[idx]), convert(data[idx + 1]), convert(data[idx +
           2]), convert(data[idx + 3]));
17   }
18   return make_float4(0.0f);
19 }
20
21 float Texture::convert(unsigned char c) const
22 {
23   return (c + 0.5f)/256.0f;
24 }
```

In look_up the index it is passed is first divided with the number of channels in the texture. This is done since the texture's data is stored in a flat array and the number of channels thus specifies how many bytes each pixel fills. The channel variable is also used in a switch case to determine how the floating data returned should be formatted. If it is a monocromatic texture then a greyscale color is returned. If the texture also contains an alpha channel this is also used in the calculation as well. If the texture has the three colour channels but no alpha channel then the alpha value returned is set to the default opague value of 1.0f. The last case is if the texture contains an alpha channel, then this is also converted and returned as well.

The convert function takes an char value in the range of 0 to 255 and remaps it to the range 0 to 1 in floating point format. What is weird is that the maximum value that can be mapped to is 255 + 0.5f / 256 = 0.998046875. A more correct way would be to add 1.0f insteaf of 0.5f.

4 COMPUTATION OF TEXTURE COORDINATES & TEXTURE LOOKUP

We add the following lines to the plane's intersect function:

```
1     if(material.has_texture)
2     {
3       float3 vector = hit.position - position;
4
5       // to get a basis change matrix we need an inverse matrix of [tangent, binormal,
            normal]
6       // as the vectors in the basis are orthogonal, we can use transposed matrix
            instead
7
8       float3 transposed1 = make_float3(onb.m_tangent.x, onb.m_binormal.x, onb.m_normal.x
            );
9       float3 transposed2 = make_float3(onb.m_tangent.y, onb.m_binormal.y, onb.m_normal.y
            );
10      float3 transposed3 = make_float3(onb.m_tangent.z, onb.m_binormal.z, onb.m_normal.z
            );
11
12      hit.texcoord = (vector.x * transposed1 + vector.y * transposed2 + vector.z *
            transposed3) * tex_scale;
13    }
14
15    return true;
```

What we do here is to remap the hit position in world coordinates into coordinates relative to the plane's center. This is done by first finding the vector that goes from the positon of the plane in world coordinates and to the hit also in world coordinates. We then use the ortonormal basis of the plane to transform this vector into the coordinate system of the plane. We then scale it with tex_scale in order to be able to control the tiling of the texture.

The next step is to implement basic point sampling. This is done via the function sample_nearest:

```
1  float4 Texture::sample_nearest(const float3& texcoord) const
2  {
3    if(!fdata)
4      return make_float4(0.0f);
5
6    int a = floor((texcoord.x - floor(texcoord.x)) * (width - 1) + .5);
7    int b = floor((1 - (texcoord.y - floor(texcoord.y))) * (height - 1) + .5);
8
9    return fdata[a + b * width];
10 }
```

The texel coordinates is found by first remapping the texcoord into a value between 0 and 1. This is done by substracting the floor'ed value. This normalized value is then multiplied with the width (or height respectively) minus one to make sure we do not go over the length of the pixel array. at the end we add 0.5 in order to take the pixel whose center is closest to the specified uv coordinates. At the end the texel coordinates is used to look into the flat array and return the correct pixel value.

5 THE ROLE OF THE PIXEL SUBDIVISION LEVEL

The next task was to render the image with different subdivisions where nearest neighbour sampling was used. The result can be seen on figure 5.1. As it can be seen the quality of the texture lookup goes up when more samples are introduced. However because the samples originate from the image plane and is not performed in the texture lookup the quality only goes up when the jittered rays has had some distance to scatter or said in another way when the rays have traveled farther. In order to improve the quality of the texture no matter how close it is we need to introduce more samples directly in the texture lookup.

6 BILINEAR SAMPLING

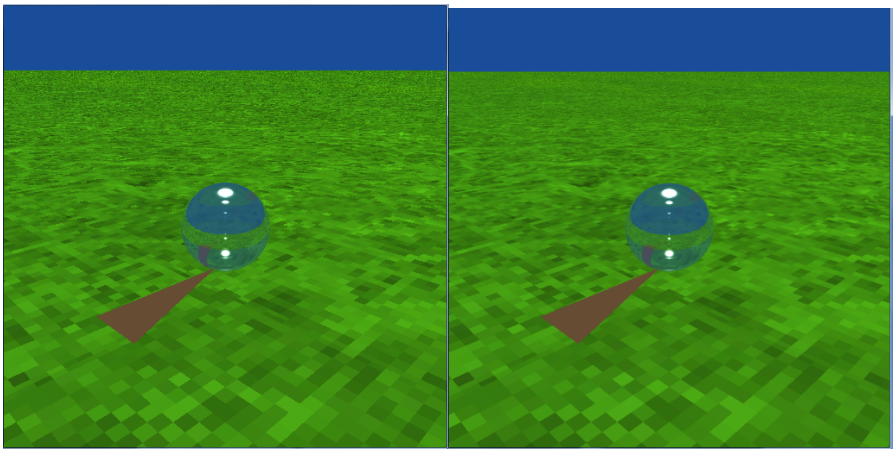
The next step is to implement bilinear sampling:

```
1  float4 Texture::sample_linear(const float3& texcoord) const
2  {
3    if(!fdata)
4      return make_float4(0.0f);
5
6    float a = (texcoord.x - floor(texcoord.x)) * width;
7    float b = (texcoord.y - floor(texcoord.y)) * height;
8
9    float2 alpha, beta, gamma, delta;
10
11   alpha = make_float2(floor(a), floor(b));
12   beta = make_float2(fmodf(ceil(a),width), floor(b));
13   delta = make_float2(fmodf(ceil(a),width), fmodf(ceil(b),height));
14   gamma = make_float2(floor(a), fmodf(ceil(b), height));
15
16   float4 c = lerp( fdata[ (int)alpha.x + (int)alpha.y * width ]      , fdata[ (int)beta.
         x + (int)beta.y * width ], a - floor(a) );
17   float4 d = lerp( fdata[ (int)gamma.x + (int)gamma.y * width ] , fdata[ (int)delta.x +
         (int)delta.y * width ], a - floor(a) );
18
19   return lerp(c, d, b - floor(b));
20 }
```

We first start by finding the texel coordinates of the sample. We then find the four pixels that are closest to this sample. After this is the bilinear interpolation performed. This is done by first doing a linear interpolation between the two sets of pixels that lies on top of each other. At the end another interpolation is done between the two points to produce the final result. The bilinear sampling produces the result seen on figure **??** As it can be seen this produces a much better result than point sampling. This is especially noticable when the texture gets magnified - is closer to the viewer. There is still aliasing farther ahead but it comes at a greater distance. When this happens the only way to improve the quality is to introduce more samples through stocastic sampling. Still the bilinear sampling does give a better result at a lover amount of jittered rays but

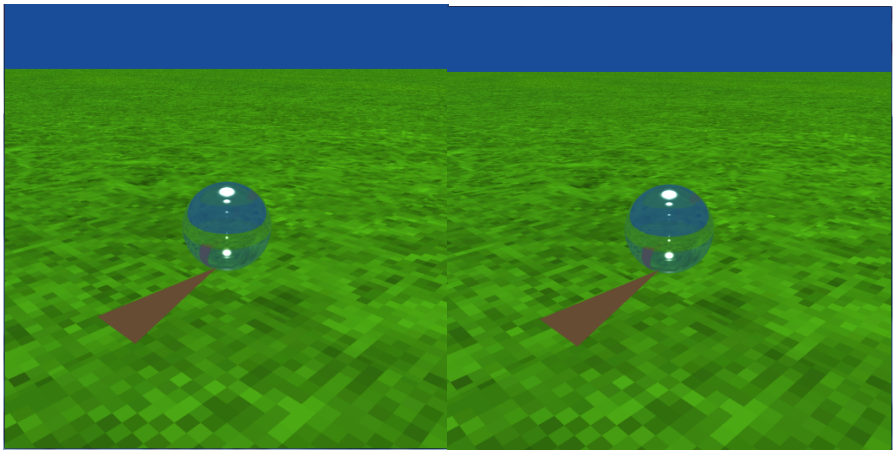
7 SCALING OF TEXTURE COORDINATES

When the texture scaling is increased to 10 it means the texture will repeat 10 times as often since the uv coordinates changes 10 times faster than if the scaling was set to 1. This results in figure 7.1



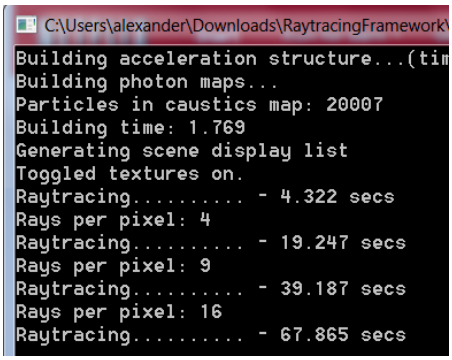
(a) Subdivision 0

(b) Subdivision 1



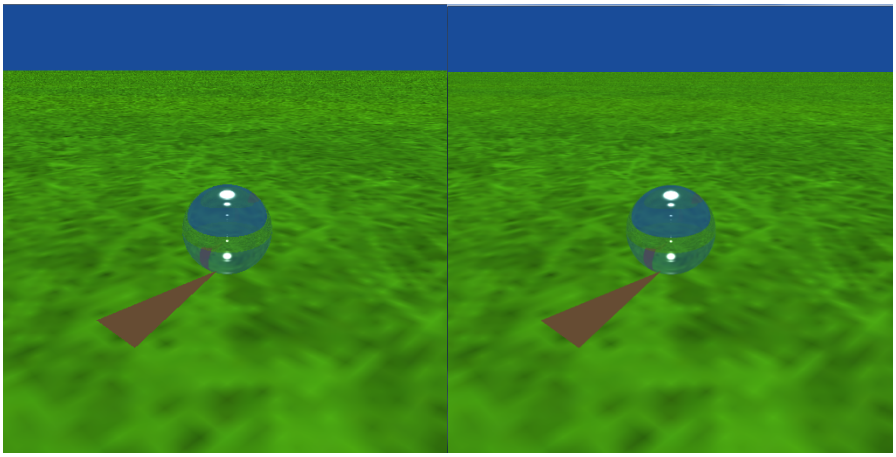
(c) Subdivision 2

(d) Subdivision 3

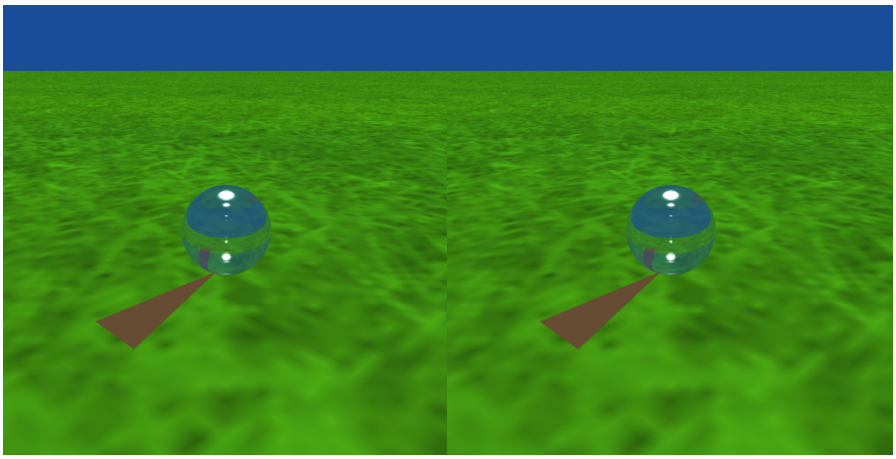


(e) Console output of subdivision

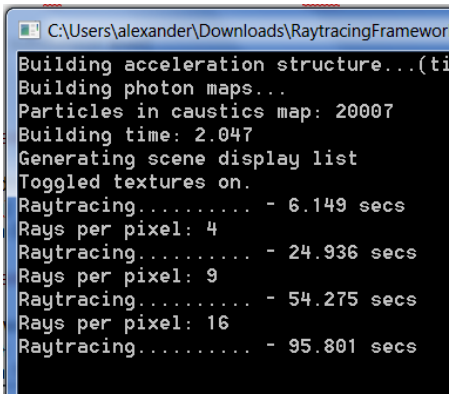
Figure 5.1: Renderings of default scene with nearest neighbour sampling



(a) Subdivision 0 (b) Subdivision 1



(c) Subdivision 2 (d) Subdivision 3



(e) Console output of subdivision

Figure 6.1: Renderings of default scene with linear interpolation sampling

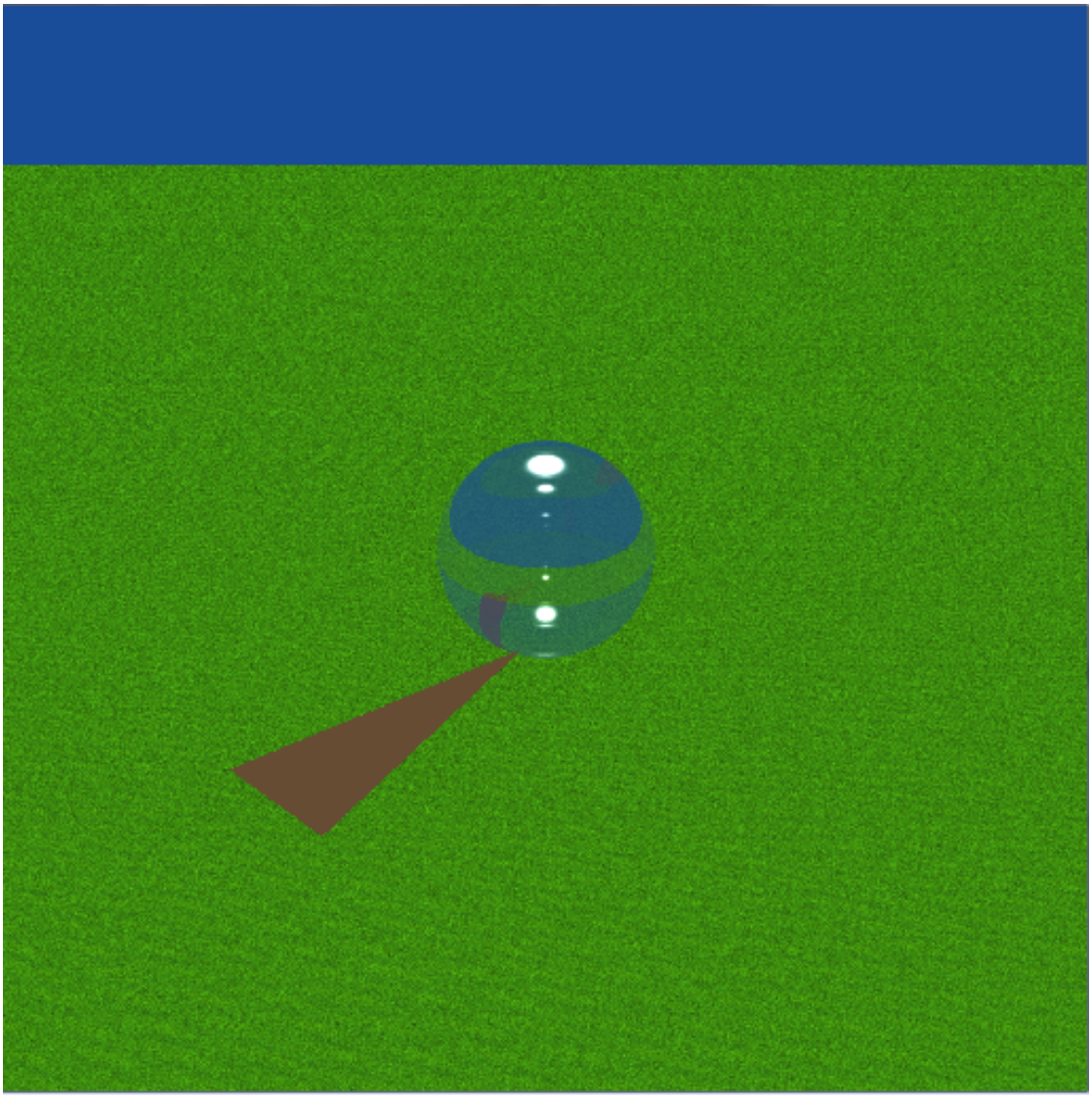


Figure 7.1: Image where texture is scaled with a factor of 10