Radiosity

Key Techniques/Algorithms

Our implementation of Radiosity has 3 key steps: tessellating the loaded scene, computing the form factors and intersections for each triangle, and computing the lighting for each pass over the scene. We had major constraints on hardware given we were programming this at home, so we decided to create heuristics and optimizations to improve our runtime; however, our solution follows the radiosity solution discussed in class.

To start, we needed to create a list of polygons, in our case, triangles, and tessellate them. We tessellated our triangle into 4 triangles in the shape of a triforce by taking the midpoints of each side of the triangle and connecting them, and recursively doing this until our desired tessellation level. Each of the triangles would have the corresponding diffuse constant from the original face.

After tessellating the triangles, we calculated the form factors between the triangles. We used the following formula: $F_{ij} = \cos(\Theta_i)\cos(\Theta_j) / \pi d^2$, where d is the magnitude of the vector from one triangle's centroid to the other, Θ_i and Θ_j are the angles between the normals of the corresponding patches and the vector from one centroid to another. The form factor was also adjusted based on occlusion; if pairs of patches were occluded by other triangles, then the form factor would be set to 0. We used a BVH with all of the patches as elements to determine occlusion.

The lighting was done via doing direct illumination and indirect illumination passes. For each patch, we keep track of the residual light R_i and the accumulated light L_i. In the direct illumination passes, for each light source, we used the following formula to determine how much light the patch received: B_i = A(d) * light intensity * D_i * visibility factor * $\cos(\Theta_i)$. To calculate the visibility factor, we divide the receiving triangle using 1 level of Triforce tessellation, and sample the centroids of the 4 sub-triangles to see if they are visible. The factor is calculated as # of sampled points / 4; the BVH is also used to determine intersection here. The function A accounts for distance attenuation, D_i is the diffuse constant for patch i, and Θ_i is the angle between the patch normal and the vector from the light source to the centroid. The B_i of each patch becomes the current residual and accumulated light for each patch. For the indirect illumination passes, we have each patch shoot it's light to each of the other patches, where the light transferred is just $B_j = F_{ij} * R_i$ using the residual light from patch i and transferring it to patch j. The residual light and accumulated light of each patch j are incremented by B_j, and the residual light of the accumulated patch is set to 0.

The accumulated light of each patch becomes the color when each triangle is drawn in OpenGL.

Implementation Details

We made significant changes to our radiosity implementation in order to get it's speed fast enough to test how it looks, since even at the second level of tessellation it would take minutes. For starters, we stored form factors in a hashmap with the triangle index as the key and the factor was the value for all visible triangles. This would significantly reduce space needed vs using a matrix, and allowed us to compute visibility at the same time. Our visibility was calculated naively, via shooting a ray from each triangle pair, to each other triangle, and calculating ray intersection, a costly $O(N^3)$. In order to better negate this, we reimplemented our BVH from our ray tracer, which improved runtime to $O(N^2 log(N))$. Both of these functions can be seen implemented in the pre_process function in RadiosityScene.h, with the actual data structure being a Map(Key: Visible Triangle Index, Value: Form Factor).

For further speed improvements, we took another note from our ray-tracer project and implemented threading. The way we did this is we split up the number of triangles each thread processed in every method that needed to do an N^2 or greater operation. This means that doing our preprocessing for form factors and visibility, and doing our light passes would all be threaded, all of which were major bottlenecks.

The last thing we did to significantly increase our code's speed was implement form factor culling. Essentially, any form factor smaller than a threshold would not check for ray triangle intersection, significantly increasing the speed of the preprocessing, and also when we do passes using our lights, there are less triangles to calculate light for. There is a major trade off for this, which is accuracy of the image. The light bounce and soft shadowing is significantly less with fewer form factors, but we attempted to account for it by tweaking our constants while attenuating and other places.

One thing not discussed so far is how we did light passes. Essentially, we loop through all of our lights, and call the function calculate_light. Each light source has a variable signifying if it's been used, if it hasn't, then we illuminate all of our patches using the light source. If the light source has been used, we perform a secondary operation in which we shoot light from each patch, to each other patch, and calculate the amount of light to be accumulated to the patch using our form factors and various constants. Once that finishes running, we add the accumulated light to a variable called diffused_light.

All of these functions discussed thus far are called in main if seeing their calls is desired, and most of them are implemented in RadiosityScene.h. There are several limitations to our solution that include accuracy of the solution depending on how much speed you want to give up, and we currently are facing a "sawtooth" effect, where our adjacent triangles don't blend smoothly. Triangle face counts after tessellating past 10000 take undesirable levels of time on a non dedicated server as well, a big issue that ideally could be resolved by faster ray-triangle intersection implementations.