Generating acoustic projections using 3D models

Jake A. Brazelton

GENERATING ACOUSTIC PROJECTIONS USING 3D MODELS

by

Jacob A. Brazelton
A Thesis
Submitted to the
Faculty
of
James Madison University
In Partial Fulfillment of
The Requirements for the Degree
of
Bachelors of Science

Committee:

____________________	John Bowers, Ph.D., Thesis Director
____________________	Michael L. Norton, Ph.D., Committee Member
____________________	Caroline P. Lubert, Ph.D., Committee Member
____________________	Mikael Glago, Committee Member
____________________	Sharon Simmons, Ph.D., Department Chair
____________________	Dr. Robert Kolvoord, Dean, College of Integrated Science and Engineering

Date: ____________________

Spring Semester 2020
James Madison University
Harrisonburg, VA

Table of Contents

List of Tables

List of Figures

Acknowledgments

The author would like to acknowledge Dr. Bowers for his immense help and support. Without his keen insight on 3D geometry this project would surely have been a loss.

Dr. Norton, Dr. Lubert, and Prof. Glago for taking the time to review this work and point it in the right direction.

The JMU Honors College for providing the opportunity and resources to complete this project.

Abstract

GENERATING ACOUSTIC PROJECTIONS USING 3D MODELS
Jacob A. Brazelton
James Madison University, 2020
Thesis Director: John Bowers, Ph.D.

Raytracing is used in commercial graphics engines most commonly for lighting effects, but it also has many uses when it comes to acoustic simulation. Adopted directly from these computer graphics programs, the formulas presented herein enable the visualization of acoustic intensity levels throughout a 3D space using Python 3 and the OpenGL library. In addition to visualization, they also provide the ability to calculate the reverberation time and critical distance of an enclosed space in relation to its size and material makeup. The described application bundles all of these components together in a Qt5 application that allows users to view the aforementioned properties of provided 3D room models as well as manipulate the surface materials to desired effects. The application has a mainly educational purpose and is intended to allow students to better understand acoustic concepts through hands-on manipulation and visualization while also providing relevant information on how these results are calculated.

Chapter 1: Introduction

Computers have been in use for decades to calculate and model the acoustics of rooms. They can often provide information like the reverberation time, dead zones, resonant frequencies, and other quantities that are quite difficult to calculate without explicit real-time testing. These findings can aid architects and acousticians in designing spaces from classrooms to cathedrals with the exact acoustic properties that they want. Despite the potential importance of these calculations, programs that do these kinds of calculations don't often describe the computations that are actually being done.

The objective of this program is to have software that calculates the acoustic properties (of 3D models) that can be used to analyze a given room. This software is not intended to reinvent an entire architectural design program like AutoCAD, SketchUp, or replace the few acoustic simulation programs out today. Instead, its aim is to provide an easy to use, free, standalone tool for presenting acoustic room properties for use by anyone. Despite dealing with many complex acoustic concepts, like decibel drop-offs and room reverberances, the software is aimed at users who are not necessarily professional Acousticians or Architects, but instead meant for anyone who wishes to make informed decisions or simply know more about the spaces they use, by providing important acoustic information. This information could aid them in key decisions like choosing acoustic treatments, determining speaker and microphone placements, or even modifying the structural make-up of the space entirely. Since this

software is aimed at non-professional users, it will provide easy to understand information about the spaces it is analyzing, while also defining the acoustic jargon to help users learn many of these important acoustic concepts.

The software can have many educational uses because it explains the acoustic concepts and can supplement in-class learning with visualization of the very same concepts in a program easily used by students as well as instructors. This may provide a major benefit compared to the few major architectural applications that already have acoustic rendering plug-ins. These applications are quite complicated to use and have steep learning curves for new users, making them not ideal for educational purposes; to use the desired acoustic plug-ins, instructor and student must already be proficient in the base application effectively. These programs, along with the few standalone acoustic simulation programs already out, can also be quite expensive (some cost upwards of $2,000 [1,2],) which may put off instructors and users.

This paper is organized as follows: Chapter 2 discusses the current literature in the field that relates to the calculations performed by the provided software. Chapter 3 gives the technical descriptions of the formulas and designs used in relation to current literature. Chapter 4 provides a detailed description of the software and its implications. Finally, conclusions and future work are discussed in Chapter 5.

Chapter 2: Literature Review

Acoustic calculation methods generally fall into two distinct categories: wave-based methods and geometric methods [3]. Wave-based methods are based on finite equations that are numerically solved to represent the sound waves being simulated. These kinds of methods are often implemented using finite element methods (FEM) or 3D waveguide meshes, which both provide very accurate acoustic calculations with the ability to model wave properties such as interference, diffraction, and scattering [4,5]. Though quite accurate, both of these methods are very computationally expensive, with time complexities of $O(n^3 log n^3)$ where n is proportional to the frequency of the sound wave being modeled [6]. This makes wave-based methods quite useful for low-frequency sound, which is greatly affected by wave properties such as diffusion and diffraction, but far too time intensive for use with mid- and high-frequency sounds.

The second category of acoustic calculations, geometrical methods, is far better equipped for the modeling and acoustic calculations that involve higher frequency sounds. Much of the basis for these methods comes from optical fundamentals where rays are good approximations for light waves, and most of the properties of these methods translate to acoustics as well. Geometrical methods treat sound waves as particles moving along directed rays emanating from a source rather than a complicated waveform and assume boundaries (surfaces) to be locally reacting, meaning that the surface impedance is the same regardless of angle of incidence [7]. This greatly simplifies the calculations of geometrical methods and helps keep them running faster than wave-based methods

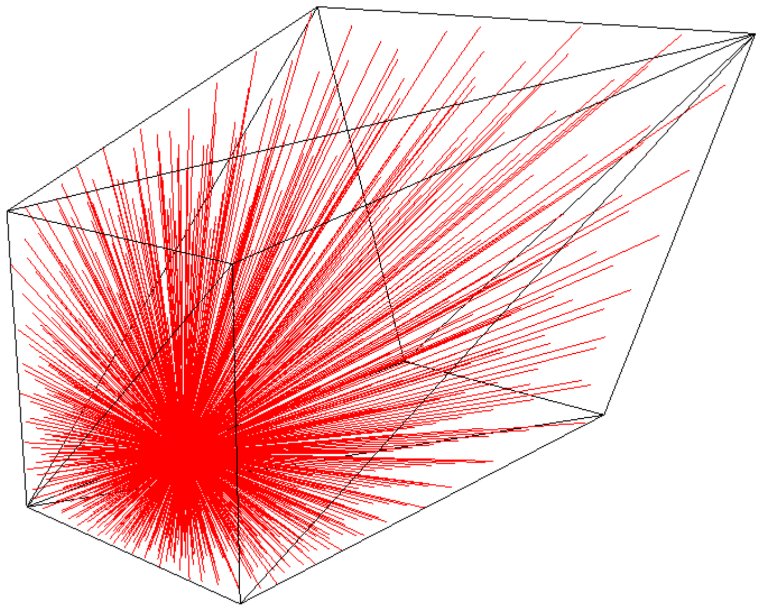

Figure 2.1: Raytracing a simple theater.

as frequencies increase, since frequency is not a strong factor in the ray representation of a wave. The downsides to geometric methods are that since waves are approximated as rays, their wave properties such as wavelength, phase, diffraction, and diffusion are often disregarded, unlike wave-based methods. Ignoring certain wave properties is the trade-off geometric methods have in exchange for being faster than wave-based methods, which often makes them ideal choices for simulating acoustic environments in video games and other graphics based applications.

Raytracing is a relatively common concept in graphics based applications such as video games and modeling tools such as AutoDesk and Maya. As stated, it involves tracing n rays from a source within a space, usually represented as a point source or spherical sound source of uniform distribution. Each of these rays is followed outwards until it intersects with a surface of the modeled room, whereupon its reflections are typically calculated using the Phong BRDF algorithm that represents a surface's

specular and diffuse properties. This process can recursively follow ray reflections to allow for an infinite number of reflections to be calculated where the point of reflection is treated as the new ray's origin. Since rays can diverge greatly upon reflection, raytracing can be very space intensive as the number of reflections increases, which is why, to keep memory costs down, there is often a cutoff point at which rays are no longer considered, such as if its energy level drops below a certain threshold. Raytracing may also take into account absorption properties of a space's surfaces, allowing for more realistic modeling since most surfaces are not 100% reflective and their levels of absorption can drastically affect how reflected rays behave.

Another more recent approach to acoustic modeling in 3D space is that of phonon mapping, which is typically used more for the auralization of a scene rather than finding certain acoustic properties and is adapted from how photons are perceived. This method traces phonons from the sound source throughout the space, where each phonon retains its energy spectrum, distance traversed from the source and other positional information [6, 8]. The phonons emanate from the source until they intersect with a surface where they stop and record the information on how far they've travelled. A listener point is then chosen and the phonon map of the room that was created is used to determine the acoustic levels at the listener's position. This method is extremely good at handling the specular and diffuse properties of sound waves when interacting with different surface materials, but its main advantage is that the phonon map can be reused for any listener point once it is generated. This is a step above pure raytracing where rays must be retraced every time the listener point moves, and is therefore a very good method to use in static 3D environments in video games and similar applications where the sound source does not move.

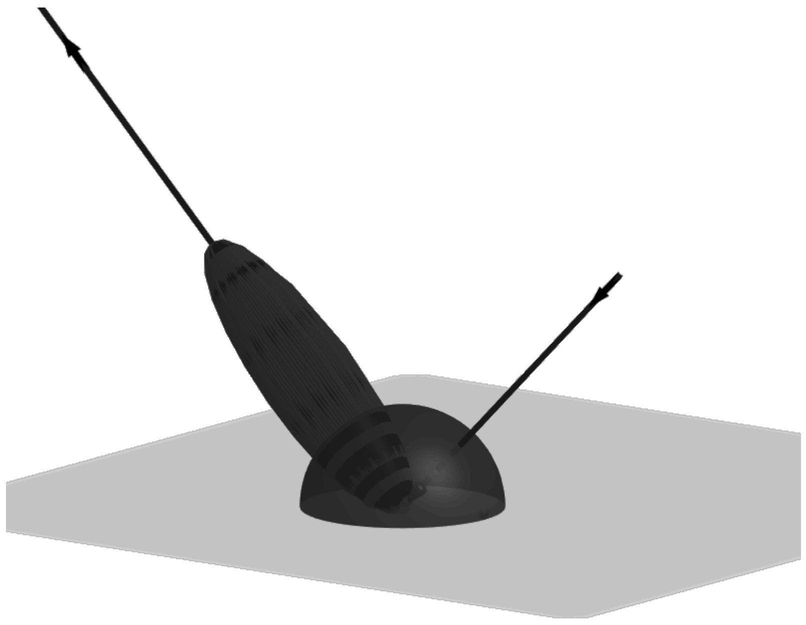

Figure 2.2: BRDF visualization.

Our method of calculating the acoustic properties of spaces pulls bits and pieces from the common raytracing technique and phonon mapping to provide the efficiency of raytracing and the specular and diffuse characteristics modeled by phonon maps utilizing the Phong BRDF model. Bidirectional reflectance distribution functions (BRDFs) model how sound reflects from a surface in accordance with the surface's diffuse and specular properties [9]. Since most surfaces are not pure mirrors to sound, the sound waves are not reflected only in the perfect reflection direction but more often outwards in a lobe defined by the surface specularity, along with some scattering of reflections defined by the diffuse properties of the surface. The Phong model is commonly used in modern graphics shaders to decide how much light intensity is present at a given point in relation to the viewer. This model can easily be applied to acoustics in this project by calculating many reflected rays per reflection and tracing them out, rather than only vetting for those that are viewable from a certain point.

Chapter 3: Project Statement

Many different pieces of software exist that attempt to perform the acoustic analysis that is the focus of this project. Despite this, many of these programs are locked behind price walls and learning curves that do not make them very appealing to the average user who just wants to learn about acoustics or analyze a given space with ease. In this project we attempt to remedy this by providing an accessible application that only deals with room acoustics while explaining the calculations to further the user's knowledge.

3.1 Raytracing

The core of the application is the raytracing algorithm that handles modeling sound waves and their interactions with room surfaces. The algorithm used is a fairly common implementation, first popularized by Arthur Appel, that utilizes the Möller–Trumbore intersection algorithm to determine with which surface a ray intersects. To simplify the calculations, all rooms are assumed to already have their surfaces triangulated when running intersection algorithms. Before any raytracing can take place, a set of n rays are generated using the Fibonacci Sphere algorithm to approximate an even distribution of ray directions radiating outwards from the sound source in a spherical distribution (See Figure A.1 in Appendix A). Each of the generated rays then has its intersection point calculated, and the resulting reflection rays are recursively evaluated for as many levels as desired.

3.1.1 Möller–Trumbore Intersection Algorithm

```python
def isInside(self, point: Vec3, v0: Vec3, v1: Vec3,
    v2: Vec3, normal: Vec3) -> bool:
    """
    Determines if the point is inside the given triangle
    bounds and on the same plane
    """
    edge0 = v1.sub(v0)
    edge1 = v2.sub(v1)
    edge2 = v0.sub(v2)
    C0 = point.sub(v0)
    C1 = point.sub(v1)
    C2 = point.sub(v2)

    return
        normal.dot(edge0.cross(C0)) >= 0
        and normal.dot(edge1.cross(C1)) >= 0
        and normal.dot(edge2.cross(C2)) >= 0
        and C0.dot(C1.cross(C2)) == 0.0
```

Figure 3.1: (Python) Point-Plain Intersection

The adapted Möller–Trumbore intersection algorithm used in this program checks each ray against every surface of the room until it finds the intersection. First, each surface is verified to make sure that the ray origin does not lie on that surface (See Figure 3.1). Once a surface is determined to not house the ray's origin, the rest of the Möller–Trumbore intersection algorithm is carried out (See Figure 3.2). The barycentric coordinates of the ray intersection point with each surface is computed, which determines if a ray actually intersects the surface, which is a triangular face, and from which direction in relation to the surface normal. Once the intersection point is proved to be within the triangular face, the barycentric coordinates can be converted to (x,y,z) coordinates, which allows the software to calculate reflections.

```python
# Check if ray is parallel to plane
pvec = ray.direction.cross(face.edge2)
det = face.edge1.dot(pvec)
if det > -EPSILON and det < EPSILON:
    continue

invDet = 1.0 / det
# Find barycentric coordinates
tvec = ray.origin.sub(face.vertices[0])
u = tvec.dot(pvec) * invDet
if u < 0.0 or u > 1.0:
    continue

qvec = tvec.cross(face.edge1)
v = ray.direction.dot(qvec) * invDet
if v < 0.0 or u + v > 1.0:
    continue

# Distance from ray origin
t = face.edge2.dot(qvec) * invDet
# Ray moves away from the plane
if t < EPSILON:
    continue

# Intersection point
phit = ray.origin.add(Vec3(*(ray.direction * t)))
```

Figure 3.2: (Python) Möller–Trumbore intersection algorithm

3.1.2 Acoustic BRDF

Bidirectional Reflectance Distribution Functions, referred to as BRDFs, are an approximation of the reflections of waves when they make contact with a surface. Every surface has a diffuse/specular/absorption ratio that describes how, and how much, a wave's energy is dispersed after surface contact, with the three values adding up to 1. The simplification of the Phong reflection model that is used in this program can be presented as such:

$$I_x = \sum_{m \in reflections} (k_d(\vec{S_m} \cdot \vec{N}) + k_s(\vec{R_m} \cdot \vec{V})^{\alpha}) \tag{3.1}$$

where I_x is the percentage of acoustic energy (in dB) of $\vec{S_m}$ at the point of incidence that a reflected ray $\vec{V}$ possesses. k_d is the diffuse reflection constant, which is the proportion of reflected rays that are scattered away from the perfectly reflected ray $\vec{R_m}$. k_s is the specular reflection constant, which is the proportion of reflected rays that are located near the perfectly reflected ray $\vec{Rm}$. $\vec{S_m}$ is the incoming ray, which will be the same for all reflections in m tested for that ray. $\vec{N}$ is the surface normal at the point of incidence. $\vec{V}$ is the current reflected ray for which I_x is being calculated. α is the shininess constant for the surface material and is larger the more mirror-like the surface. All vectors are normalized and the perfect reflection $\vec{R_m}$ is calculated using:

$$\vec{R_m} = 2(\vec{S_m} \cdot \vec{N})\vec{N} - \vec{S_m} \tag{3.2}$$

Figure 3.3 shows the code implementation of the described Phong reflection model with a known incidence ray.

```python
def generate_brdf(Sm: Ray, phit: Vec3, startdB: float,
    distanceFromOrigin: float, face: Face):
    """
    Generates an array of rays whose starting dB levels are
    calculated using the Phong BRDF model
    """
    Rm = Sm.calcReflection(phit, face.normal,
        distanceFromOrigin, startdB)

    V = generateV(phit, face.normal,
        face.normal.dot(Rm.direction), startdB,
        distanceFromOrigin, 100)

    diffuse = face.kd * Sm.direction.dot(face.normal)

    for ray in V:
        specular = face.ks * Rm.direction.dot(ray.direction)
        ray.startdB = startdB * (diffuse + (specular**face.a))

    V = [Rm] + V
    return Rm
```

Figure 3.3: (Python) BRDF algorithm

The function `calcReflection` implements the described perfect reflection equation (3.2) **with `Rm` storing both its origin and direction vectors. `generateV` returns a list of rays in a hemisphere around the surface normal utilizing the Fibonacci Sphere algorithm mentioned in Section 3.1. `V` is then iterated through with** I_x **being calculated each time and then multiplied by the incoming ray's acoustic energy level (in dB) at the point of incidence after absorption to determine how much acoustic energy each reflected ray possesses.**

3.2 Acoustic Calculations

Despite the raytracing algorithms making up the core of the computations in this program, the resulting data needs to be converted to acoustic properties for the user to understand the it. Acoustics is the study of mechanical waves in gases, liquids and solids most often referred to as vibrations or sound. The most common objective unit for measuring acoustic intensity is the decibel (dB), which is measured on a logarithmic scale with base 10. An important property of the decibel is the change in level by +/- 6dB when the listener halves or doubles their distance from the sound source, respectively. This means that decibel level changes also occur on a logarithmic scale, meaning intensity levels of sound appear to change far more rapidly in close proximity to the sound source rather than far away. Decibels are an accurate measurement of what the human ear perceives as ”loudness” and will be used in this paper to convey sound intensity.

3.2.1 Decibel Calculations

As previously stated, decibels are measured on a logarithmic scale to better represent the hearing patterns of the human ear. How decibel levels change as the listener's position changes is represented by:

$$L_2 = L_1 + 20\log_{10}(\frac{r_1}{r_2}) \tag{3.3}$$

where L_2 is the new sound pressure level in dB, L_1 is the sound pressure level in dB at distance r_1 from the sound source, and r_2 is the new distance from the sound source.

This equation is valid for any distance r_1, r_2 where neither are ≤ 0 and by default decibel levels of sound sources are measured at 1 unit (typically meters) from the source. Due to the Law of Superposition, sound waves do not distort when passing through each other but instead combine amplitudes while occupying the same space. The combination of amplitudes, known as *interference*, can be positive when the wave-forms are in phase and negative (destructive) when they are out-of-phase. This can lead to an effect referred to as phasing, where "beats" can be heard when two or more sound waves that are close in frequency combine due to the interference patterns produced. The decibel level of two or more sounds waves combined is not as easily calculated as simply summing the decibel levels of all waves involved and is represented as:

$$L_\Sigma = 10 \log_{10} \sum_{i=1}^{n} (10^{\frac{L_i}{10}}) \tag{3.4}$$

where L_Σ is the summed decibel level, n is the number of waves being summed and L_i is the decibel level of each wave. From this equation it can be seen that when two sound waves with the same decibel level intersect, the sum of their levels is 3 dB higher than their individual level. It is also notable that if sound waves differ in decibel level by more than 10 dB, the lower level will add a negligible amount to the higher, allowing for code optimization by vetting decibel levels before summing.

3.2.2 RT$_{60}$ and Critical Distance

Reverberation is the persistence of sound in a room after it is produced. For example, in a large space, like a church, you can hear the sound in the room well after you stopped making it. The reverberation of a room is dependent on the frequency band of the sound produced, the area of the room, and its makeup, i.e. what materials its

reflective surfaces are made of and how much sound energy they absorb. A common objective measurement of reverberation is known as RT$_{60}$ (Reverberation Time 60dB), which is the measure of how long the sound pressure level of a room takes to drop by 60dB after it was generated. The equation for calculating the RT$_{60}$ of a space is:

$$RT_{60} = k * \frac{V}{A} \tag{3.5}$$

where k is a constant factor, V is the volume of the room in m^3 and A is the total absorption surface area of a room in m^2. k is derived from:

$$k = \frac{24 \ln 10}{c_{20}} \tag{3.6}$$

where c_{20} is the speed of sound in air at 20 degrees Celsius, meaning $k = 0.161$ when using meters ($c_{20} = 343m/s$). A is derived from:

$$A = \sum S * \alpha \tag{3.7}$$

where it is the summation of each room surface's absorption area with S being the area of the surface in m^2 and α being the Sabine absorption coefficient of the surface with $\alpha \in [0, 1]$.

Knowing the RT$_{60}$ of a space also allows for the space's critical distance to be easily calculated. The critical distance of a room is the point within it where the sound pressure level of the reverberant (reflected) sounds are equal to the sound pressure level directly from the source. This 'sweet spot' can be found manually if you move around a room while a speaker is playing and find a spot where the sound is much

louder than everywhere around it. This location is dependent on the area of the room as well as the makeup of its reflective surfaces and is conveniently calculated with the RT$_{60}$ value of a room. The equation for finding the critical distance is:

$$d_c = \sqrt{\frac{24 \ln(10) * V}{16\pi * c_{20} * RT_{60}}} = 0.057 * \sqrt{\frac{V}{RT_{60}}} \tag{3.8}$$

where V is the volume of the room in m^3, c_{20} is the speed of sound in air at 20 degrees Celsius, and RT_{60} is the reverberation time of the room in seconds. Both of these calculations are very useful in designing spaces meant to have specific rates of sound decay and are easily done in addition to the ray-based calculations of the program.

Chapter 4: Analysis

4.1 Graphical User Interface

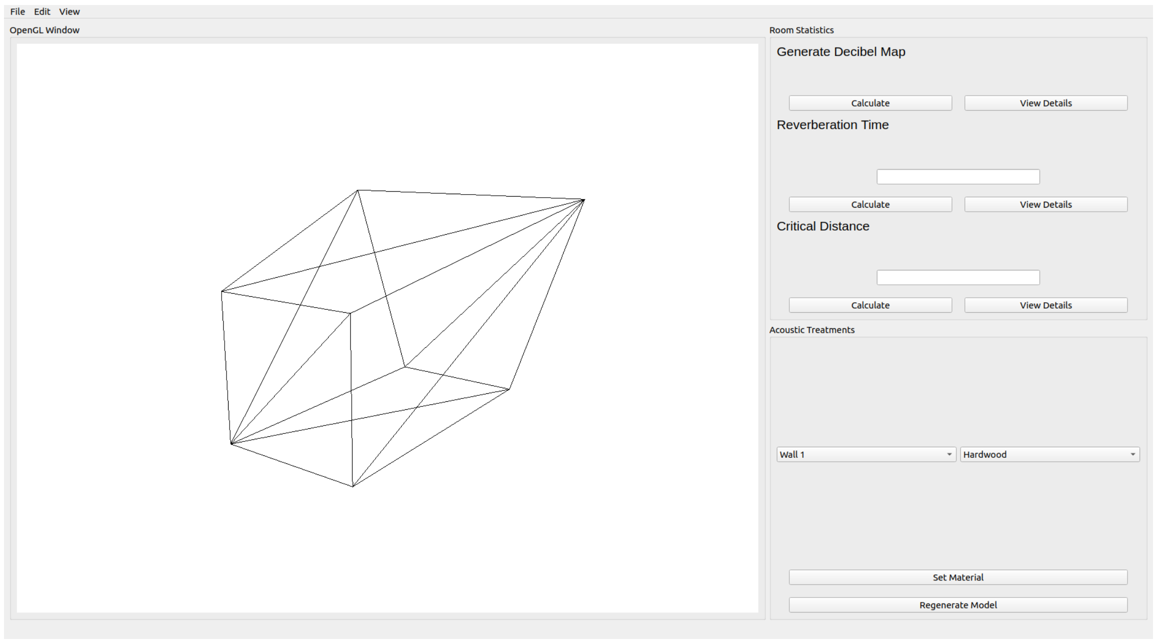

Figure 4.1: Application GUI

This project is intended for use by everyday users and teachers to help learn more about room acoustics. To meet the easy-use goal of this endeavor, the graphical user interface (GUI) of the application had to be simple in design and apparent in its functionality. Figure 4.1 shows the main GUI of the application, which is broken into four major components: the taskbar, the OpenGL window, the acoustic calculations box, and the materials box.

4.1.1 The Taskbar

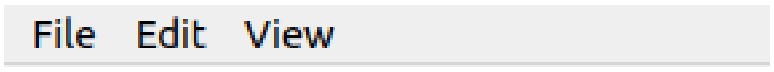

Figure 4.2: GUI Taskar

The taskbar of the application GUI provides the basic functionality that users would expect to find. Under the *File* tab there are two options: *Open* and *Export...*. Both open up a file explorer dialog where the user can select a *.obj* file to open or give the name and location of a *.png* file that is a screenshot of the OpenGL window. The *Edit* tab contains the physical options for *Undo* and *Redo* which can also be hot-keyed using CTRL-Z and CTRL-Y respectively. Lastly, the *View* tab gives the user the option of hiding the acoustic calculations and materials boxes to optimize viewing of the OpenGL window.

4.1.2 The OpenGL Window

The OpenGL window is the main visual tool of the application and also allows the most interaction by the user. Any *.obj* file opened in the application is rendered there in wireframe, allowing the user to see the clean visualization of the space. Once rendered, the user is able to manipulate it by moving the object around using the mouse, rotating it on the x- and y-axis with the left mouse button while holding shift, and rotating it on the z-axis with the right mouse button while holding shift. The user can also zoom in and out of the object using the scroll wheel. When the decibel map is calculated for the object, the OpenGL window will also show the color mapping across the surface of the object for a visual representation of how sound decays throughout the room (See Figure 4.3).

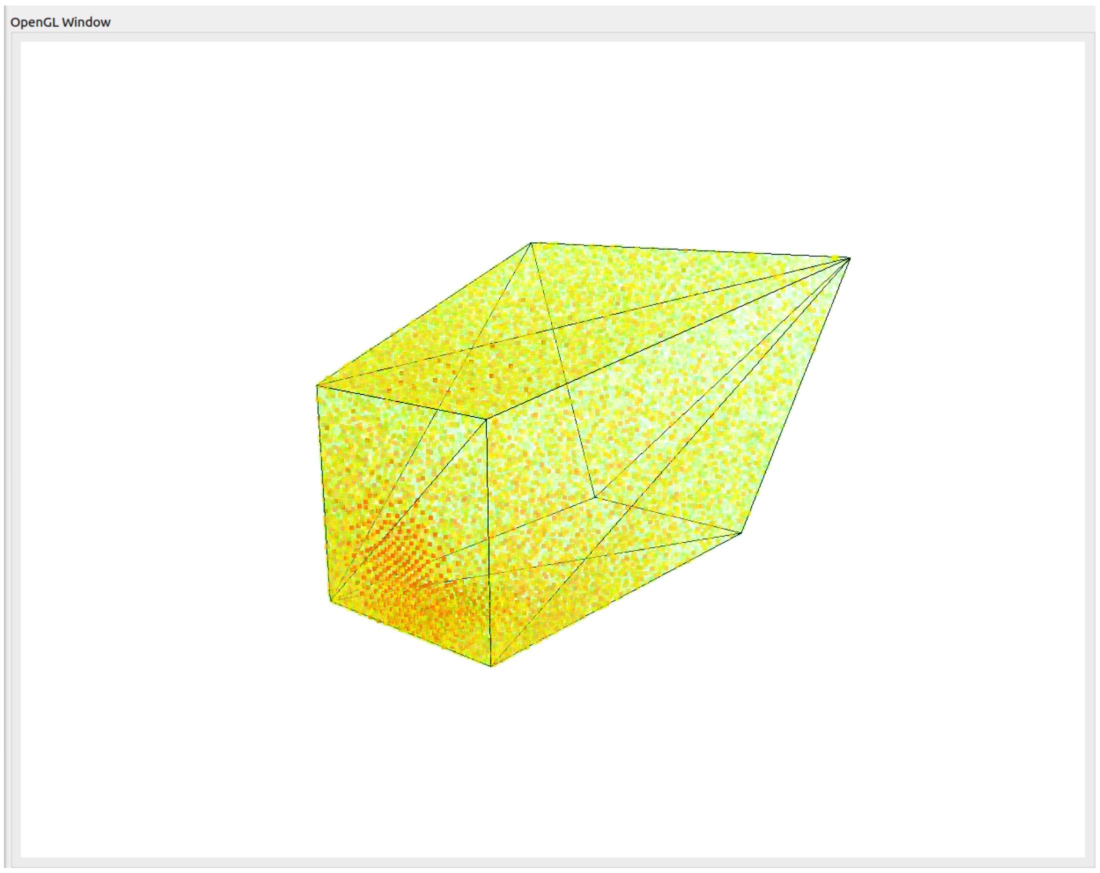

Figure 4.3: OpenGL Window

4.1.3 The Acoustic Calculations Box

The Acoustic Calculations Box controls the running of the acoustic analysis functions described in Sections 3.1 and 3.2. Clicking the *Calculate* button in the Generate Decibel Map section runs the raytracing algorithm that calculates decibel drop-off throughout the room and renders that as a color map in the OpenGL window. The *Calculate* button in the Reverberation Time and Critical Distance section run their respective calculations and present them in their respective text boxes. The *View Details* button for each section is more educational and provides a brief description of the section's topic, how it's calculated, and provides a link to an external source where the user can learn more about the topic.

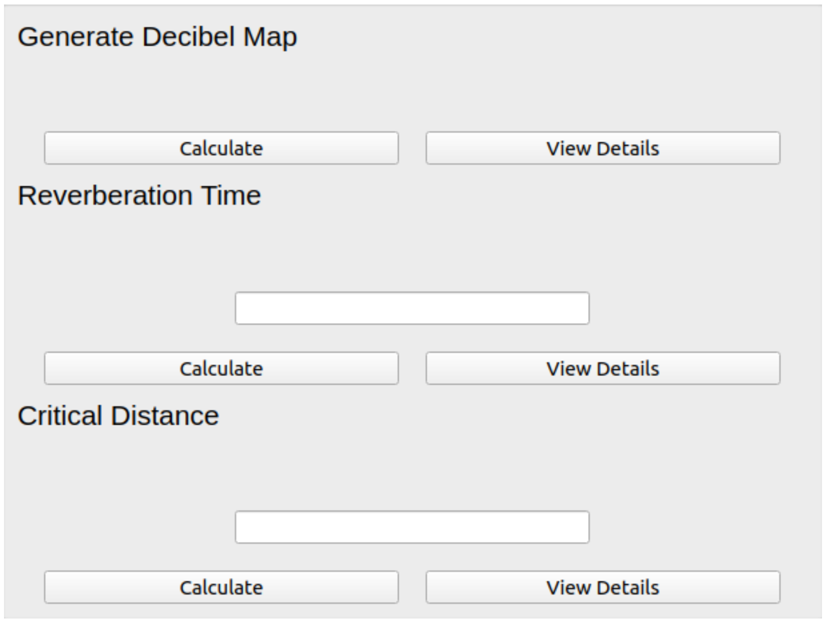

Figure 4.4: Acoustic Calculations Box

4.1.4 The Materials Box

The Materials box controls what type of material makes up each surface of the room. Changing these allows users to test out how different materials absorb acoustic energy at different rates and also gives a more accurate view of the room they are modeling. This is a rather rudimentary implementation that lets the user select from a drop–down menu of surfaces and a drop-down menu of material options and then set the material to the surface using the *Set Material* button. After the desired materials have been set, the *Regenerate Model* button reruns the raytracing algorithm to generate the updated decibel mapping, basically acting as a duplicate of the *Calculate* button in the Acoustic Calculations box. The materials provided for selection are common room materials with well known absorption ratios, making them easy to calculate and recognizable to the user.

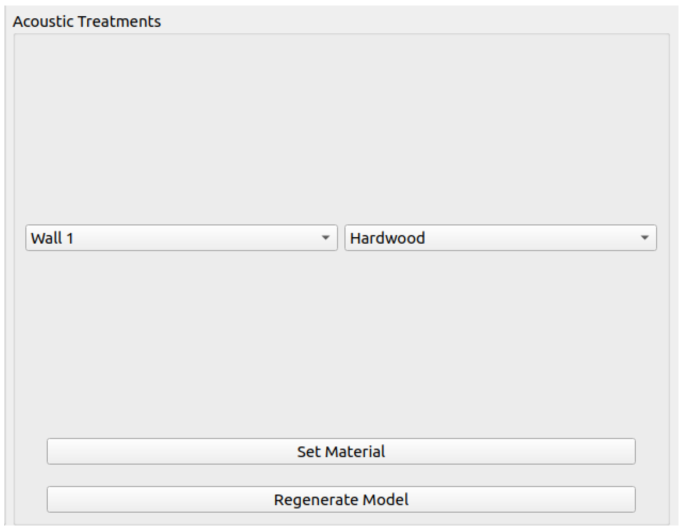

Figure 4.5: Materials Box

4.2 Application Design

Coding this project was just as much a learning experience as was researching the acoustics, and it was constantly revised throughout the process. The project began in Java utilizing the Processing framework, a wrapper for the OpenGL library, which is often used for interactive animations and 3D rendering. Despite its promising features, Processing fell short for its lack of transparency to underlying OpenGL elements that were necessary to access and its lack of easy UI integration. These issues ultimately led to the switch to Python 3, which allowed direct access to OpenGL and the use of the robust UI framework Qt [10].

4.2.1 Qt Framework

The Qt framework is a robust UI design system that is used across many platforms by major companies like LG, Mercedes Benz, AMD, and Valve, in desktop applications like AutoDesk Maya, Electrum, and Telegram, and even desktop environments

like KDE Plasma. Qt also has the added benefit that it directly supports OpenGL, making it the perfect framework to build the application and ensure ease of use and easy access by many users. The GUI described in Section 4.1 is built entirely using the Qt framework, which provides the necessary components of buttons, menus, drop-downs, and even the OpenGL window with minimal code, allowing for little time needed to be spent figuring out all of the framework's ins and outs and more time for coding the complex algorithms needed for the acoustic calculations that use raytracing.

4.2.2 OpenGL

This application relies heavily on the fact that users can see the room models that they wish to analyze and be able to manipulate them, which drives the need for the OpenGL library. OpenGL is another cross-platform library that provides an API for rendering 2D and 3D vector graphics used across many industries that utilize embedded graphics processing. Being an industry standard, there were plenty of resources for learning how to use the OpenGL library, which helped immensely in fast tracking its integration into the program. As mentioned before, the fact that it integrated seamlessly into the Qt framework only made it more useful, allowing the code to flow seamlessly together while writing it.

4.3 Code Development

4.3.1 Reading .OBJ Files

The first hurdle of the code development process was determining the acceptable file formats of the 3D room models and how to parse them for use in the application. The *.obj* file format was the obvious choice to use due to its simplicity and wide use in other 3D modeling software, making it easy for users to export room models from their favorite software and import them into the application. The format provides the vertices of the model, the vertex normal of each and the faces comprised of vertices, giving the program everything it needs to construct the model in OpenGL. Figure A.2 in the Appendix shows an example *.obj* file that constructs a cube. The v means vertex, vn means vertex normal, and f means face, where each input to a face is v/vt/vn. (vt means texture vertex, which is not used in the application)

4.3.2 Storing Object Data

When a room model is parsed from a *.obj* file, its many geometric components are stored in a way that makes them easy to reference during later calculations.

Vertices and Vectors

The most basic geometric property of a model is its vertices, which in 3D space are vectors comprised of three coordinates (x,y,z). The `Vec3` class fills this role by being a wrapper for the `numpy` library's `array` that only holds three floating point numbers which can handle storing coordinates and direction vectors. The class provides the functionality to add and subtract vectors, calculate dot and cross products, normalize vectors and more to allow easy manipulation of coordinate and direction vectors without repeated, cluttering `numpy` code.

Faces

All rooms have surfaces, which are comprised of a set of at least three connected vertices, referred to in the program as a face. The `Face` class assumes this functionality, storing an array of its vertices as `Vec3` objects and a `Vec3` representing the face's surface normal, which is calculated from the vertex normals (vn) of the *.obj* file or manually from the vertices. Each `Face` also stores the edge vectors $\vec{AB}$ and $\vec{AC}$ for use in the Möller–Trumbore intersection algorithm (see Figure 3.2), its surface area for RT_{60} and Critical Distance calculations, and the surface material (by default hardwood.)

Rays

The last main component for modeling a space and performing raytracing on it is the ray itself. The `Ray` class represents all rays used in the application and is what the Fibonacci Sphere algorithm (see Figure A.1 in the Appendix) outputs to model rays being sent in all directions from the initial sound source. The `Ray` class keeps track of each ray's origin as a `Vec3`, direction vector as a `Vec3`, distance traveled from the sound source, and decibel level at the ray's origin.

4.3.3 The Raytracing Process

```
    # Intersection point
    phit = ray.origin.add(Vec3(*(ray.direction * t)))

    # Calcualate the dB level at the intersection
    newDistanceFromOrigin = ray.distanceFromOrigin +
        ray.origin.distance(phit)
    dbChange = dropOff(ray.distanceFromOrigin,
        newDistanceFromOrigin)
    newdB = ray.startdB - dbChange

    # Log Point
    currPointdB = self.pointDict.get((phit.x, phit.y, phit.z))
    if currPointdB is not None:
        self.pointDict[(phit.x, phit.y, phit.z)] = sumLevels(
            [newdB, currPointdB]
        )
    else:
        self.pointDict[(phit.x, phit.y, phit.z)] = newdB

    if rNum > 0:
        # Calculate the reflected rays
        reflections = generate_brdf(
            ray, phit, newdB, newDistanceFromOrigin, face
        )

        for ray in reflections:
            if ray.startdB > 0:
                self.intersect(ray, rNum - 1)
```

Figure 4.6: (Python) Point Decibel Level Calculation

The process of raytracing inside of a room model begins with the selection of a point–source from which the rays emanate, representing an omnidirectional sound source. This is entered manually and can be anywhere in the space, including on a surface to represent in-wall speakers. From that point the spherical distribution of rays generated by the Fibonacci Sphere algorithm is iterated over to determine which face each ray intersects using the Möller–Trumbore intersection algorithm. Each ray from the sound source has its distance from the sound source set to "1" and starting decibel set to a user-determined level.

Intersection Decibel Calculations

Upon finding an intersection with a `Face` object of the model, the intersection point's distance from the origin and decibel level for each ray is calculated (See Figure 4.6, continuation of Figure 3.2). A `dict` object is used to store the intersection point and decibel level at that point every time a ray intersects with a surface. That `dict` object is checked to see if there was any other ray that previously hit the same point, and if so the decibel levels are summed (see Equation 3.4), otherwise, the intersection point and its decibel level are added to the `dict`.

BRDF Reflections

Material	125 Hz	250 Hz	500 Hz	1000 Hz	2000 Hz	4000 Hz
Hardwood	0.19	0.23	0.25	0.30	0.37	0.42
Carpet	0.03	0.09	0.20	0.54	0.70	0.72
Drywall	0.29	0.10	0.05	0.04	0.07	0.09
Brick	0.05	0.04	0.02	0.04	0.05	0.05
Concrete	0.01	0.01	0.01	0.02	0.02	0.02
Foam	0.25	0.50	0.85	0.95	0.90	0.90

Table 4.1: Absorption Coefficients

After the decibel level calculations at an intersection point occur, the reflections of the ray-surface intersection are computed using the BRDF algorithm described in Equation 3.1 and implemented in Figure 3.3. The material of the `Face` object involved also comes into play as its absorption level determines how much acoustic energy actually leaves the surface in a perfect reflection. Table 4.1 shows the absorption coefficients of common materials used in the application[11]. Each of the reflections generated are then iterated through and run through the same Möller–Trumbore intersection algorithm to find their intersections, calculate the distance traveled from the sound source and the decibel level at that point, add it to the `dict` and repeat for the desired number of reflection levels, denoted by `rNum`. This process calculates the ray intersections in a depth-first order, removing the need to store rays and their intersection points in large arrays that would be necessary in a breadth-first implementation, which was used initially before realizing how much unnecessary information was being stored.

4.3.4 Rendering the Model

Once all of the raytracing and point-decibel calculations are completed, the last task of the program is to display the newly computed decibel map in an understandable way. To convey decibel levels as colors, it made sense to represent them on a spectrum from red to cyan, avoiding blues and purples due to them being generally darker colors, thus ruining the transition from dark to light. The Hue, Saturation, Value (HSV) color system provided the perfect functionality, as the hue (color) is represented in degrees 0-360, making it easily mappable to the range 0-120 used for decibels. This allows for any decibel level to be converted to HSV, inversely so that 120dB equals red (hue 0), and then from HSV to Red, Green, Blue (RGB) values (see Figure A.3) that OpenGL could render.

Initially, the decibel levels were displayed as rays themselves, changing colors as they moved further away from the sound source, but when reflections were added, the visual became a jumbled mess of colored lines that conveyed no real information due to the sheer number of them. The solution to making the results visually understandable was to change to just coloring the points where rays intersected with a surface, providing a cleaner view of the model that more resembled a heat map of decibel levels. To clean up the visualization even further, points also become more transparent the lower the decibel level, to ensure that higher decibel level points are more prominent. Without adjusting the transparency of the lower level points, the high number of them created by diffuse angles from the BRDF algorithm cloaked every model in greens and light blues, providing an inaccurate view of a room's decibel drop-off.

```python
def render(self):
    gl_list = gl.glGenLists(1)
    gl.glNewList(gl_list, gl.GL_COMPILE)
    gl.glShadeModel(gl.GL_SMOOTH)
    gl.glPointSize(5)
    gl.glBegin(gl.GL_POINTS)
    # Sort the intersection points by dB level
    points = sorted(
        self.pointDict.items(), key=lambda item: item[1],
        reverse=True
    )
    # Adjust alpha values based on dB level
    for vec, dB in points:
        if dB > 100:
            gl.glColor4f(*dBtoColor(dB), 1)
        elif dB > 80:
            gl.glColor4f(*dBtoColor(dB), 0.65)
        elif dB > 60:
            gl.glColor4f(*dBtoColor(dB), 0.25)
        elif dB > 40:
            gl.glColor4f(*dBtoColor(dB), 0.125)
        else:
            gl.glColor4f(*dBtoColor(dB), 0.0625)

        gl.glVertex3fv(vec)
    gl.glEnd()
    gl.glEndList()
    return gl_list
```

Figure 4.7: (Python) Rendering Ray-Surface Intersections

Chapter 5: Conclusion and Future Work

The application created provides users with a hands-on acoustic analysis experience that can be used to teach complex acoustic ideas or for simple utility by amateur acousticians. Its design in Python 3 using the Qt framework and OpenGL make it highly portable, allowing user access across most platforms, and easy for other developers to understand its code if it is open sourced and modifiable. While this application fulfilled the goal of the project, there are a few ways that it could be modified to improve user experience and increase the efficiency of the raytracing process.

- **A glaring issue is the low quality UI that only provides the bare essentials for user interaction. The styling does not stand out from older software and the odd spacing detracts from the clean look that was intended. Using the Qt Designer software rather than just hard coding the UI elements would be a potential fix that would really clean up the UI and make it look more like a professional product.**

- **Another issue is the time it takes to complete the raytracing process, as it runs on one thread and has to handle running the Möller–Trumbore intersection algorithm, which has a time complexity of $O(f)$, where f is the number of faces in the room. That means that the raytracer has a time complexity of $O(fn^{r+1})$ with reflections, where n is the number of rays spawned at the source and each reflection and r is the number of reflection levels. This issue can be solved by splitting the set of initial rays over multiple threads that run the raytracing algorithm on their subset, enabling the process to return visual results to the user**

much faster.

- **A more interactive way of deciding the sound source placement would also be a useful tool, especially since it can currently only be modified in the code. Allowing the user to click a speaker icon that they could move around the inside of the model would let the raytracing operation run multiple times from different locations in the model. Currently, the user must know how to modify the code to change the speaker location.**

- **The last major improvement to the program would be to smooth out the decibel heat map. The current iteration where colored dots are placed on the model's surface works, but processing the points and creating smooth gradients would remove the clutter of lighter color dots that don't mean much energy-wise when next to dots greater than 10dB higher. A possible solution to this would be to create a triangulation of each face's dots and vertices, treat each triangle as an individual face, and use OpenGL's vertex color interpolation to create the gradient.**

Appendix A: Code Samples

```python
def generateRays(self) -> list:
    """
    Generates an array of rays to use for the raytracing
    """
    points = []
    offset = 2.0 / self.raynum
    increment = math.pi * (3.0 - math.sqrt(5.0))

    for i in range(self.raynum):
        y = ((i * offset) - 1) + (offset / 2)
        r = math.sqrt(1 - pow(y, 2))

        phi = ((i + 1) % self.raynum) * increment

        x = math.cos(phi) * r
        z = math.sin(phi) * r

        points.append(Ray(self.origin,
            Vec3(x, y, z).normalize(), 1, self.startdB))
    return points
```

Figure A.1: (Python) Fibonacci Sphere Algorithm

```
# cube.obj

v  0.0  0.0  0.0
v  0.0  0.0  1.0
v  0.0  1.0  0.0
v  0.0  1.0  1.0
v  1.0  0.0  0.0
v  1.0  0.0  1.0
v  1.0  1.0  0.0
v  1.0  1.0  1.0

vn  0.0  0.0  1.0
vn  0.0  0.0 -1.0
vn  0.0  1.0  0.0
vn  0.0 -1.0  0.0
vn  1.0  0.0  0.0
vn -1.0  0.0  0.0

f  1//2  7//2  5//2
f  1//2  3//2  7//2
f  1//6  4//6  3//6
f  1//6  2//6  4//6
f  3//3  8//3  7//3
f  3//3  4//3  8//3
f  5//5  7//5  8//5
f  5//5  8//5  6//5
f  1//4  5//4  6//4
f  1//4  6//4  2//4
f  2//1  6//1  8//1
f  2//1  8//1  4//1
```

Figure A.2: (.OBJ) Cube.obj

```python
def dBtoColor(level):
    """
    Converts given dB level (0-120) to RGB using HSV values
    - Range from Red to Cyan
    - Red  ( >= 120dB ) = (0, 1, 1) HSV
    - Cyan ( <=   0dB ) = (180, 1, 1)  HSV
    """

    # Ensure level is in proper bounds
    level = 0 if level < 0 else level
    level = 120 if level > 120 else level

    hue = (level - 120) * -1.5  # HSV Hue
    return hsv_to_rgb(hue / 360, 1, 1)
```

Figure A.3: (Python) Decibel to RGB Color Conversion

Bibliography

[1] AFMG Technologies. (2011) Ease. [Online]. Available: http://ease.afmg.eu/

[2] Odeon A/S. (2019) Odeon. [Online]. Available: https://odeon.dk/

[3] R. Prislan and D. Svenšek, "Ray-tracing semiclassical low frequency acoustic modeling with local and extended reaction boundaries." *Journal of Sound and Vibration*, December 2018. [Online]. Available: https://doi.org/10.1016/j.jsv.2018.08.041

[4] V. Easwaran and A. Craggs, "On further validation and use of the finite-element method to room acoustics," *Journal of Sound and Vibration*, vol. 187, no. 2, October 1995.

[5] N. Röber, U. Kaminski, and M. Masuch, "Ray acoustics using computer graphics technology," *Proceedings of the 10th International Conference on Digital Audio Effects, DAFx 2007*, September 2007. [Online]. Available: https://dafx.labri.fr/main/papers/p117.pdf

[6] M. Bertram, E. Deines, J. Mohring, J. Jegorovs, and H. Hagen, "Phonon tracing for auralization and visualization of sound," *IEEE Visualization*, pp. 151 – 158, 2005. [Online]. Available: https://doi.org/10.1109/VISUAL.2005.1532790

[7] T. D. Rossing, *Springer Handbook of Acoustics*. New York, NY, USA: Springer, 2007, p. 996.

[8] S. Siltanen, T. Lokki, S. Kiminki, and L. Savioja, "The room acoustic rendering equation," *The Journal of the Acoustical Society of America*, vol. 122, no. 3, pp. 1624–1635, 2007. [Online]. Available: https://doi.org/10.1121/1.2766781

[9] J. Durany, T. Mateos, and A. Garriga, "Analytical computation of acoustic bidirectional reflectance distribution functions," *Open Journal of Acoustics*, vol. 05, 2015. [Online]. Available: https://doi.org/10.4236/oja.2015.54016

[10] The Qt Company. (2020) Qt. [Online]. Available: https://www.qt.io/

[11] Acoustic Traffic LLC. Absorption coefficients. [Online]. Available: http://www.acoustic.ua/st/web_absorption_data_eng.pdf