Accelerated Beam Tracing Algorithm and Its Implementation in Matlab

MUMT 618 Project Report

Harish Venkatesan

1 Introduction

Various acoustic phenomena take place within a room [3]; the surfaces in a room reflect sound specularly, where the angle of incidence equals the angle of reflection (when the surfaces are smooth and irregularities are much smaller compared to the wavelength of the sound), and diffusely, where the sound is scattered in several directions (due to irregularities in surfaces of the order of magnitude of the wavelength), absorb sound and dissipate them as heat and also transmit a small percentage of energy through them. At edges where two surfaces meet, acoustic waves of wavelengths larger than the discontinuity undergo diffraction, where the waves appear to bend around the edge. These phenomena can be modelled using two different classes of techniques; numerical acoustics and geometrical acoustics [2]. Numerical acoustic techniques involve solving the wave equation at sampled locations in the room (ex. Finite-Difference-Time-Domain, Boundary-Element-Modelling, etc.) which are computationally very expensive to be performed at interactive rates. Geometrical acoustic techniques, on the other hand, assume rectilinear propagation of waves and provide an approximation of wave propagation at reduced computational costs [3]. Several algorithms that show great potential for real-time computation with high degrees of approximation have been presented in literature. One such algorithm is the focus of this project.

Out of the many acoustic phenomena that take place within a room, specular reflections are the most perceptually significant in localization of sounds, which occur in the early part of the room response [1]. The image source technique (figure 1) is fairly straightforward as it involves mirroring the source successively about all reflecting surfaces within the room until a predetermined order of reflection is reached. Though the algorithm is simple and can be extended to complex rooms in theory, the computational complexity and the memory requirement increases exponentially with increasing reflection order, hence limiting its application to low orders of reflection and simple geometrical structures.

Ray tracing is a Monte Carlo method that involves projecting a large number of rays from the source with a pre-defined distribution and tracking them as they pass through the listener until they have travelled a fixed distance or the energy of the ray falls below certain threshold. Although ray tracing has had great success in graphics rendering, one of its drawbacks in acoustics is that for a point sized listener, the probability of rays passing through is very small which leads to poor performance.

Beam tracing involves tracking of a finite set of beams, instead of individual rays, emitted by the source as they undergo reflections within the room. This technique solves the issue of the point sized listener in ray tracing since the cross-section of a beam increases with increasing distance, which can be seen as the volume of the listener increasing as a function of distance from the source, thereby

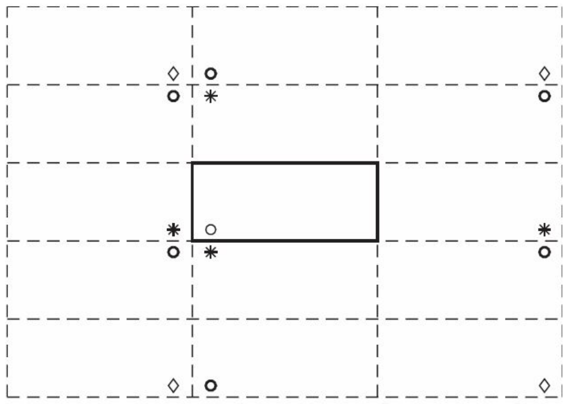


Figure 1: An example showing image source method for a shoe box shaped room in 2 dimensions [3]

increasing the probability that a ray passes through the listener. Beam tracing can also be viewed as an optimization of the image source technique, where it produces narrower image source trees by omitting the computation of irrelevant image sources [3].

In [1], an accelerated beam tracing algorithm that computes beam trees and performs path validation at interactive rates is presented. Two optimization techniques are mentioned, the fail plane and the skip sphere techniques, that help accelerate the path validation step resulting in a two-fold overall performance improvement. In this project, a detailed study of the functioning of the algorithm is presented along with a demo of the algorithm on Matlab with wrappers and GUI written for the C++ implementation of the algorithm found on Samuli Laine's website[1]. Since the source files did not contain adequate comments, the analysis presented in this paper was majorly done through manual code tracing.

The structure of this report is as follows; section 2 presents the precalculation step and the beam tree construction, section 3 presents the path validation step along with the two optimizations incorporated, section 4 presents the Matlab implementation and finally, section 4 presents concluding remarks.

2 Precalculation

2.1 Building the Binary Space Partitioning Tree

A room is defined by all the surfaces that make up the room. All curved surfaces of the given room are first tessellated into piece-wise flat surfaces and a list containing all the polygons of the room is made. With the set of polygons defining the room, a binary space partitioning (BSP) tree of the room is built before the actual beam tracing part. The BSP is built as follows:

Step 1: Find the axis-aligned bounding box (AABB) enclosing all polygons defining the room.

Step 2: Find the optimal split axis and the best split position of the parent bounding box such that the difference in the number of polygons on either sides of the split plane is minimized.

[1]The source code for the EVERT library can be found in `https://users.aalto.fi/~laines9/publications/laine2009aa_code.zip`

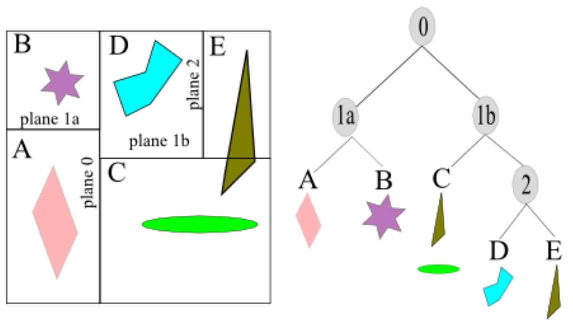


Figure 2: An example axis aligned BSP tree in 2 dimensions.

Step 3: Put all polygons fully or partially on the left side of the split plane in the left child node whose new AABB is covered on the right side by the split plane and go to step 2. Continue to step 4 if no further split is possible.

Step 4: Put all polygons fully or partially on the right side of the split plane in the right child node whose new AABB is covered on the left side by the split plane and go to step 2. Continue until no further split is possible.

The resulting tree structure is a balanced Axis-Aligned Binary Space Partitioning tree. This tree is made balanced in order to improve the efficiency of traversal. Figure 2[2] shows an example of such a tree with five polygons in a 2D space. The BSP tree computation is done once for the room unless there is a change in the room geometry.

2.2 Beam Tree Construction

The beam tracing algorithm involves building a tree structure that tracks the path of the beams cast from the source up to a required order of reflection. This step does not take into consideration the position of the listener and solely depends on the position of the source.

Beam tree construction is also considered a precomputation step that is processor intensive and is done once for a given source position and room geometry. Every time either of them change, a new beam tree must be computed. Hence, the computational cost of the beam tree construction restricts movement of the source, i.e., this algorithm is not fully dynamic. Nevertheless, the authors of [1] claim that this computation is sufficiently quick for small and simple rooms and can be used to compute image sources at interactive rates.

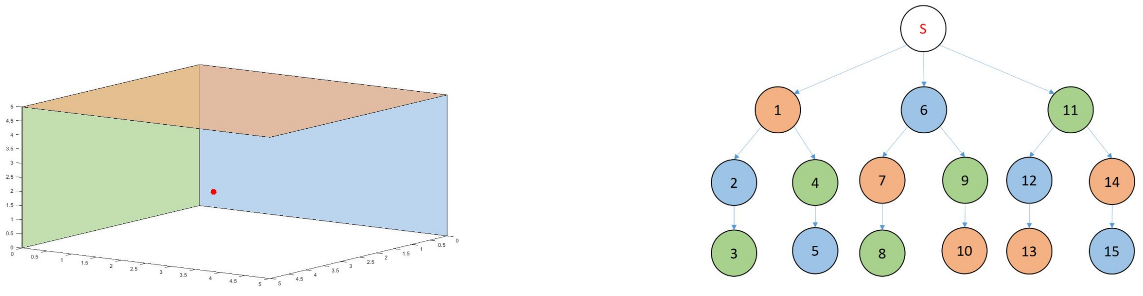
The beam tree is computed as follows:

Step 1: Find the first order image sources from all the surfaces in the room by reflecting the source about all the surfaces. Populate the first level of the tree with the surfaces with parent id=-1.

Step 2: For each image source, cast a beam with the bounding surfaces defined by the image source and the edges of the reflecting polygon.

Step 3: For each beam, find all the intersecting AABBs by traversing the room BSP tree and performing frustum culling. Frustum culling is a common technique used in computer graphics to identify objects within the viewing frustum.

[2]Spatial Data Structures: `https://www.bogotobogo.com/Games/spatialdatastructure.php`



(a) (b)

Figure 3: (a) shows an example with three surfaces and a source (red dot) and (b) is the beam tree of the constructed for the given source position. The numbers in the tree nodes indicate the order in which the nodes are added to the tree and the colours of the nodes indicate the reflecting polygons.

Step 4: Identify those polygons in the intersecting AABB other than the parent polygon itself, that have a non-negligible surface area intersecting the beam.

Step 5: Populate the next level of the tree with the newly found polygons along with their parent IDs.

Step 6: Find the image sources obtained from the newly found polygons and go to step 2.

Step 7: Continue until the required order of reflections is reached or there are no further reflections.

Figure 3 shows an example of a beam tree constructed for a room with three surfaces. Visibility calculations are not performed during the beam tree construction and are done during the reverse ray-tracing path validation step since the improvement in performance outweighs the increase in memory requirement. This beam tree is then used in the path validation step which is discussed in section 3.

3 Path Validation and Proposed Optimizations

The path validation step involves a quick ray tracing from the listener to the source through the nodes of the beam tree to indentify unobstructed paths. Starting at any node in the previously constructed beam tree, the ray tracing is performed as follows.

Step 1: Set the listener as the target and the image source for the current node as the source.

Step 2: Construct a ray between the source and the target and verify if:

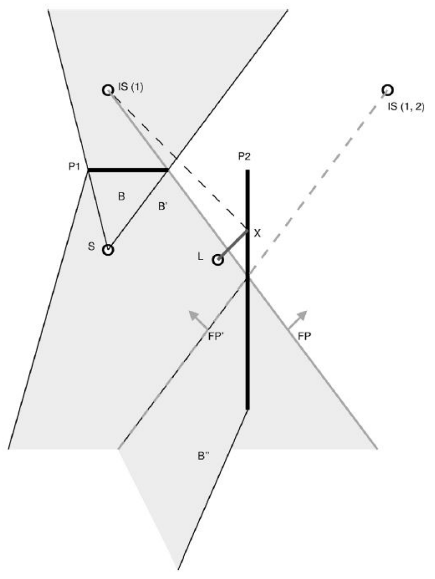
(a) The source and the listener are on either sides of the reflecting surface,

(b) The ray between the source and the target passes through the reflecting surface.

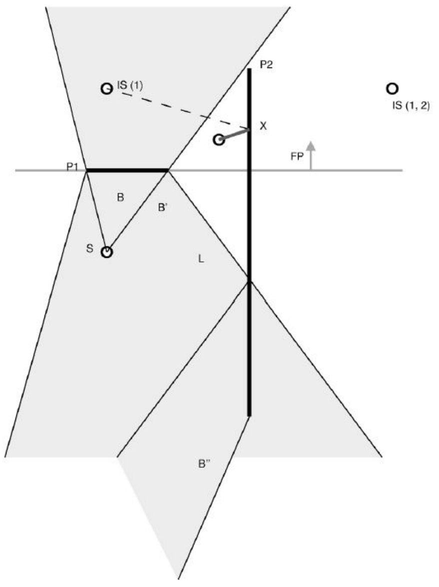
Step 3: If the above conditions satisfy, set the point of intersection between the ray and the reflecting plane as the target and the image source of the parent node as the source. If the conditions fail, reject the path.

Step 4: Go to step 2. Repeat until all the nodes in the branch are traversed up to the root node (the source).

The above steps are performed for all the nodes in the beam tree to validate paths of all reflection orders. Path validation is performed each time there is a change in the listener position. The process



(a)



(b)

Figure 4: In (a), line segment X-IS(1) misses the plane P1. The fail plane FP is propagated to the next level in the branch as FP'. In (b), points X and IS(1) are behind the plane P1 and this plane is propagated to the next level as FP. (Source [1])

of going through the entire tree is a computationally expensive and redundant task. To speed up this process, two optimizations are proposed in [1]; the fail plane optimization and the skip-sphere optimization.

3.1 Fail Plane Optimization

In step 2, if a direct path between two nodes of a branch is invalid, paths to the source starting at nodes at lower levels on the same branch would also be invalid and validating these paths would be unnecessary. In order to avoid extra computation, a fail plane is found where the path validation fails.

If condition in step 2b fails, the fail plane is that plane of the beam originating from the image source, whose perpendicular distance to the target is the smallest (see figure 4a). In step 2a, the reflecting surface itself is the fail plane (see figure 4b). In both these cases, the path would be valid if the target was behind of the fail plane instead of being in front of it.

This fail plane is propagated down the branch to all the nodes by mirroring the plane at each node by the reflecting surface. Thus, before validating a path to the source, it would be sufficient to just check if the listener is in front of the fail plane to reject the path. Further explanation of this optimization is given in [1].

3.2 Skip Sphere Optimization

This optimization technique involves grouping of neighbouring nodes of the beam tree into buckets of a specified size. For each bucket, a sphere centred at the listener location with radius equal to

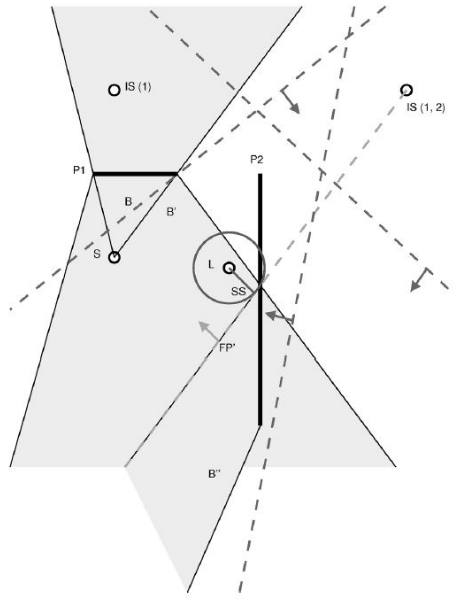


Figure 5: Fail planes for the nodes of the bucket are shown in dotted lines. The skip sphere is centered at the listener location with a radius equal to the distance between L and fail plane FP. As long as the listener L is inside the skip sphere SS, there would be no need to validate paths in the bucket. (Source [1])

its distance from the nearest fail plane is defined. As long as the listner is within this sphere, paths from all the nodes in the bucket would be invalid and can be straightaway rejected. If the listener moves out of the skip sphere, it can be guaranteed that at least one path in the bucket has changed. Figure 5 illustrates this with an example.

The two optimization techniques drastically improve the performance of the algorithm by reducing the number of path validations, there by making it possible to compute reflection paths at interactive rates. [1] provides results of comprehensive timing tests conducted for rooms of increasing number of surfaces and various levels of complexity.

4 Matlab Wrappers and GUI for the EVERT library

The extensive memory management and object oriented programming capabilites offered by C++ make it the best choice for the implementation of such an algorithm with strict timing and performance constraints. The EVERT library[3] was originally written in C++ by Samuli Laine and later maintained by Markus Noisternig. Reimplementing the algorithm on another programming language would be an arduous task and may not guarantee optimum performance. However, since C++ does not provide easy to use functions for visualization of the algorithm, Matlab wrappers[4] for the C++ classes were written as part of this project to function as a test bed, along with a GUI.

Matlab classes that internally make use of the corresponding C++ classes were written (not all

[3]The source code for the EVERT library can be found in `https://users.aalto.fi/~laines9/publications/laine2009aa_code.zip`

[4]Matlab wrappers for the EVERT library can be found in `https://github.com/HarishJayanth/EVERT_matlab`

functions of the underlying C++ classes are implemented) with [5] as a guide. In a way, the Matlab classes are mirrors of the C++ classes. Since Matlab functions do not return pointers of objects for retaining references, a hash table is maintained within the wrappers written in C++ that contain integer keys for every instantiated object, whose value is the pointer to the object in the memory. These objects can be referenced within Matlab using their key. The objects are instantiated such that the Matlab wrappers and the C++ source have shared access to them for efficient memory management. This way, deleting an object within the Matlab environment would automatically deallocate the memory occupied by the corresponding C++ class object. The wrappers are compiled inside the Matlab environment using the "mex" command. The Git repository4 contains a Matlab script named "libevert$_mex_compile.m''thatcompilesamexfileintotherightdirectory.$

4.1 The GUI

A GUI was built using Matlab's GUI development tool, GUIDE. Using the GUI, one can compute the impulse response (echogram) for a given room, source and listerner positions up to a specified reflection order. Figure 6 shows an example of a concert hall with a source (red dot) and a listener (green dot). The blue lines indicate the reflection paths between the source and the listener up to 3^{rd} order.

Every reflecting surface absorbs sounds differently for different frequency bands and reflects the rest. The "material.dat" file available as a part of the EVERT library contains a list of known materials with absorption coefficients α_{oct_N} given for 10 octave bands. The attenuation that must be applied to each octave band due to reflection is given by $G_{atten_N} = 1 - \alpha_{oct_N}$. The overall gain to be applied for a path would then be cumulatively given by,

$$G^p_{atten_N} = \prod_i G_{i,atten_N}$$

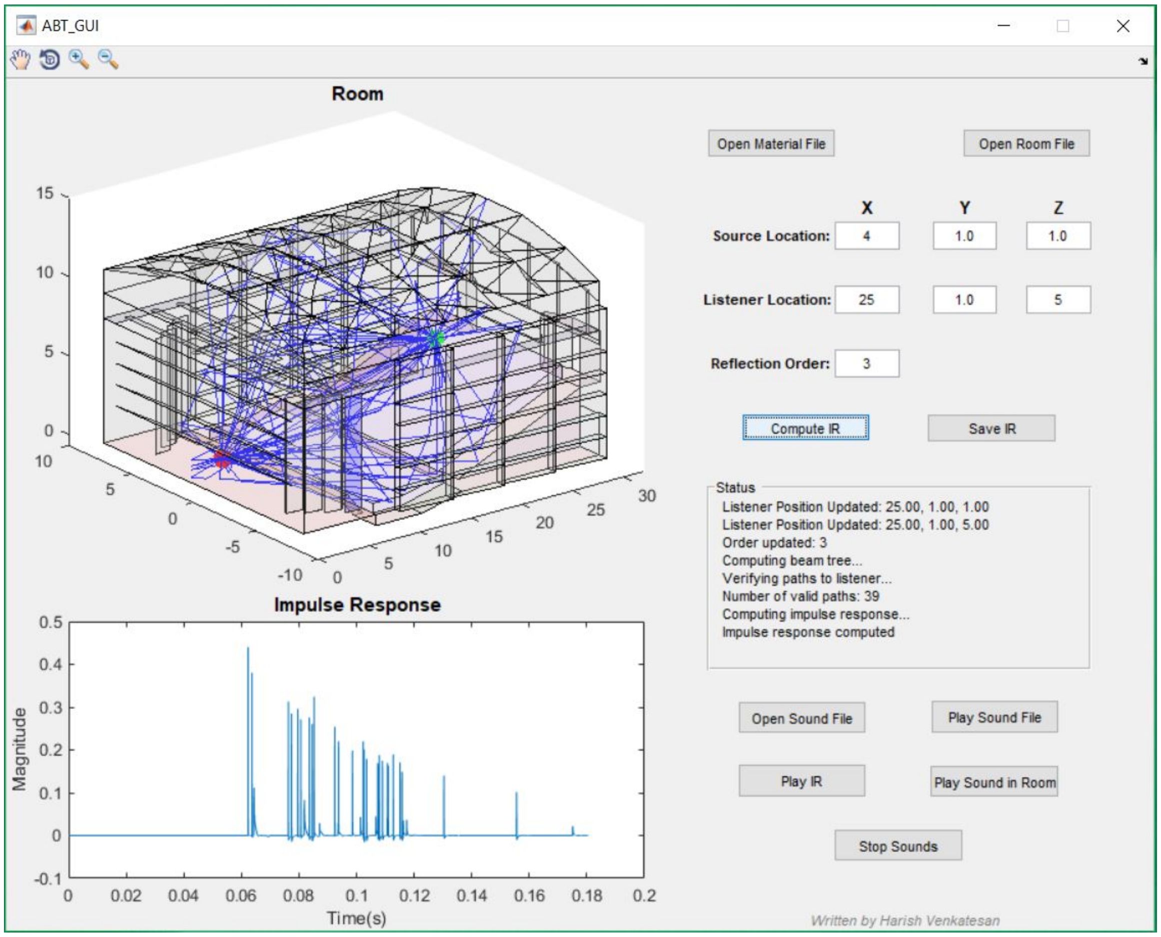
The octave band filters are first implemented in the frequency domain on a log-2 frequency scale such that the filters sum in parallel resulting in 0dB magnitude throughout the spectrum (figure 7). The frequency response of the filters are then dewarped to a linear frequency scale. Time domain filter kernels of length 128 samples at 48kHz sampling rate are built for each path by applying the above computed gains to the bands in the frequency domain, converting the summed response to minimum phase response[6] and then computing the inverse fourier transform.

The spherical path loss gains for all the path are found with respect to the direct path. These gains are applied on the filter kernals previously computed. The impulse response kernel is then filled by placing the filter kernels with appropriate delays found using the path lengths after performing interpolation to insert them at fractional time delays. Figure 6 shows the impulse response computed for the given room setting.

The GUI also provides the capability of saving the impulse response into a .wav file for use outside Matlab. Within the Matlab a user can open a sound file using the GUI, apply the computed impulse response and listen to the early reflections of the sound as though it was played in the room at the given source and listener positions.

[5]Guide to using C++ classes within Matlab by Jonathan Chappelow: `https://github.com/chappjc/MATLAB`

[6]Conversion to Minimum Phase - CCRMA: `https://ccrma.stanford.edu/~jos/fp/Conversion_Minimum_Phase.html`



	X	Y	Z
Source Location:	4	1.0	1.0
Listener Location:	25	1.0	5
Reflection Order:	3		

```
Status
Listener Position Updated: 25.00, 1.00, 1.00
Listener Position Updated: 25.00, 1.00, 5.00
Order updated: 3
Computing beam tree...
Verifying paths to listener...
Number of valid paths: 39
Computing impulse response...
Impulse response computed
```

Figure 6: This figure shows the reflections up to 3^{rd} order with the source (red dot) on the stage and a listener (green dot) at the back of the hall in the audience. The lower plot shows the impulse response for the given setting. This room is a part of the EVERT library.

5 Conclusion

In this paper, a detailed study of the functioning of the EVERT library has been presented, with an explanation of the precalculation of the room BSP tree, construction of the beam tree for a given source position and the ray-tracing path validation step along with the two optimization techniques presented in [1]. Although there are several citations in [1] pointing to academic research papers for the implementation of the algorithm, the information about the exact methodology is somewhat obscure. Code tracing of the EVERT library gave a clearer picture of the exact implementation, which has been documented in this paper. Apart from that, a set of Matlab wrappers for the C++ classes in the EVERT library and a simple GUI for visualization were also presented in this paper.

Although the EVERT library is an exceptional implementation of the accelerated beam tracing algorithm proposed by Laine et al., certain parts of the implementation can further be improved in order to speed up the algorithm. Firstly, in the skip sphere optimization technique, the nodes of the beam tree are grouped into buckets which are then process sequentially in the EVERT implementation. Parallelizing the bucket processing part of the algorithm can drastically improve

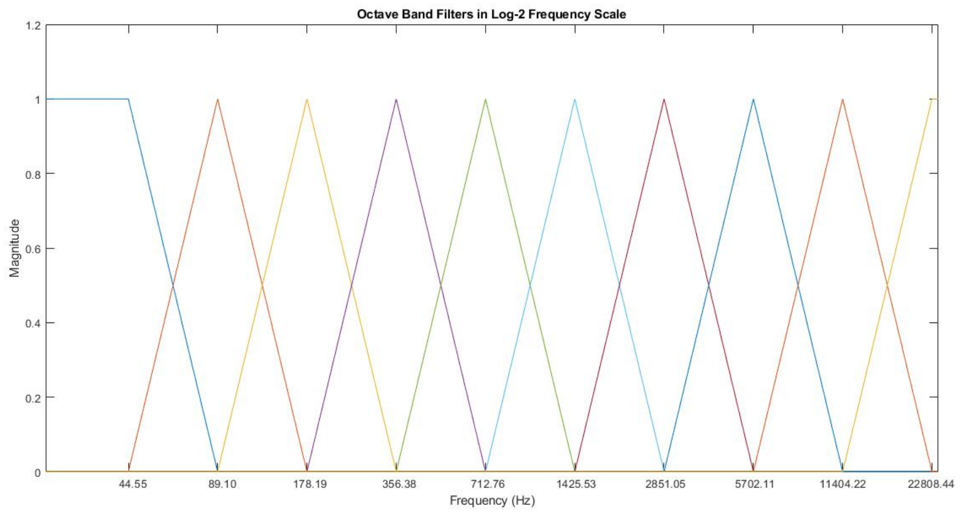


Figure 7: Octave band filters with the first filter being a low-pass filter and the last filter being a high-pass filter.

the performance of the path validation step.

Secondly, in the EVERT implementation, the image sources of different reflection orders are computed several times in the validation step of a single path, by mirroring the sources at each node with the corresponding reflecting polygon. This redundant computation of the image sources can be avoided by storing the image source locations which are available in the beam tree construction step, within the nodes of the beam tree along with the already stored reflecting polygons. Although this might provide a barely perceivable improvement in the algorithm, it would certainly reduce some processing load which can be used else where.

Future steps would involve further optimizing the current implementation by making the above suggested improvements and conducting formal timing tests of different segments of the implementation. Since the algorithm does not take into account the directivity of the source, which is perceptually significant in the real-world, it would be interesting to explore how the directivity information could potentially be utilized to help improve the algorithm.

References

[1] Samuli Laine et al. “Accelerated beam tracing algorithm”. In: *Applied Acoustics* 70 (Jan. 2007), pp. 172–181. DOI: `10.1016/j.apacoust.2007.11.011`.

[2] Nikunj Raghuvanshi et al. “Precomputed Wave Simulation for Real-time Sound Propagation of Dynamic Sources in Complex Scenes”. In: *ACM Trans. Graph.* 29.4 (July 2010), 68:1–68:11. ISSN: 0730-0301. DOI: `10.1145/1778765.1778805`. URL: `http://doi.acm.org/10.1145/1778765.1778805`.

[3] Lauri Savioja and U. Peter Svensson. “Overview of geometrical room acoustic modeling techniques”. In: *The Journal of the Acoustical Society of America* 138.2 (2015), pp. 708–730. DOI: `10.1121/1.4926438`. eprint: `https://doi.org/10.1121/1.4926438`. URL: `https://doi.org/10.1121/1.4926438`.