A Brief Guide to Creating a Radiosity Engine

Joseph Eichholz, eichholz@rose-hulman.edu

Ross Program, Summer 2022

This document will guide you, in broad strokes, through the functionality required to build your own radiosty engine. By that, we mean the engine that takes in a scene and then decides what color to paint each face in the scene.

Some code has been provided that does the least interesting details for you; namely reading in geometry, writing out geometry, and displaying results. The code will run in either Matlab or Octave – a Python implementation would be most welcomed!

Creating Scenes

Getting Objects to Draw

One easy-to-parse and common file format for specifying geometry is the .obj file. You can find .obj files that describe objects at a huge number of different online websites. You need to be a little bit careful – our engine can only understand .obj and .mtl files, nothing else. It will not respect any texturing; texturing wont stop our code from running, we will just ignore the texture information so your object may not render as you expect. Finally, you need to make sure that you aren't dealing with overly large models. A fast engine takes much care to write, and often "cheats" a bit. Our engine will not cheat, but will also be much slower. Any scene (thats the sum of all your objects) over 30,000 faces will likely not be practical to light.

Arranging Objects into a Scene

A few .obj files have been provided to the Discord channel for you. You may use them or download your own. Once you have some objects that you like, you'll want to arrange them into a scene. A number of tools can do that, but I had good luck with 3D Builder. 3D Builder is a free app from Miscrosoft. The .obj file format is also used in 3D printing. Any software that lets you make a model for 3D printing should be useable for arranging your objects into a scene. Save your scene as another .obj file.

Specifying a Light Source

We will need to specify which objects are emitting light in our scene. **This is not a part of the .obj standard**. We'll need to specify this information manually by editing the .obj file produced by 3D-Builder. Most software that writes .obj files will group your faces by object. For example, all of the faces that make up a sphere would immediately follow a line which says

```
o <groupname>
```

meaning that those faces are part of the *groupname* object. 3D-Builder uses unhelpful names like `Object.1` and `Object.2`, but other software might do something different. We can indicate that an entire object is emitting light by writing `#light <power level>` on an independent line. The `#` is not a typo! It is not a part of the .obj standard, so we need to start with a `#` sign so that other programs treat it as a comment. All faces from the `#light <power>` line until the next object line will be assumed to be emitting light at the given power level. The light is assumed to be the same color as the color of the object itself. That is, if a face is specified to be green, and the face is emitting, then the face is assumed to be emitting green light.

There is no way to change this. Lighting is assumed to be turned off with the start of the next object, or can be turned off with `#endlight`.

For example, suppose that our .obj file read:

```
o Object.8
f 8 9 10
f 32 24 25
f 16 5 32
.
.
.
```

We can specify that this object will actually be emitting light at power level 200 (a reasonable number to start with) by modifying the file to say

```
o Object.8
#light 200
f 8 9 10
f 32 24 25
f 16 5 32
.
.
.
```

Reading a .obj File

The file `readObjFile.m` will read a .obj file modestly well. Read the documentation for full details,but the basic usage is:

```
[vertexList, faceList, colorList, emissionList,faceGroups]=readObjFile(filename)
```

You can see if you are properly reading in the data using `renderData`, also provided. Once you have read data from the .obj file you can run

```
renderData(vertexList, faceList, colorList)
```

the scene should look like what you saw in the software you used to arrange things.

Viewing .obj Files

You can open a .obj file in 3D-Builder or any similar software, it is a pretty common standard. You can use the provided `readObjFile` and `renderData` code, or you can use any of a number of free viwers online. You have to be a little careful, as online viewers like to add their own lighting; you need a dumb one. The one at: https://3dviewer.net/ is perfectly suitable.

Writing the Lighting Engine

As we saw in lecture, finding the color to draw each face boils down to solving three linear systems of equations. In this section we detail what exactly the linear system of equations to solve is; and give an idea of how to structure your code to compute the system. Leave solving the linear system of equations to whatever software package you are using.

The Linear System

Recall that we are trying to find the intensity of red, green, and blue to color each face of the scene. Define $\tilde{\mathbf{u}}^r$, $\tilde{\mathbf{u}}^g$, and $\tilde{\mathbf{u}}^b$ to be the vector of intensity of red, blue green, respectively. That is, the (r,g,b) code for face j will be $(\tilde{\mathbf{u}}^r_j, \tilde{\mathbf{u}}^g_j, \tilde{\mathbf{u}}^b_j)$. At the end we may need to scale our results to get appropriate visibility, we'll see that in another subsection. First, however, we'll need to find $\mathbf{u}^r$, $\mathbf{u}^g$, and $\mathbf{u}^b$.

We'll use F_j to denote the j^{th} face of the scene, write $\mathbf{n}_j$ for the outward facing normal vector from F_j, and write $\mathbf{e}_j$ for the emission on F_j. Finally, define

$$V_{i,j} = \begin{cases} 1 & \text{if } F_i \text{ is visible from } F_j \\ 0 & \text{otherwise.} \end{cases}$$

We need to solve the linear system of equations

$$(I - C^r)\tilde{\mathbf{u}}^r = \mathbf{e}^r$$

for the intensity of red light on each face, and similar for the green and blue light. In this context $\mathbf{e}^r_j$ is the intensity of red light emission on F_j and $\mathbf{e}^r$ is called the red emission vector.

The matrix C^r is defined as

$$C^r_{ij} = V_{ij}\rho^r_j \frac{A_j}{9} \sum_{k=1}^{3}\sum_{l=1}^{3} S_{ijkl}$$

where

$$S_{ijkl} = \frac{[\mathbf{n}_j \cdot (\mathbf{F}_{ik} - \mathbf{F}_{jl})]\,[\mathbf{n}_i \cdot (\mathbf{F}_{jl} - \mathbf{F}_{ik})]}{\|\mathbf{F}_{ik} - \mathbf{F}_{jl}\|^4}.$$

Above, $\mathbf{n}_j$ is defined to be the outward pointing normal vector from F_j, and $\mathbf{F}_{jl}$ is defined to be vertex l of F_j. The symbol $\cdot$ means vector dot product.

We must take care, however, as if F_i and F_j are neighboring or very near faces, then it is possible that they share vertices and as such it is possible that $\|\mathbf{F}_{ik} - \mathbf{F}_{jl}\| = 0$. There is an entire branch of numerical analysis that deals with estimating integrals whose integrands are singular. For us, however, we might set

$$S_{ijkl} = 0$$

if $\|\mathbf{F}_{ik} - \mathbf{F}_{jl}\|$ is too small. Too small may be difficult to determine depending upon the software we used to generate the scene and the size of the scene itself. A reasonable rule is to set

$$S_{ijkl} = 0 \text{ if } \|\mathbf{F}_{ik} - \mathbf{F}_{jl}\| < 0.005 \text{ the radius of the scene.}$$

Notes and Details

The outward pointing normal vector $\mathbf{n}_j$ must be computed correctly. The standard is to specify the scene in such a way that the vertices of each face are listed counterclockwise. In this case the outward pointing normal vector may be safely computed as

$$\mathbf{d}_j = (\mathbf{F}_{j2} - \mathbf{F}_{j1}) \times (\mathbf{F}_{j3} - \mathbf{F}_{j1}),$$

$$\mathbf{n}_j = \frac{\mathbf{d}_j}{\|\mathbf{d}_j\|}.$$

The symbol $\times$ means vector cross product.

The visibility matrix V is time intensive to compute. Many engines use clever tricks to speed up the computation; however, we will use a more brute force approach. The code `computeVisibility(vertexList,faceList)` will compute the matrix V. Unfortunately, the code is slow in Matlab but nearly unusably slow on Octave. I have provided several visibility matrices for your use for the scenes I provided. You can also see the next section for alternatives and improvements that you might make.

Scaling the Color Results

Once you have lit a scene, you might need to scale the resulting color vectors in order for viewers to display properly. In particular, you will need to either scale your results to be in the range $(0, 1)$ or $(0, 256)$. Finally, you'll need to scale the percieved brightness of your emitting faces. If you do not scale the brightness of the emitting faces, you might find that the emitting faces totally overwhelm the rest of the surfaces. Here is a reasonable rule. Let F_J be the set of all faces which are emitting, and let F_K be the set of all faces which are not emitting. Let $C_J = \max\{\tilde{\mathbf{u}}_j^r, \tilde{\mathbf{u}}_j^g, \tilde{\mathbf{u}}_j^b | F_j \in F_J\}$ that is, F_J is the largest color of any emitting face. Similarly, let $D_K = \max\{\tilde{\mathbf{u}}_k^r, \tilde{\mathbf{u}}_k^g, \tilde{\mathbf{u}}_k^b | F_k \in G_K\}$ be the largest color intensity on any unlit face. In order to get unlit and lit faces back into the same scale we set

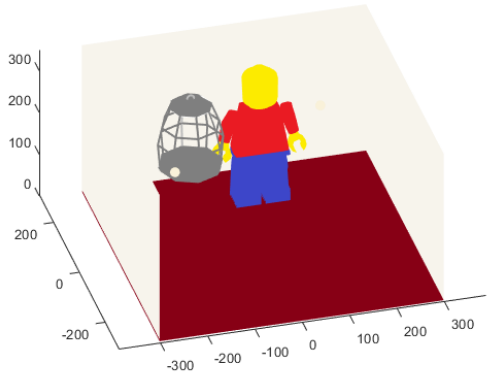
$$\tilde{\mathbf{u}}_i^{r\ scaled} = \begin{cases} 0.9\tilde{\mathbf{u}}_i^r / C_J & \text{if } F_i \in F_J \\ 0.8\tilde{\mathbf{u}}_i^r / D_K & \text{if } F_i \in G_K \end{cases}$$

This should eliminate the problem of emitting surface overwhelming the unlit surfaces. This also matches with human visual perception; past a certain point our eyes cannot tell if we increase the emission rate of a surface. Our eyes just say "Wow, that's bright!", and leave it at that.

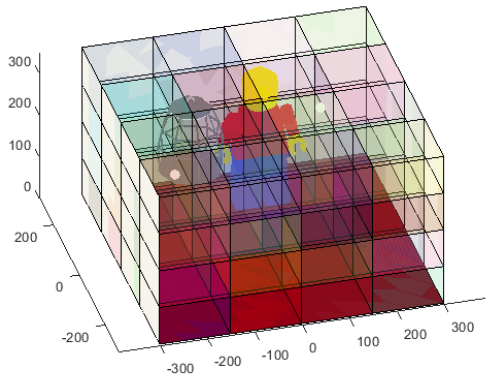
Computing the Visibility Matrix V.

Watch out! This code takes a long time to run! The natural approach to this problem is to take every pair F_i, F_j, and then see if any face F_k occludes the view from F_i to F_j by checking to see if a line drawn from F_i to F_j intersects the face F_k. In the worst case this approach will make M^3 comparisons if there are M faces in the scene. If M is even modestly large this is quickly untenable.

The code in `computeVisibility` tries to take a more sensible approach. Given a scene, for instance the one below:

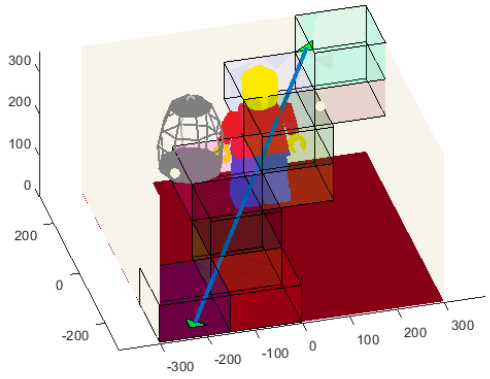


we first divide space into cubes such that no cube contains more than a preset number of faces. We may well end up with cubes that look like these:



Critically, we pre-compute which cubes neighbor each other only once when we create the cubes. With each cube we create a list of faces contained within the cube.

Now, we still consider each pair of faces F_i and F_j. However, for each pair of faces we create a plan of cubes to inspect – these are the cubes that could possibly contain faces which occlude the view from F_i to F_j. Because we know in which cube F_i is, in which cube F_j is, and which cubes neighbor each other, it is relatively quick (though non-trivial) to compute each plan. An illustration of a plan is:



We can see that by only checking faces that are members of the indicated cells we will avoid a significant amount of work. The trade-off is that for each pair of faces we must now create a plan of cubes to inspect; balancing those will be different on each scene.

Despite the algorithmic improvements in this code, it is still fairly slow in Matlab. On Octave, because the interpreter is not nearly as optimized as that of Matlab, the code is pretty well unusable. I have provided several visiblity matrices for you to use as you wish; you may load them with the `load` command.

For those that have experience compiling code, I included `computeVisibility.c`. That code is much faster, but you'll need to compile the C code yourself and then run it. To compile:

```
gcc computeVisibility.c -o computeVisibility -lm -fopenmp
```

Once compiled:

```
./computeVisibility inputFile.obj outputFile.vis
```

should do the trick just fine.

Possible Improvements to Computing Visiblity

Without getting really fancy, I can see a few improvements that the very interested student might attempt in order to speed up the visibility code.

First, we might run the code in parallel. Computing V_{ij} is completely independent from computing V_{ik}, and so it would be trivial to compute them at the same time. In Matlab the Parallel Computing Toolbox is needed, but then a simple `parfor` command would do the trick. Octave has similar parallel computing abilities, but the code rewrite would be slightly larger.

Second, we might use a hierarchical approach. As you can see in the figure above, there are large swaths of open space. It doesn't make sense to make lots of cubes to measure space that is empty. We might try a similar approach but with an *octtree* instead. You can read about quadtrees, the two-dimensional counterpart to octtrees, here: https://en.wikipedia.org/wiki/Quadtree.

A larger step would be to rewrite all of this code in a faster language. Even Python, not normally considered a fast langauge, would likely outperform Matlab and Octave on this problem. Naturally, C would be a faster choice, but something like Rust migth be a good compromise. Combining C with an octtree and parallel computing would likely leave us with a very reasonable algorithm.

A good initial move would be to write just the visibility calculation in another language. It could read the .obj file, compute visibility, and then write a 0/1 text file as output. Then you could have Matlab read the text file and proceed. Matlab and Octave are actually quite fast at the remaining steps of the lighting computation.

As a final improvement, one might try a 3D version of Bresenham's line algorithm:
https://en.wikipedia.org/wiki/Bresenham's_line_algorithm.

Writing and Viewing the Results

Once you have solved for $\tilde{\mathbf{u}}^{r\,scaled}$, $\tilde{\mathbf{u}}^{g\,scaled}$, and $\tilde{\mathbf{u}}^{b\,scaled}$, congratulations! You have completed lighting! In order to write your results to file you just need to call `writeObjFile`. Example useage is:

```
writeObjFile('myrenderedfile.obj',vertexList,faceList,[ured,ugreen,ublue]);
```

Conclusion

You now have all the detail required to do radiosity rendering. If you decide to undertake the project please let me know! My email address is listed above. If you have any questions about the provided code, decide to translate it into another language, decide to improve the visibility computation, etc., please feel free to get in touch; I'd love to hear about it!