Rendering assignment week 5-6

Alexander Birke
study nr: `124044` email: `alex@alexanderbirke.dk`

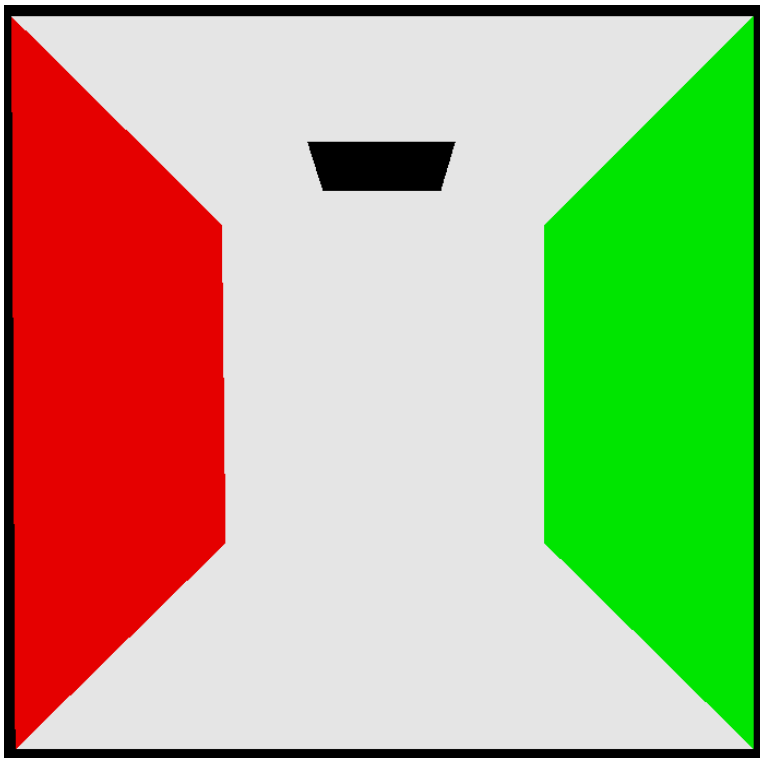
December 17, 2012

Introduction

This is my handin for week 5 and 6 of the Rendering course. The handin concerns the Radiosity method, which is specifically suited to do render LD*E paths.

Part 1 - Basic Setup & Meshing

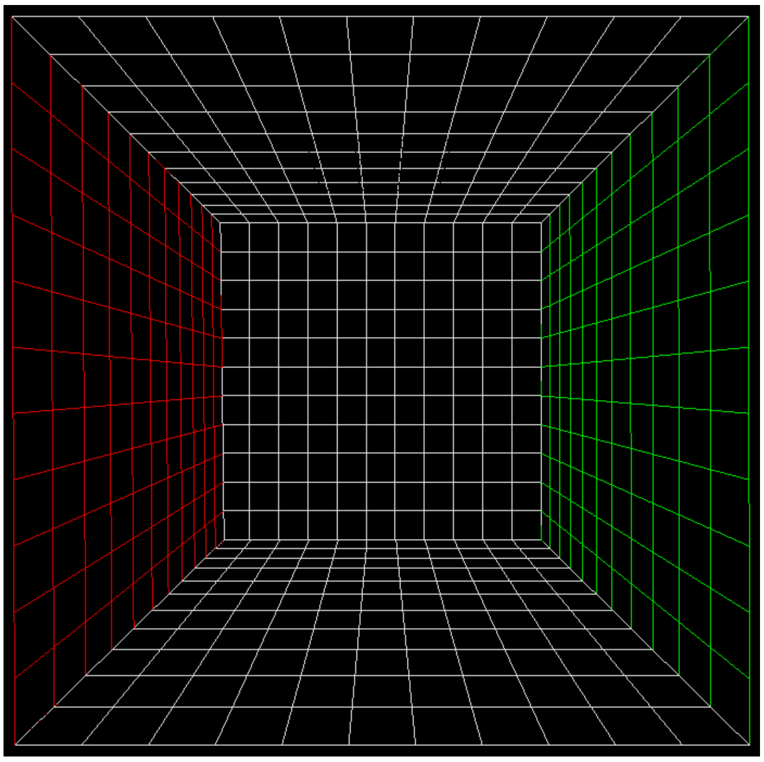
When the radiosity framework is first compiled and run it produces the following flat shaded image:



The next part is to implement tesselation of the scene geometry. This is done with the Mesher library and the code for doing this is the following:

```
//output variables
vector<MyPolygon*> tesselated_polygons;
vector<MyVertex*> tesselated_vertices;

//patch sizes
const float faceSize = 20;
const float lightSize = 20;

//tesselation
Mesher::mesh(loaded_vertices,loaded_polygons,tesselated_vertices,
    tesselated_polygons, faceSize, lightSize);

polygons = tesselated_polygons;
vertices = tesselated_vertices;
```

And the result looks like this:



Part 2 - The Progressive Refinement Method

Progressive refinement is a method to do radiosity that is designed to over time produce a better and better result. This way the process can be cancelled if the artist finds that the result generated is not satisfactory. In the normal formulation of the radiosity method each patch gathers light from each other. In the progressive refinement method the radiosity is shot from the "brightest" patch, the patch with most unshot radiosity, to all the other patches in the scene. A patch thus has to fields, one that stores how much radiosity it has recieved overall, and another that stores how much radiosity it has recieved that it has not distributed out to other patches yet. The field for overall radiosity is used to calculate the color of the patch.

The first part is to calculate the form factor between the patches based on the following analytical formula:

$$F_{dAidAj} = \int_{Aj} \frac{\cos \Phi i \cos \Phi j}{\pi r^2} \mathrm{d}A_j$$

If the patches are sufficiently small we do not need to take the integral over the patch since the form factor does not change that much when it is sufficiently small The function can therefore be calculated in the following way:

```
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* maxEnergyPatch = maxenergy_patch();

   //for all patches
  for(unsigned int i = 0; i < polygons.size(); i++)
  {
      //find the vector between the two patches
    Vec3f lineBetweenPatches = maxEnergyPatch->center - polygons[i]->center
        ;
      //and the normalized version of it
    Vec3f normalizedLine = lineBetweenPatches;
    normalizedLine.normalize();

    //calculate phi for each patch

    float phi1 = dot(normalizedLine, polygons[i]->normal);

    if(phi1 < 0)
      phi1 = 0;

    float phi2 = dot(-normalizedLine, maxEnergyPatch->normal);

    if(phi2 < 0)
      phi2 = 0;

    //finally calculate the form factor
    polygons[i]->formF = polygons[i]->area * phi1*phi2 * M_1_PI / pow(
        lineBetweenPatches.length(), 2 );
  }
  // Return the maximum patch
  return maxEnergyPatch;
}
```

The next part is the function that distributes the unshot radiosity from the patch with most unshot radiosity to the other patches. This is based on the formula:

$$B_j = R_j B_i F_i j \frac{A_i}{A_j}$$

Where i refers to the patch being shot from and j refers to the patch being shot at. The code for doing this is shown below:

```
bool distributeEnergy(MyPolygon* maxP)
{
  //if a null pointer is given, return false
  if(maxP == 0)
    return false;

  // Distribute energy from the maximum patch to all other patches.
  for(unsigned int i = 0; i < polygons.size(); i++)
  {
    //we do not distribute energy from the max energy patch onto itself
    if(polygons[i] == maxP)
    continue;

      //calculate the radiosity recieved by patch i
    Vec3f radiosity = polygons[i]->diffuse * polygons[i]->formF * ( maxP->
        area / polygons[i]->area) * maxP->unshot_rad;

    //add this to patch i's radiosity and unshot radiosity
    polygons[i]->unshot_rad += radiosity;
    polygons[i]->rad += radiosity;
  }

  // Set the unshot radiosity of the maximum patch to zero and return true
  maxP->unshot_rad = Vec3f(0);

  return true;
}
```

The function is quite simple. First it is tested if the function is passed a null pointer, if that is the case it returns false. Next it loops over all patches, skips the patch with max energy since a patch cannot distribute energy onto itself. Then the above formula is calculated and the result added to the current patch's radiosity and unshot radiosity.

Next, we need a function to calculate the color of each patch. The color is equal to the radiance of the patch. The relationship between radiosity and radiance for a diffuse reflector is:

$$B = L\pi$$

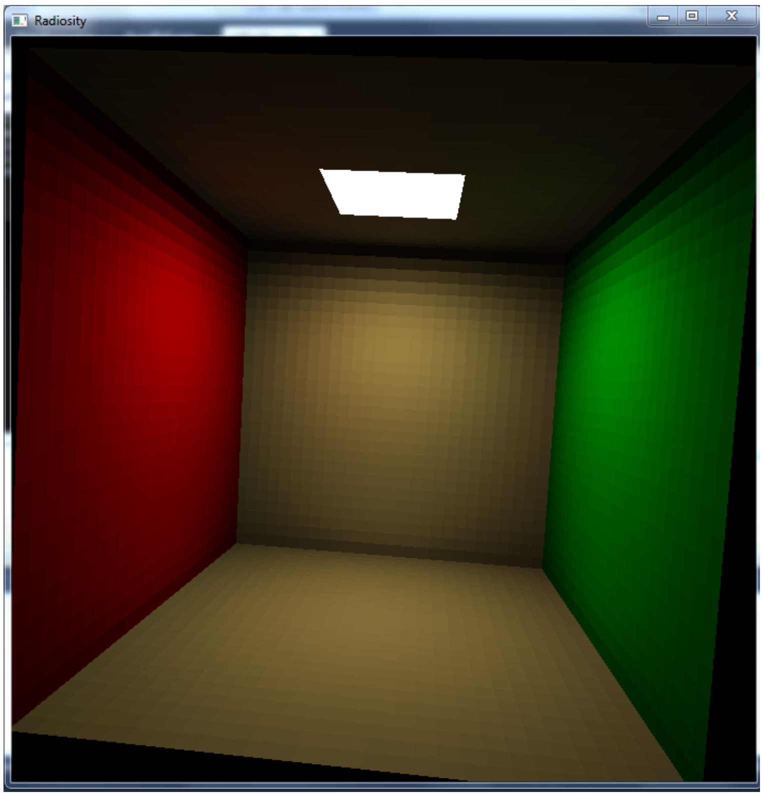
So this means the color is obtained by dividing the radiosity with phi.

```
void colorReconstruction()
{
  for(unsigned int i = 0; i < polygons.size(); i++)
  {
    polygons[i]->color =  polygons[i]->rad * M_1_PI;
  }
}
```

The last thing in this part is to call the function from inside the display function:

```
void display()
{
    MyPolygon* maxP = calcAnalyticalFF();

    distributeEnergy(maxP);

    colorReconstruction();

    glViewport(0,0,screen_size,screen_size);
    ball->do_spin();
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColor3f(1.0,0.0,0.0);
    glLoadIdentity();
    ball->set_gl_modelview();
    displayMyPolygons();
  glutSwapBuffers();
}
```

The result looks like this:



Part 3 - Smoothing via Nodal averaging

The main task here is to compute the color of each vertex based on the color of the patches that it is a part of. The strategy used it to first reset the colors of all the vertices, then go trough all the patches and add the color of that patch to the vertices it is connected to. To obtain the color of a vertex we then divide with the number of patches it is connected to thus finding the mean color value. The function to calculate color is therefore changed to:

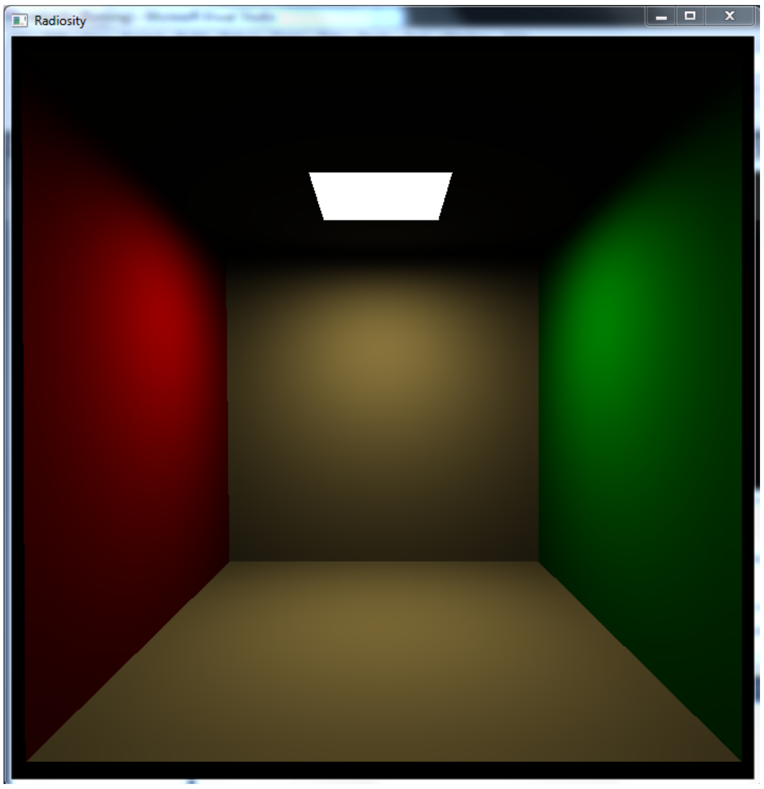
```
void colorReconstruction()
```

```
{
  for(unsigned int i = 0 ; i < vertices.size() ; i++)
  {
    vertices[i]->color = Vec3f(0);
    vertices[i]->colorcount = 0;
  }

  for(unsigned int i = 0; i < polygons.size(); i++)
  {
    polygons[i]->color =  polygons[i]->rad * M_1_PI;

    for(unsigned int k = 0 ; k < polygons[i]->vertices ; k++)
    {
      vertices[polygons[i]->vertex[k] ]->color += polygons[i]->color;
      vertices[polygons[i]->vertex[k] ]->colorcount += 1;
    }
  }

  for(unsigned int i = 0 ; i < vertices.size() ; i++)
  {
    vertices[i]->color =  vertices[i]->color / vertices[i]->colorcount;
  }
}
```

Next we need to change the DisplayMyPolygons function so it uses the vertex colors instead of the patch's color:

```

void displayMyPolygons()
{
  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++)
    {
      glColor3f(vertices[ polygons[i]->vertex[j] ]->color[0], vertices[
          polygons[i]->vertex[j]]->color[1], vertices[ polygons[i]->vertex[j
          ]]->color[2]);

      Vec3f position = vertices[polygons[i]->vertex[j]]->position;
      glVertex3f(position[0], position[1], position[2]);
    }
    glEnd();
  }
}
```

Since OpenGL's fixed function pipeline is based on Garroud shading and the shading is already set to be smooth we do not have to do more. The result looks like this:



Part 4 - Occlusion via the Hemicube Method

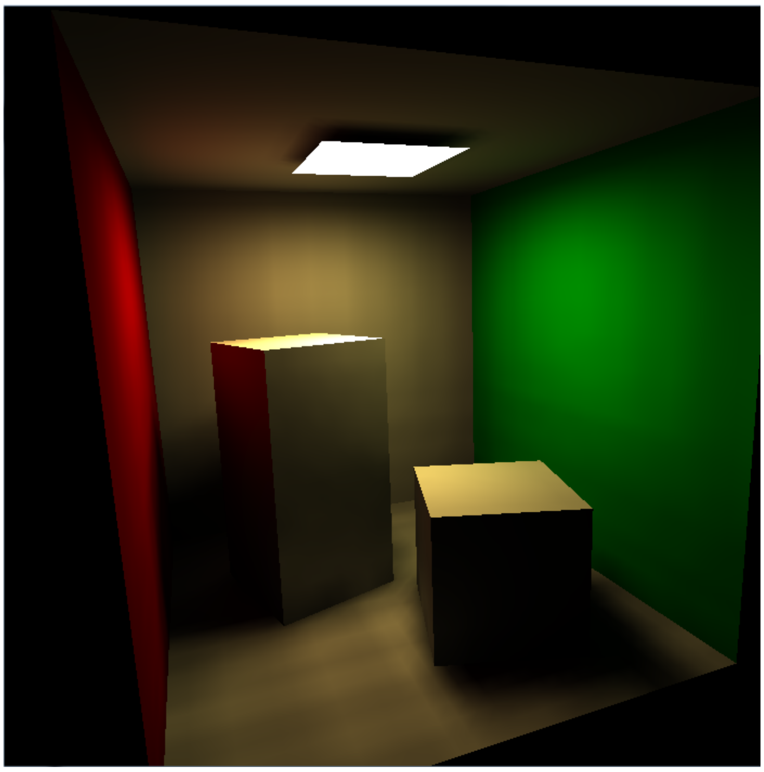
There is two functions that need to be implemented in order to use the hemicube method, the first is renderPatchIDs that prepares the scene to be rendered to the hemicube by setting the color of each patch to its index (plus one to avoid that areas where there are no patches are considered as the first patch in the vector). The main task is therefore to change what is done in the DisplayMyPolygons function to use the index of the patch instead. The index therefore needs to be changed into a RGB value. This is done with bit masks and bit shifting:

```
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)
    //if(4==polygons[i]->vertices)
      glBegin(GL_QUADS);
    else if (3==polygons[i]->vertices)
      glBegin(GL_TRIANGLES);
    else
      assert(false); // illegal number of vertices

    unsigned int index = i + 1;

    unsigned char r = ( index & 0xff0000 ) >> 16;
    unsigned char g = ( index & 0x00ff00 ) >> 8;
    unsigned char b = ( index & 0x0000ff );

    glColor3ub(r, g, b);

    for (int j=0;j<polygons[i]->vertices;j++)
    {
      Vec3f position = vertices[polygons[i]->vertex[j]]->position;
      glVertex3f(position[0], position[1], position[2]);
    }
    glEnd();
    }
}
```

Next, the form factors can be calculated. Here the hemicube class is used. This requires that a virtual camera is set up which looks in the direction of the max energy patch's surface normal, has its eye point in the plane of the patch and the up vector also lying in the same plane. When the scene has been rendered the indexes are read from the frame buffer and then used to calculate the form factor of each patch. The function looks like this:

```
MyPolygon* calcFF(Hemicube* hemicube)
{
   for(unsigned int i = 0; i < polygons.size(); i++)
   {
    polygons[i]->formF = 0;
   }

    MyPolygon* maxP = maxenergy_patch();

  Vec3f up = vertices[ maxP->vertex[0] ]->position - maxP->center;

  hemicube->renderScene(maxP->center, up, maxP->normal, renderPatchIDs);

  hemicube->readIndexBuffer();

  for(unsigned int i = 0 ; i < hemicube->rendersize ; i++)
  {
    for(unsigned int k = 0 ; k < hemicube->rendersize ; k++)
    {
      unsigned int index = hemicube->getIndex(i,k);

      if(index > 0)
      {
        float f = hemicube->getDeltaFormFactor(i,k);

        polygons[index - 1]->formF += f;
      }
    }
  }
  // Return the maximum patch
  return maxP;
}
```

With the hemicube the cornell box looks like this:



Part 5 - Answers to Questions

What is the effect of changing the subdivision level of the light source?
if the level is too low banding or aliasing artifacts appear. This is because the hemicube method is an approximation to calculating the form factor between two patches.

What happens when you change the size of the hemicube? And why does it happen? What can be done in order to avoid this problem?
If the size of the hemicube is to small aliasing will mean that artefacts will be present in the final image. The cause of this is that the projection of the patches down on the hemicube cannot be represented properly if the hemicube's resolution is to low. The mean way to fix this problem is therefore to use the right resolution that produces an image without artefacts but which does not take to long time to evaluate.

What are the advantages and disadvantages of radiosity as compared to ray tracing?
The advantages are:

- viewpoint independent which means that for a static scene it only needs to be calculated once and the viewport can then be moved around where it is only the projection of the geometry that is needed to be calculated. This is probably also an advantage in architectural visualisations.
- Color bleeding. A surface with one color will project that color onto nearby surfaces.

- Soft shadows.

The disadvantages are:

- slower than for example basic ray tracing, and not much research has gone into how it can be optimized.
- Light and shadow leakage can occur if patch boundaries does not coresspond to boundaries of the geometry
- only LD*E paths can be simulated with the radiosity method.
- if only Garraud interpolation is used between patches, mach banding can occur if the patches are large enough.
- insufficient meshing strategies can cause discontinuities in the radiosity across a surface. This can for example be seen on my rendering of the cornell box where the box most in the front does not have shadow in its left lower corner where it should have.