Jackson Barker
007749531
COMP 4490
Ray Tracer Project

Ray Tracer Project

Feature 1: Acceleration

I chose a Bounding Volume Hierarchy as my acceleration structure.

How it works

To create the bounding volumes, after the vertex data is read in from the JSON file, createBoundingVolumes() will find the minimum and maximum for each of the x, y, and z values and store these in a BVHnode struct attached to the mesh object. It will then subdivide the triangles of the BVHnode into 2 halves and continue on recursively until a single triangle is found, at which point a leaf node is created. The structure is a binary tree of BVHnodes.

Any ray that needs to check meshes will use traverseBoundingVolumes() to find the appropriate triangle. If 2 triangles bounding boxes are hit (like when 2 triangles make a square, thus having the same bounding box) then the point-in-triangle test is performed to see which one it actually hit.

Strengths

The BVH had a great impact on the rendering time for scenes with lots of meshes (or few meshes with lots of triangles.) For example, in scene i.JSON with the checkered floor, the render took ~3:43 in the original ray tracer. When the BVH was implemented, that same scene took just 17 seconds, ~7% of the original time.

Weaknesses

The structure does not explicitly group triangles based on proximity but rather makes the assumption that triangles listed in the JSON files are listed in a reasonable order. For the given scenes c-g this does seem to be the case but if a file were given with ‘random’ order to the triangles, the structure would have similar performance to looping through all the triangles like before.

Feature 2: Transformations

I chose to do mesh transformations.

How it works

After a mesh's triangles are read into the Object struct, transformMesh() is called to transform the vertex data. First, a center of mass is calculated for the mesh to give a point to rotate around. This is calculated as the average of all of the barycenters of all of its triangles. The function calculates translation, rotation, and scale matrices and multiplies each point by their product. This is done before the bounding volumes are created so no extra work needs to be done there. (see readme for exact transformations used.)

Strengths

Will translate, rotate, and scale correctly and is capable of non-uniform scaling as well. Since this transformation is done directly to the vertex data, and the bounding volumes are created based on that transformed data, rays can behave as they were before and do not need to be transformed to test meshes.

Weaknesses

Transformations are hard-coded so changes between scenes are done by modifying values in the code.

Feature 3: Improved Quality

I chose to do Ward anisotropic distribution and Oren-Nayar diffuse reflectance.

How they work

Oren-Nayar is done by checking if the object has a roughness component (a boolean) and branches from regular Phong diffuse calculations in calcLight() if so. Ward anisotropic is done in a similar way.

Strengths

Regardless of roughness value, Oren-Nayar gives a softer, better looking diffuse component. Ward changes the shape of the highlight from round to something else based on the vectors additional involved.

Weaknesses

For Oren-Nayar diffuse calculations, the roughness value of the object doesn't seem to have as large of an impact on the result as it should.

Ward highlights look a little strange. While I have confidence in my algorithm for Ward anisotropy, I believe this comes from an insufficient understanding of how to correctly obtain additional vectors involved for the 'direction' of the anisotropy.

Feature 5: Techniques

I chose to do distribution ray tracing to achieve soft shadows with area lights and glossy reflections.

How they work

For soft shadows, area lights are used. Area lights in my implementation are essentially point lights with an additional 'radius' component. Shadow testing now casts a number of rays (64 works well) to random points within the light. Points are determined by adding random values [-radius, radius] to the position of the light. The vectors are either (1,1,1) for lit or (0,0,0) for in-shadow and these are averaged into a percentage of the light that the point receives. Transmission is accounted for and will return the percentage of light that gets through for a single ray ((1,1,1) * transmissive). Done in calcLight() and calcAreaLightPercentage().

For glossy reflections, the mirror reflection ray r is calculated as normal. That is then used to generate slightly 'perturbed' rays and the reflections in those directions are calculated. A "gloss" component of the object is used to activate the glossy reflection code and determine the 'sharpness' of the reflection (a higher gloss results in more blur, gloss of 0 results in perfect mirror reflection.) These results are averaged and given as the output for the reflective component of the final colour output. Done in calcLighting() and calcGlossyReflection() which makes calls to the original calcReflection().

Strengths

Area lights produce nice, soft gradients for shadows instead of hard lines produced by other lights.

Glossy reflections work well and the "gloss" component of objects affects the degree of glossiness correctly.

Weaknesses

For this particular implementation, soft shadows are only generated for lights of type "area" and thus, other types of lights will still produce hard shadows.

Glossy reflections take far longer to calculate than soft shadows and so fewer rays are used for them (~8 vs. 64 for soft shadows) to prevent it from taking unreasonably long to render. Even with higher samples (16), there is still a slightly speckled appearance. The amount of gloss seems to have a drastic effect on the render time despite the same number of samples being taken. (see readme for runtimes of different combinations.)

References

1. Soft shadows/area lights and glossy reflection:
    a. http://web.cse.ohio-state.edu/~shen.94/681/Site/Slides_files/drt.pdf
    b. http://ray-tracing-conept.blogspot.com/2015/01/reflection-and-glossy-reflection.html
2. Oren-Nayar diffuse:
    a. https://lonalwah.wordpress.com/2013/11/29/oren-nayar-reflectance-diffuse-getting-rough
    https://en.wikipedia.org/wiki/Oren%E2%80%93Nayar_reflectance_model
3. Ray-box intersection:
    a. https://www.cs.cornell.edu/courses/cs4620/2013fa/lectures/03raytracing1.pdf
    b. https://developer.arm.com/docs/100140/0302/advanced-graphics-techniques/implementing-reflections-with-a-local-cubemap/ray-box-intersection-algorithm