TEAM MEMBERS:
Shenghan Chen, Jincheng Xu, Siyang Wang

Q1: Approximately how many hours it took you to finish this assignment: 20+ hours

Q2: Your overall impression of the assignment. Did you love it, hate it, or were you neutral? One word answers are fine, but if you have any suggestions for the future let me know.
I really loved it, this assignment has helped me understand many concept that I overlooked during the lecture. I think it is very comprehensive -- the line intersecting face, using recursion/loop to find images of sources, and use of scene graph, transformation to world coordinate, sounds and impulse response handling -- it contains many interesting facets, which otherwise we would not be able to appreciate and comprehend completely had we not do this assignment. Also, this is one of the project cool to show around since it is very presentable. The fact that we can use web GUI and inspect element to test is very helpful, as it provides Integrated Development Environment usually not seen in scripting/loosely bounded language coding.

Core Technique

Image Source Generation:
Image Sources are found by transforming the face normals into world coordinates and solving as ray-polygon intersection. In particular, all the faces are traversed only once using a recursive function probing down the scene file hierarchy and stored in an array as a field of scene to save computational time.

Path Extraction:
In rayIntersectPolygon(), the interior point test is carried out by computing the cross product of a directed edge with a vector from one of its vertex to the intersection point, then computing its dot products with all the other such cross products to see if the results are all positive (so that all the cross products are in the same direction, indicating an interior point). In particular, the dot products can't be zero, otherwise the intersection takes place at an edge, which would cause problems when one of the source and the receiver is in a box and the other is outside, because the path might unexpectedly enter/exit the box through the edge. We suppose this to be a (not so interesting) special case.

For path extraction, we start from the receiver to each image source in scene.imsources and then continue tracing the ray from the intersection point to the image source's parent, recursing all the way until it's the source.

We tackled the two **special cases** that you mentioned: first, in case the source is in a box and the receiver is outside, we make sure that even if the ray from the last intersection point to the source intersects a face, the length is longer than the distance between them; second, by starting tracing the ray from the receiver, the ray should always hit the generating face first, regardless of what's between the intersection point and the image source.

There are several special cases that we noticed but ignored anyway (which we believe is the expected way to handle them in this project) because they are either trivial or impractical to address. In particular, we discard a path when there's image source sitting on its generating face.

Impulse Response Generation:

In this section, we compute the impulse response of the sound traveled through different paths. We do this by first calculating the time it takes for the sound to travel through each routes, and calculating the intensities heard by the receiver for each path. The intensities decays can be obtained by multiplying the attenuation/decay using rcoeff at each faces the path hits, as well as 1/((1+distance traveled)^p).

Since in path, the first element is receiver, and the last element is the source, two of which without the rcoeff. To handle this special case, we calculate the distance of last path segment first before we enter into the for loop.

Once we have the time required for each path as well as the decays, we have the impulse response. We stored information in array, the size of which is the longest time of path (s) * sample rate (44100/s). We then store each impulse response on its corresponding index, which in this case is (each time for arrival) * (44100). When the index falls in between bins, we used a nearest index technique, basically assigning the impulse to the new index where the original index is closest to.

Scene File Submissions

SceneEllipsoidal:

in this scene file, receiver and souce are placed inside a ellipsoid with eq. x^2/50+y^/40+z^2/40=1. And they are initially placed at two foci of the ellipsoid. Then we and listen to the convolution, the received sound is relatively clear. Next, I move the receive a little bit, and compute and listen to the convolution again. This time the received sound is much weaker. This effect is due to the property of foci. If you emit rays in all direction of light from one of the foci, all of the reflected rays will intersect at the other. That's why we hear clear sound in the first case.

HierarchicalRotation:

In this scene file, I first rotate the table (as in table.scn) for arc cos(3/5) degree around y-axis, then put it on a floor.

Reverberation Effect:

This scene file uses the same setting as SceneEllipsodal, except that the reflection coefficient is 0.1 instead of 0.9. The convolution using the same sound is weaker than SceneEllipsoidal.

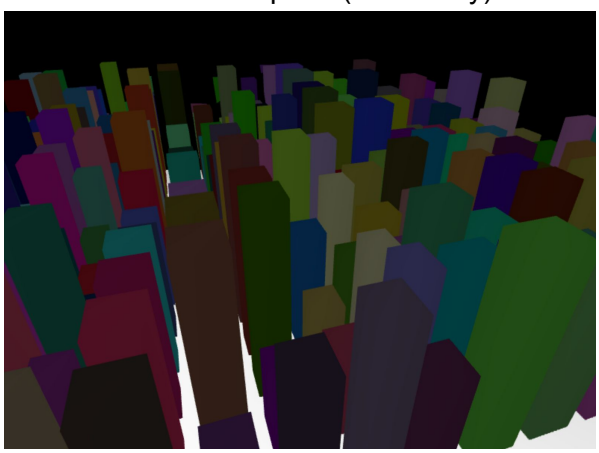
Near Versus Far:

For this question, we created a function which builds a random “city” on a floor of size 500*500. We just need to change the value of the scale factor “m” (explained below), thus creating the effect of “near” and “far”. To shorten the computing time, we only calculate convolution when order is 1. When m is 1, the sound of the convolution is noticeably weaker

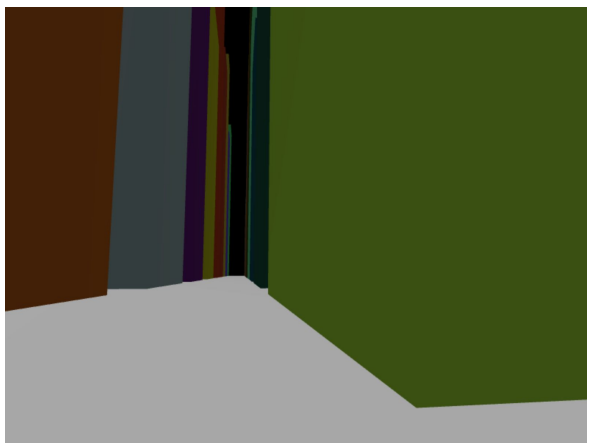
than the original sound. When m is 0.01, however, the sound of the convolution is almost the same as the original sound. This makes sense because with less distance travelled, the sound lost will be less. So the received sound will be louder.

City Environment:
Inspired by Chris' suggestion, I wrote a function to generate a city scene with randomly sized streets and buildings and randomly placed receiver and source. Click the very top button in index.html to build a random city. Some parameters are set as local varibles in the function, such as dimension of the city, ranges of building size and height, and range of street width. Note there is a scale factor m used to scale the entire city, so for example setting it to 0.01 gives a 1:100 miniature. By default m=1 so that the city is "to scale" (i.e. all the dimensions and sizes are in meters). I set the external viewpoint to be from the highest point of the city, which might makes the city looks smaller, but it actually is to scale (and you might want to change the keyboard navigation speed so it doesn't move painfully slow at 2.5m/s). Therefore there isn't a physical scene file, but I hope that could be tolarated since I'm doing (at least) the same work, and doing it this way feels cooler and hopefully gives a more beautiful city. (The reason that I hardcoded the parameters in the function is that I feel like spending too much time messing with input boxes and buttons isn't the purpose of this project.) Also worth mentioning is that a really to-scale city scene doesn't lead to satisfactory results in terms of finding paths within a relatively reasonable order. For order > 2 the computational part takes so long that our program crashes almost every time, so I didn't set the city dimension to be greater than 500. (For visual purposes only, d = 1000 is also good.) For a path to be present in a low order, I have to limit the receiver and the source within a distance of at most one block, so they are either "across the street" or "around the corner", otherwise there's no reachable path between them. I also have to make the streets much wider than real scale, for a similar reason.

External viewpoint (above city)



Source/Receiver viewpoint (on the street)



Art Contest:
Since our city scene doesn't have a physical scene file, I wonder if we could ender my "random city generator" function into the art contest.

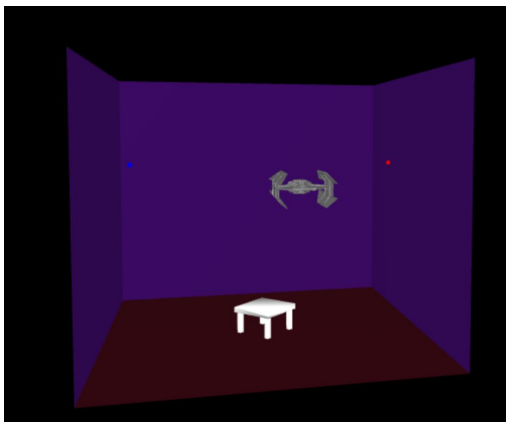
Computational Improvements

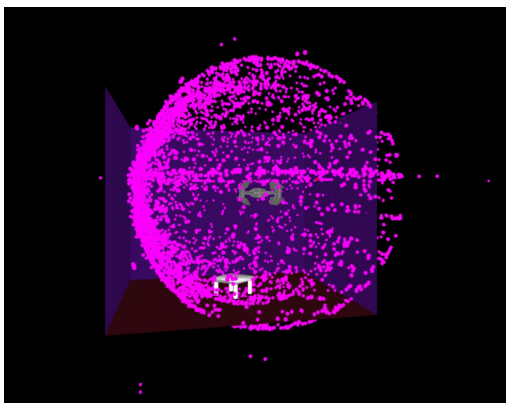
Scene Graph Bounding Box Pruning:
For simplified.scn with 1600+ faces, the console logged that it takes 0.373s to compute bounding boxes and extract paths of order 1, while it takes 9.322s to extract paths without using bounding boxes.

Bounding Boxes are built recursively by finding the ranges of x, y, z coordinates of all the vertices in a node. (In cases where one of the min/max pairs is equal, indicating the node to be within a single plane, we just build a "bounding plane" to save a little bit of computational time.) Also note this feature can be turned off by setting the enableBBox variable to false, which might be helpful to build the city scene since all the hierarchy in it is a dummy transformation node controlling the scale, therefore using bounding boxes simply doubles the computational time. Bounding Boxes are computed only once for each scene and stored in an array as a field of each node to save computational time.

Simplified Meshes:
We created simplified.scn with 15733 faces, while its simplified mesh contains 1573 faces. The console logged that it takes 0.373s to extract paths of order 1, while it takes 47.622s to extract paths (both with bounding box enabled and including time to compute bounding boxes, otherwise the computational time is way too long).

Rendering effect





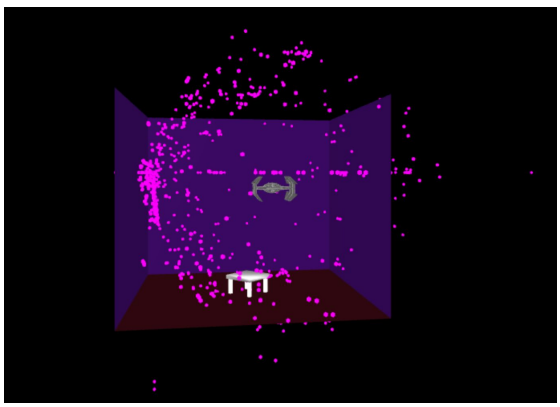


Image sources without simplification Image sources with simplified mesh

Other

GUI Augmentation / Debugging Tools:
Two visual debugging features are developped: one is showing the light blue bounding boxes (edges only) and the other is a short yellow normal coming out of each reflection face, which actually is a combination of two suggested debugging features - drawing intersection points and face normals.