Danmarks Tekniske Universitet

Project Report: Radiosity

Authors

Ojvind Nilsson - s213061

December 14, 2022

Contents

1 Introduction

During the course of 02562 Rendering we explored different introductory aspects of rendering of scenes using a ray tracing approach. One of the main topics that were implemented in the course was global illumination. Global illumination is a resource intensive task to do using only the CPU approach we were developing throughout the course. Global illumination is a technique which adds depth and realism to a rendered image by adding soft shadows in a way that light can bounce around a scene without being directly illuminated by a light source. It was for these reasons that the chosen project for the course was that of implementing the radiosity approach to global illumination. This project takes a look at a guided implementation using a framework provided by the course coordinator that acts as a foundation for the implementation for radiosity [1]. The methodology section will cover what radiosity is in greater detail, while the implementation section focuses on the code base for the project. The project was developed using additional course literature that was provided to guide the student in this rendering technique, as well as, an additional lecture provided by an external lecturer. In the discussion section this project also does a comparison of the radiosity approach to the ray tracing approach done in the course with some additional suggestions to improvements that could be made.

In order to get acquainted with the framework the course coordinator developed a project guide which was followed during the implementation of this project. The basic developments that were done in the guided project implementation of this project are the following:

- Scene discretization using meshes.
- Progressive approach to finding maximum energy in a scene every step.
- Implement a analytical implementation of radiosity calculations.
- Implement Nodal averaging for linear interpolation across patches.
- Hemicube rendering for computing visibility.

The radiosity framework can be ran by using Visual Studio and compiling the windows solution in the ‘lib‘ folder at the root of the project structure. Once that builds successfully the main part of the code can be ran by going to ‘exercises/Radiosity‘ folder and running the ‘radiosity.sln‘ solution.

2 Methodology

The project has been implemented using a different framework than the one that has been used in the course this far. Instead of the normal ray-tracing framework we will be using radiosity which is a rendering technique used to render globally illuminated scenes in an effective manner. Global illumination is the lighting of a scene by indirect light done by calculating the bounces light takes to get to a point on the scene. The way we implemented global illumination in the course was through a technique called ambient light which uses a constant to modify the diffuse reflectivity of objects.Radiosity refers to the definition that every scene has an innate source of energy that needs to be conserved or equalized throughout the scene. This approach treats a light source just like any other surface except for the fact that it is instantiated to a non-zero measure for radiosity (more on this later). Radiosity also makes the assumption that all surfaces are perfect Lambertian diffuse surfaces. So the technique basically assumes that each of surfaces in a scene will receive its radiosity value (which can be thought of as its brightness value) by the value it reflects and the value it radiates. Essentially all surfaces can be both a light source and a normal surface depending on its radiosity. This technique is fast to calculate in comparison to ray tracing (explored in the Discussions) and delivers good color bleeding and results of global illumination. The formulation for the project was derived from the supporting literature provided by the course instructor. [2] [3]. Alongside a radiosity lecture given by Utrecht University. [4]

2.1 Scene Subdivision

In order to perform the calculations instead of calculating the radiosity for every vertex in the scene, radiosity implements a scene subdivision similar to that which we explored in the course. The scene is subdivided into areas or as they are referred to "patches". These patches are areas that are mapped unto the world. As such the patches are object space coordinates where the calculation can take place regardless of the view of the camera. In comparison to the ray tracing we have done in the course which takes place in image space coordinates.

2.2 Radiosity Analytical Solution

Radiosity is built on Lambertian surfaces as such it builds upon the same reflectance function we have seen before in the course. The basic formulation for the radiosity method is shown below:

$$B_j = \rho_j H_j + E_j$$

This formula explains that the radiosity B value of a surface j is given by the reflectivity of a surface ρ_j multiplied by the Energy incident coming in H_j and the energy that the surface is emitting E_j. In this equation the only unknown factor is the energy of incident light hitting the surface of the object H_j. That is because like other models; The amount of light emitted from a surface must be specified as a parameter in the model, the location and intensity of

light sources must be specified, and the reflectivity of the surface must also be specified. We can derive the energy coming in to a surface by summing all the energy that is contributed by all other visible surfaces. The formula for which can be seen below:

$$H_j = \sum_{k=1}^{N} B_k f_{jk} \quad j = 1..N$$

Each of the patches can calculate their radiosity element B_j based on their emittance E_j and diffuse reflectance ρ_j. However, radiosity differs in that in the form factor of the patches differ. The form factor calculation is based on a few mathematical assumptions. First of that for every patch i we can know the distance from a point on the patch P to the point Q of another patch j. This is done by the distance from the center of a patch C_j to the center of any other patch C_k. We know that any two lines between the patches are almost parallel and that if $n_j \cdot n_k < 0$ then every point of patch j is visible form patch k and vice versa. We can then get a simplified form factor f_{jk} which is calculated using the formula below, Where u_{jk} is the unit vector pointing from the center of patch j to the center of patch k:

$$f_{jk} = \frac{1}{\pi} \frac{A_k}{||C_j - C_k||^2} |u_{jk} \cdot n_j| \cdot |u_{jk} \cdot n_k|$$

Given the result of each patches form factor we can then recombine the emittance facor based on the areas of the patches so that the radiosity function uses the formula below:

$$B_j = E_j + \rho_j \sum_k f_{jk} B_k \frac{A_k}{A_j}$$

It is worth noting that the fraction of light leaving element k and arriving at element j (aka the form factor) depends on; the shape of the patches, the relative orientation of the patches, visibility of the patches, and distance between the patches. As well, as the formula for the radiosity being a linear system in where in order to find the radiosity of one patch we first need to find the radiosity of a different patch. This is shown by restating the radiosity formula to the formula below:

$$E_j = B_j - \rho_j \sum_k f_{jk} B_k$$

Meaning that there needs to be a pipeline of solving radiosity where the form factor is calculated before the radiosity is calculated.

2.3 Radiosity Using Hemicubes

In order to solve the form factor calculation there are a few solutions like matrix building. However this can be quite intensive in complex scenes and instead you can make estimations of the form factor by sampling at each patch. The sampling is normally done like in ray-tracing in where you sample at unit hemisphere around the patch. The sample is then mapped unto the hemisphere which you can then check for visibility by using the z-buffer on

most render engines. This approach changes the way radiosity works by instead of storing color of the radiosity function it stores the identifier of the mapping index. The idea that the f_r is calculated through the integral of a hemisphere. That is if we surround a patch with a hemisphere and sample across it at random points we are then taking the orthographically projected radiosity form the other patch to be used for the calculation as shown in Figure 1:

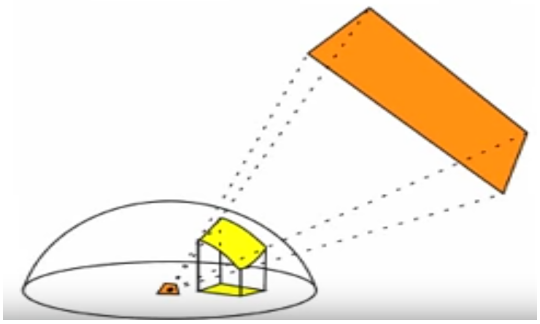

Figure 1: Hemisphere Mapping [4]

However as we say in the path-tracing for the ray-tracing framework this can be quite resource intensive. We can instead make an approximation of this approach using a so called hemicube that surrounds the patch for sampling instead. In order for the approximation to work we need to weight each of the sampling values to take into account that the world cannot be projected perfectly unto a flat surface like the face of a cube. Each pixel of the hemicube will there fore contribute to the form factor result. The formula for the form factor can thus be seen below:

$$F = \Delta F_a + \Delta F_b + \Delta F_c + ...$$

Where $a..N$ is the pixel mappings made from the hemisphere. This is best shown by Figure 2:

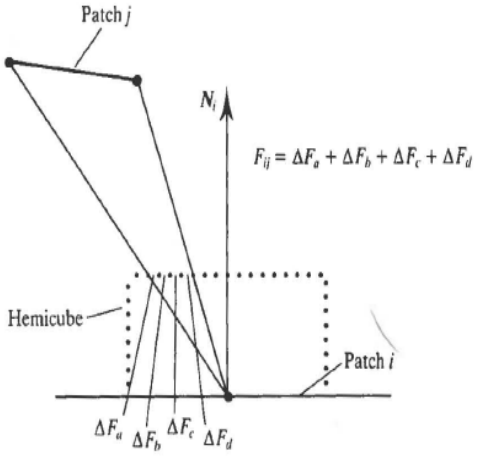

Figure 2: Hemicube Form Factor Calculation [3]

3 Design and Implementation

3.1 Mesh Integration and Scene Subdivision

Instead of the traditional ray marching approach we have been used to throughout the course, radiosity breaks down the scene into certain areas of energy transfer which are referred to as "patches". Patches are just a finite number of areas that the program subdivides the scene into. We can call the ‘Mesher.cpp‘ file constructor in order to subdivide the scene. We do this inside the the ‘radiosity.cpp‘ file which from now on will be referred to as the main file. Specifically we do this in the ‘main‘ function and replace the loading of the polygons and vertices with the following line of code:

```
1    Mesher::mesh(loaded_vertices, loaded_polygons, vertices, polygons,
        patch_subdiv_size, light_subdiv_size);
```

‘

These patches can be seen if the scene is switched to render a with ‘GL_LINE_LOOP‘. The way that one changes the GL drawing method in the code is under the main file inside the ‘displayMyPolygons‘ function. The render result of dividing the scene into patches and drawing it using the ‘GL_LINE_LOOP‘ method is shown in Figure 3.

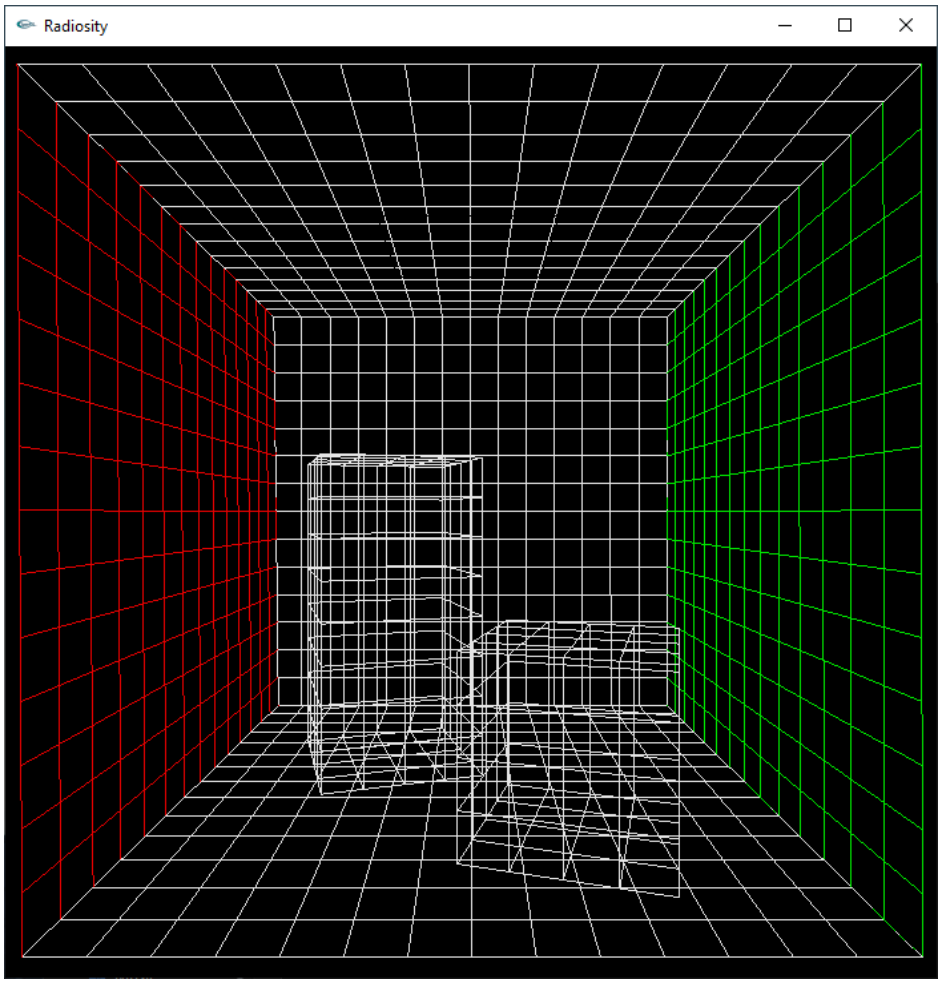

Figure 3: Patches Render

3.2 Analytical Radiosity

Lets change the scene a bit so that we can implement our light source and radiosity calculations. To do this we will use the Cornell Box scene without any of the blocks. Changing scenes is a bit simpler from the course framework in that we simply have to change one file instead of sending in command arguments. The file we change is the ‘Resource.xml‘ file under the resource files folder. This file is read to setup some of the variables in the scene we will look into at a later time. For now we can focus on changing the scene that is loaded by changing the variable under ‘file_name‘. We will be using ‘cornell_1.x3d‘ for the Cornell Box without the blocks.

We can now get into the development of the lighting for the radiosity technique. At first we will look at the **Progressive Refinement** implementation [3]. Essentially we need to implement four different functions in the ‘ProgRef.cpp‘ file in order to do this. First we need a function that returns the patch with maximum energy in the scene. This function is pretty straightforward and is shown below:

```
    MyPolygon* maxenergy_patch()
{
  return *std::max_element(polygons.begin(), polygons.end(), polygon_cmp);
}
```

‘

We then need a function that calculates form factors between this patch and all other patches. We do this in the analytical method shown in the supplementary literature [2]. This is the simplest form of radiosity in where first the scene is divided into a finite number of typically rectangular sections (patches). Then for every single one of these patches we calculate their Lambertian diffusion using the previously seen BRDF. Therefore the implementation of the function that calculates the form factors is shown below:

```
    MyPolygon* calcAnalyticalFF()
{
  // Reset all form factors to zero
  for (unsigned int i = 0; i < polygons.size(); i++)
    polygons[i]->formF = 0.0f;

  // Find the patch with maximum energy
  MyPolygon* maxEnergy = maxenergy_patch();

  // Calculate the form factors between the maximum patch and all other
     patches.
  // In this function, do it analytically [Watt 2000, Sec. 11.2] or [B, Sec.
      31.10].

  // Max Energy Patch variables
  Vec3f n_k = maxEnergy->normal;
  Vec3f C_k = maxEnergy->center;
  float A_k = maxEnergy->area;

  vector<MyPolygon*>::iterator iter;
  for (iter = polygons.begin(); iter != polygons.end(); ++iter)
  {
    if ((*iter) == maxEnergy) {
      continue; // Skip the maximum patch to not check it against itself.
    }

    Vec3f n_j = (*iter)->normal;

    // This gets rid of the green.
    //if (!(dot(n_j, n_k) < 0)) {
    //  continue; // Skip the current patch if it is not visible to the
```

```
        maximum patch.
30      //}
31
32      float A_j = (*iter)->area;
33      Vec3f C_j = (*iter)->center;
34
35      Vec3f distance = C_j - C_k;
36      float r2 = dot(distance, distance);
37
38      Vec3f u_jk = normalize(distance);
39
40      float x = max(dot(n_j, -u_jk), 0.0f);
41      float y = max(dot(n_k, u_jk), 0.0f);
42      float scale = x * y;
43
44      // Calculate the form factor between the maximum patch and the current
            patch.
45      (*iter)->formF = M_1_PI * (A_j / (r2)) * scale;
46    }
47
48    // Return the maximum patch
49    return maxEnergy;
50  }
```

‘

We also need a function that distributes the energy across the patches and throughout the scene. This is the radiosity function. The implementation of which is shown in the code below:

```
1   bool distributeEnergy(MyPolygon* maxP)
2   {
3       if (maxP == nullptr)
4       return false;
5
6       // Distribute energy from the maximum patch to all other patches.
7       // The energy of the maximum patch is in maxP->unshot_rad (see
            DataFormat.h).
8       vector<MyPolygon*>::iterator iter;
9       for (iter = polygons.begin(); iter != polygons.end(); ++iter) {
10      if ((*iter) == maxP){
11          continue; // Skip the maximum patch to not check it against itself.
12      }
13
14      float A_j = (*iter)->area;
```

```
    float A_k = maxP->area;

    Vec3f E = maxP->unshot_rad;
    float f_jk = (*iter)->formF;
    Vec3f rho_j = (*iter)->diffuse;

    // Calculate the radiosity of the current patch.
    Vec3f B = E * rho_j * f_jk * (A_k / A_j);

    (*iter)->rad += B;
    (*iter)->unshot_rad += B;

    }

  // Set the unshot radiosity of the maximum patch to zero and return true
    maxP->unshot_rad = Vec3f(0.0, 0.0, 0.0);
    return true;
}
```

‘

We then use the result of the two previous function which is the radiosity per patch and use it to set the color. We iterate through the patches again and according to the hint in the function we use the nodal average to compute the color of all the patches. This process can be seen in the code below:

```
    void colorReconstruction()
{
  // Set the colour of all patches by computing their radiances.
  // (Use nodal averaging to compute the colour of all vertices
  //  when you get to this part of the exercises.)
  vector<MyPolygon*>::iterator iter;
  for (iter = polygons.begin(); iter != polygons.end(); ++iter)
  {
    (*iter)->color = (*iter)->rad * M_1_PI;
  }
}
```

‘

If we then go back to the main file and under the ‘display‘ function we call the ‘distributeEnergy‘ and the ‘colorReconstruction‘ functions inside we get a radiosity rendering that gets progressively better as the main OpenGL loop calls the display function. See Results section for the results of this implementation.

3.3 Implementing Smooth Shading

If we see our results from the previous section we can see the individual patches in the scene, this was useful throughout the testing and development of our formulas but we don't want this effect in the production version of the renderer. If we go back to the implementation of the 'colorReconstruction' method instead of giving each patch the color we can give each of the vertices a color representation. To do this we use the hint in the function and do an averaging of the colors per vertex as shown in the code below:

```
1  // Reset all colours to zero
2  for (int i = 0; i < vertices.size(); ++i)
3  {
4    vertices[i]->color = Vec3f(0.0, 0.0, 0.0);
5    vertices[i]->colorcount = 0;
6  }
7
8  // Sum the colours of all vertices
9  vector<MyPolygon *>::iterator iter;
10 for (iter = polygons.begin(); iter != polygons.end(); ++iter)
11 {
12   (*iter)->color = (*iter)->rad * M_1_PI;
13   for (int i = 0; i < (*iter)->vertices; ++i)
14   {
15     vertices[(*iter)->vertex[i]]->color += (*iter)->color;
16     vertices[(*iter)->vertex[i]]->colorcount += 1;
17   }
18 }
19
20 // Average the colours of all vertices
21 for (int i = 0; i < vertices.size(); ++i)
22 {
23   vertices[i]->color /= (float)vertices[i]->colorcount;
24 }
```

'

Once we have this implemented this we have to change the 'displayMyPolygons' function inside the main file so that it uses the vertex color which is done commenting out the 'glColor3f' call that is inside already and instead moving it inside the vertex loop that is there as shown in the code below:

```
1 for (int j=0;j<polygons[i]->vertices;j++) {
2
3       Vec3f position = vertices[polygons[i]->vertex[j]]->position;
4       Vec3f color = vertices[polygons[i]->vertex[j]]->color;
5       glColor3f(color[0], color[1], color[2]);
```

```
      glVertex3f(position[0], position[1], position[2]);
    }
```

‘

3.4 Hemicubes

Luckily the implementation of hemicubes and sampling them has been provided by the coordinator of the course and the calculation of each of the cube pixel form factors do not need to be calculated. We can do so by calling the ‘getDeltaFormFactor‘ function for each of the pixels of the hemicube. Each pixel of the hemicube will there fore contribute to the form factor result so we can simply add that to be our resulting form factor. We can do this by implementing the final function ‘calcFF‘ within the ‘ProgRef.cpp‘ file which essentially will be responsible for iterating through the hemicube on each patch by calling the ‘renderScene‘ function on the hemicube. This implementation of the ‘calcFF‘ will also require the implementation of the ‘renderPatchIDs‘ function who’s responsibility will be to set the color to the patch index using ‘glColor3ub‘ similar to how we previously implemented the ‘colorReconstruction‘ function. The implementation of the ‘renderPatchIDs‘ function can be seen in the code below:

```
void renderPatchIDs()
{
  // Render all polygons in the scene as in displayMyPolygons,
  // but set the colour to the patch index using glColor3ub.
  // Look at the Hemicube::getIndex function to see how the
  // indices are read back.
  for (int i = 0; i < polygons.size(); i++) {
    if (4 == polygons[i]->vertices) glBegin(GL_QUADS);
    else if (3 == polygons[i]->vertices) glBegin(GL_TRIANGLES);
    else assert(false); // illegal number of vertices

    for (int j = 0; j < polygons[i]->vertices; j++) {
      Vec3f position = vertices[polygons[i]->vertex[j]]->position;
      GLubyte i1 = (i + 1) & 255;
      GLubyte i2 = ((i + 1) & (255 << 8)) >> 8;
      GLubyte i3 = ((i + 1) & (255 << 16)) >> 16;
      glColor3ub(i3, i2, i1);
      glVertex3f(position[0], position[1], position[2]);
    }
    glEnd();
  }
}
```

‘

Then we take the form factor result at each pixel on the hemicube and use it to generate our overall form factor as shown in the code below:

```
MyPolygon* calcFF(Hemicube* hemicube)
{
  // Reset all form factors to zero
  for (unsigned int i = 0; i < polygons.size(); i++)
    polygons[i]->formF = 0;

  // Find the patch with maximum energy
  MyPolygon* maxEnergy = maxenergy_patch();

  // Compute a normalized up vector for the maximum patch
  // (use the patch center and one of the patch vertices, for example)
  Vec3f up = normalize(maxEnergy->center - vertices[maxEnergy->vertex[0]]->
     position);

  // Render patch IDs to the hemicube and read back the index buffer
  hemicube->renderScene(maxEnergy->center, up, maxEnergy->normal, &
     renderPatchIDs);
  hemicube->readIndexBuffer();

  // Compute form factors by stepping through the pixels of the hemicube
  // and calling hemicube->getDeltaFormFactor(...).
  for (int y = 0; y < hemicube->rendersize; ++y)
    for (int x = 0; x < hemicube->rendersize; ++x)
    {
      int idx = hemicube->getIndex(x, y) - 1;
      if (idx >= 0)
        polygons[idx]->formF += hemicube->getDeltaFormFactor(x, y);
    }

  // Return the maximum patch
  return maxEnergy;
}
```

‘

To test the implementation we return to the ‘display‘ function in the main file and instead of calling ‘calcAnalyticalFF‘ we use ‘calcFF(hemicube)‘ we can determine which way we want to render by using the variable ‘use_hemicube‘ which can be set in the ‘form_factor_method‘ variable in the ‘Resource.xml‘ file. As such we determine which method to render in using the following code:

```
if(use_hemicube)
    distributeEnergy(calcFF(hemicube));
  else
    distributeEnergy(calcAnalyticalFF());

  colorReconstruction();
```

‘

Then if we set the variable in the ‘Resource.xml‘ file to hemicube we render using the hemicube method but if it is set to anything else we render using the analytical method.

4 Results

As the project was a step implementation of the radiosity technique with multiple results this section is split into several subsection show casing the results of their corresponding part. It was decided to show the results this way instead of just a final result of the entire project to show how development occurred and specifically to show how the analytical implementation differed from that of the hemicube implementation.

4.1 Results of Analytical Radiosity

This section covers the result of the analytical radiosity before the implementation of the Gouraud shading. This was the first step into the development of the radiosity engine. Radiosity was implemented in a progressive manner which means that the rendering can increase in quality as the samples of the radiosity function are increased. The analytical version of implementation has hard shading across the patches which makes the render have a blocky appearance. This is due to each of the patches having radiosity measures that are different enough to be noticeable. We can see in Figure 4 how the light source at the top of the scene is a consistent color. This is due to the variance between the radiosity levels of the light source being so minute compared to the rest of the scene.

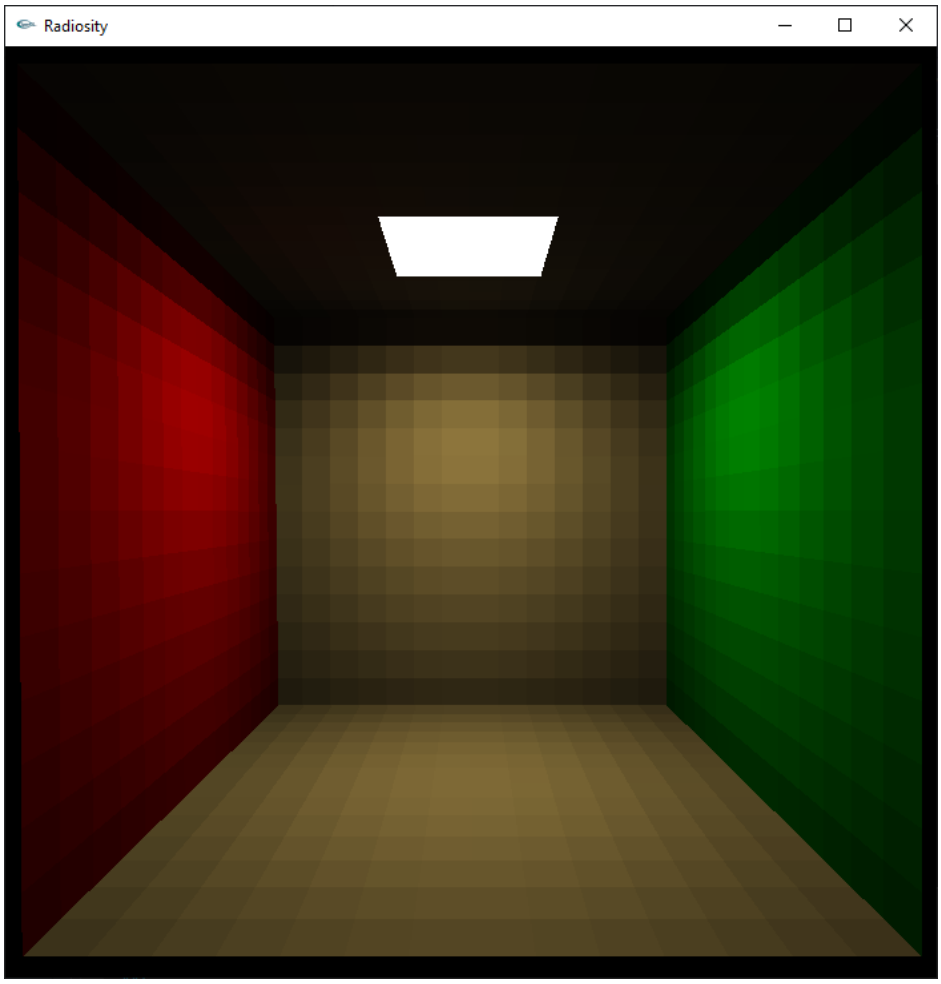

Figure 4: Initial Render of Analytical Method

Once the radiosity function has sufficient samples (usually around 30 seconds of rendering) the scene's radiosity levels will propagate cleanly across the scene. This results in a render that really showcases the benefits of the radiosity method for global illumination. In Figure 5 we can see that places where the colored walls meet the tan colored planes(especially on the roof) there is some bounce back of the light which colors the tan plane. This color bleed effect is what happens as light bounces from colored surface to another surface.

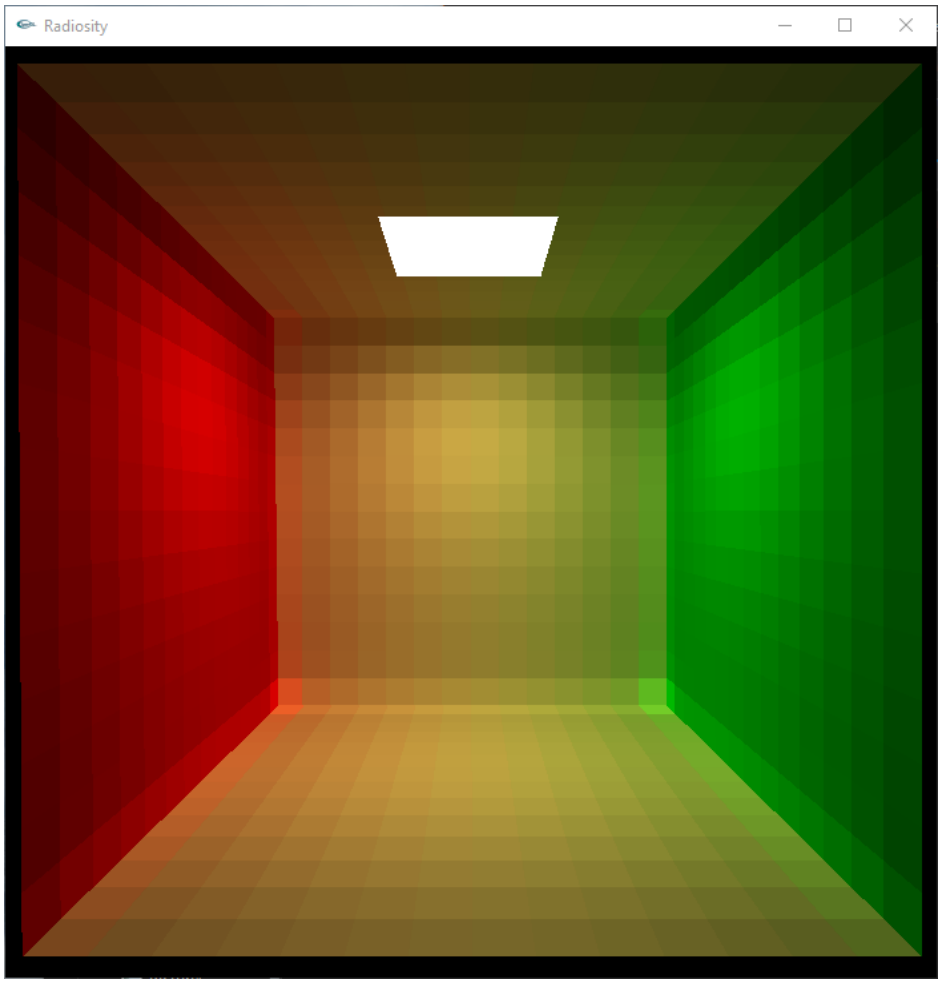

Figure 5: Final Render of Analytical Method

This effect of color bleed does not show up in the render results shown by Cornell which can be seen in Figure 6. However, the effect does make the scene feel like a more vibrant room filled entirely by a ceiling light. Ignoring this effect we can see that in comparison to the final render of the radiosity, the render provided Cornell does match the lighting provided by radiosity. This is true with the highlights at the top center of the walls that are close to the light.

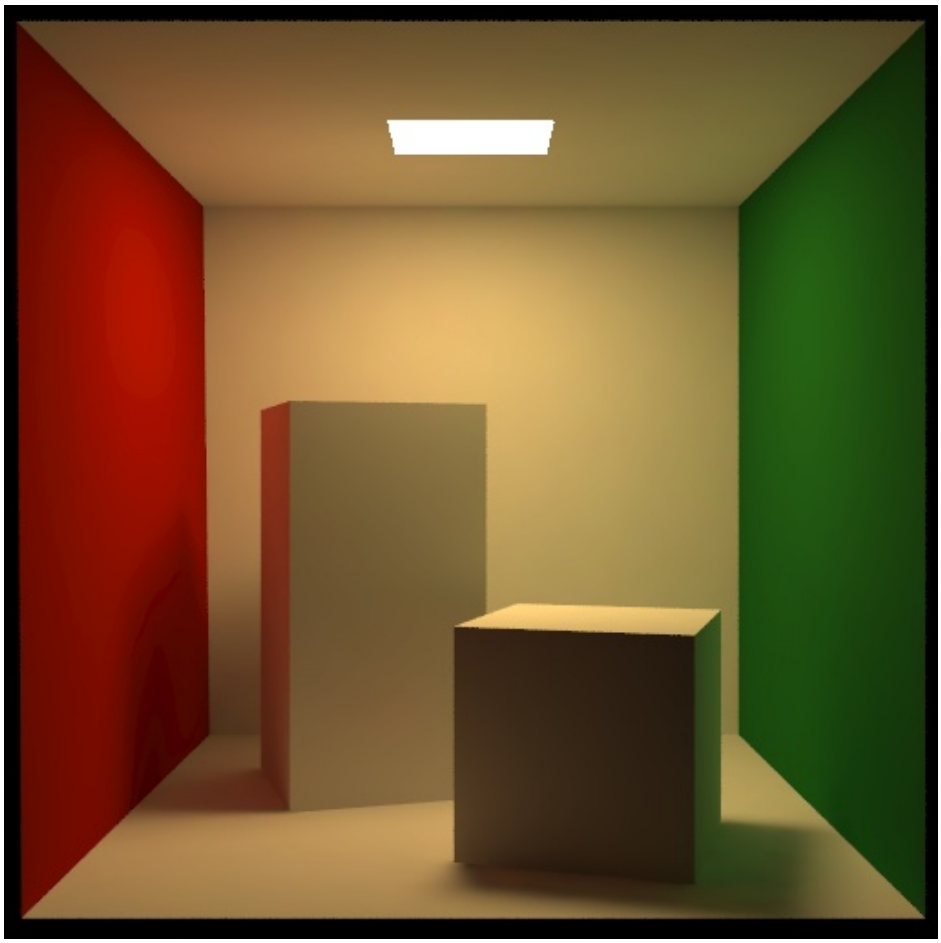

Figure 6: Render Shown by Cornell [5]

4.2 Results of Smooth Shading

This section covers the results of the analytical radiosity after the implementation of the Gouraud shading. The implementation of the coloring not being based on the patches but instead on the vertex location of the patches really improved the lighting to be closer to that in the Cornell render. An unexpected effect that the smooth shading had was that of bringing back the shadowing that happens at the corners of the room. These dark patches are shown in the initial rendering of the analytical method but get washed out as sampling increases. They are still there in the 5 but are harder to see as the contrast is not as strong. The results of the smooth shading can be seen in Figure 4.2.

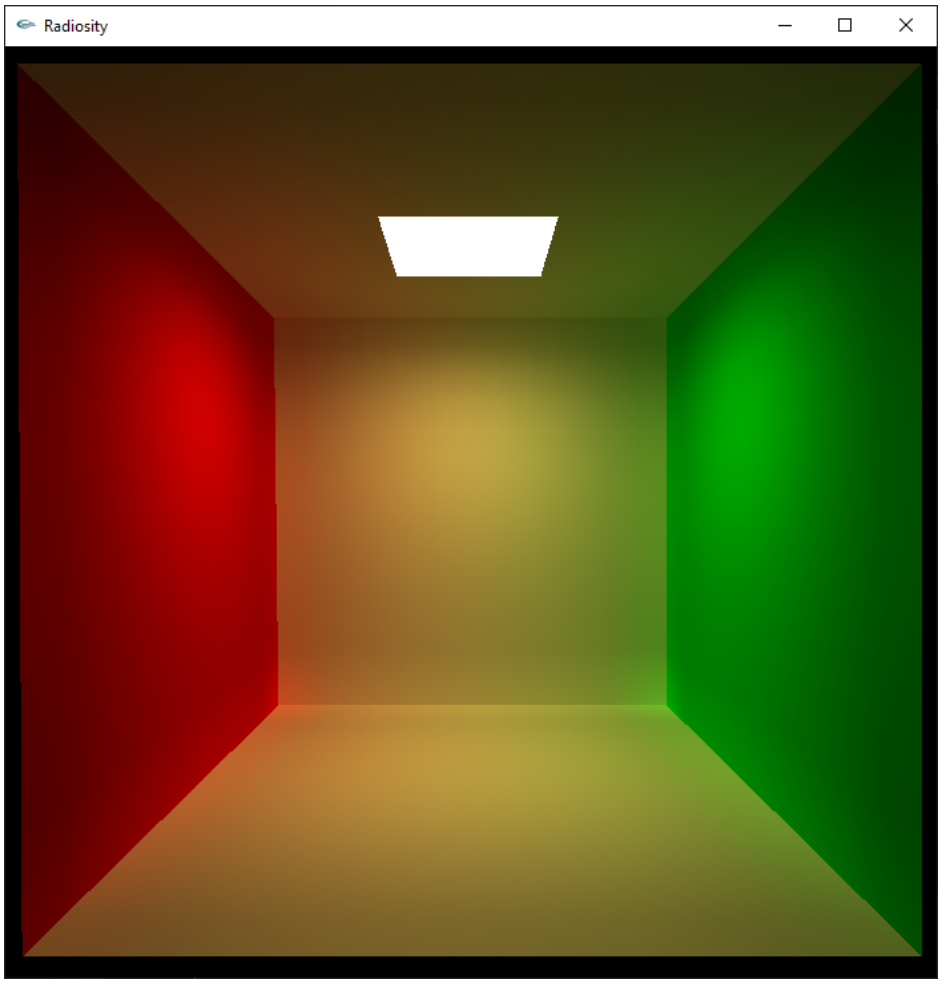

Figure 7: Render of Analytical Method Using Smooth Shading

4.3 Results of Hemicube Radiosity

This section covers the results off the hemicube radiosity and shows how how shadows are incorporated into the radiosity rendering. The use of the hemicube is an approximation for global illumination and the quality of the render depends on the size of the hemicube as well as the subdivision level that was used in the scene (discussed further in the discussions section). Overall, the implementation of the hemicube improved the scene not just through shadowing but removed the hard green and red bottom corners of the room. Radiosity has rendered the Cornell box in an accurate manner. One of the nice things that the radiosity method does is edges of planes and intersections. The top of the room has this nice border of the intersections between the walls where there is a line of shadow. One place that this hemicube method improves over the method used in the course is the red reflection on the side of the large rectangular polygon.

Figure 8: Render of Hemicube Render

5 Discussion

This section expands on the concepts that were learned with the implementation of the project. This section also covers a discussion on how the radiosity engine compares to the ray-tracing algorithm that was developed in the course. Lastly this section goes over a quick discussion of possible improvements that could be done to the radiosity engine so that the results are closer to the render result provided by Cornell.

5.1 Concepts in the Project

One of the foundational aspects to the radiosity method is the breaking down the scene into patches. These patches are what is used to calculate the radiosity values to then radiate off to other patches that are visible. This subdivision of the scene occurs first in the render pipeline under our implementation. When testing the engine it was unnoticeable the time it took to subdivide the scene as the vast majority of the render time was held up by the

calculation and sampling of the hemicube. This is because the Cornell box and blocks scene that we are rendering is not very polygon complex. That is the entire scene is made up of just a handful of polygons; one for each of the faces of the boxes and the rooms, one for the roof, one for the floor, one for the light. If we remember some of the complex scenes that we rendered in the course like that of the utah teapot. The teapot was a polygon complex object to render. If we now think that such objects would need to be subdivided for the radiosity to perform the calculations, it is problematic to think that the render time would increase exponentially with the complexity of the scene.

5.2 Radiosity in Comparison to Ray Tracing

During the implementation of the project, it was stated quite early on that radiosity treats every surface as a diffuse surface. If we compare that to the implementations done for the ray tracing engine we developed in the course, we can immediately think of some drawbacks that such diffuse renderer could have. During the course we talked about lighting systems that handle shading in a non-diffuse manner. Such a type of lighting was specular lighting. The implementation of the project did not attempt to implement a specular object such as glass ball like we did in the course though the formulas presented show that radiosity depend on diffuse values. One of the benefits that radiosity would have due to the same factor would be that it handles the diffuse reflections of color quite well, something that we had not implemented throughout the course. Radiosity not being able to render specular reflection would mean that it would need to be combined with the ray tracing approach to handle both which would still bring the benefits of radiosity over ray tracing. One of those benefits that was tested for during the implementation of the project was the speed at which radiosity was calculated for the scene. Keep in mind both the engines rendered the simple scene that is the Cornell box quite fast but the difference in the testing was that for around similar quality results ray tracing took a substantial amount of time. The problem that ray tracing has when doing soft shadow and global illumination was that the image would start of with a lot of noise and black dots. These black dots take quite some time to be corrected by ray tracing. Where as the radiosity engine has a deliverable result straight away but with more sampling time the image quality simply increases.

5.3 Future Work

One of the main drawbacks from the render results of the radiosity engine is that it doesn't have any implementation of anti-aliasing currently. This reduces the quality of the render as especially at the edges of objects it is quite obvious the pixelization that occurs. A good implementation of anti-aliasing will improve the render of the blocks in the scene especially on the sloped edges at the top of the blocks. Radiosity depends on a few assumption in the way that we have implemented for this project and thus the rendered scene is not quite exact but based as an approximation. Reducing the needed approximations would be a logical step to improving the radiosity engine. One of such approximations would be that of mapping the world to a hemicube instead of a hemisphere. Implementing the cosine weighted hemisphere like we have done in the ray tracing engine would improve the results

of the render. As previously stated radiosity is an overall efficient method of doing diffuse lighting, combining the diffuse lighting with other lighting methods such as the glossy and reflective lighting of the ray tracer along with the photon mapping for the render equation would be desired.

References

[1] "02562 Rendering - Introduction, https://courses.compute.dtu.dk/02562/."

[2] J. F. Hughes, A. V. Dam, M. McGuire, D. F. Sklar, J. D. Foley, S. K. Feiner, and K. Akeley, "Computer Graphics: Principles and Practice (3rd Edition)," p. 1264, 2013.

[3] A. Watt, "3D Computer Graphics (3rd Edition)," p. 624, 1999.

[4] "Computer Graphics 2011, Lect. 10(1) - Radiosity - YouTube."

[5] "Cornell Box Data, https://www.graphics.cornell.edu/online/box/data.html."