Simulation of sound reflections through Image Sources

For the Sound in Interaction course, held by
Professor Federico Avanzini

Made by Gianmaria Forte

Introduction

Image sources

Image Sources constitute an **approach used to simulate Early Reflections**: the first phase of reverberation, consisting of sparse signals with high enough energy to be perceived separately.

Given the surfaces of a room and the positions of source and listener, **reflections are represented by image sources**: clones of the original source, whose location is obtained by mirroring its position along the surfaces on which sound reflects. These image sources are then **used to compute the paths followed by sound rays** inside the room; these sound rays serve to approximate with acceptable accuracy the behaviour of soundwaves.

The position of **each image preserves the distance travelled by the sound reflection and the angle of arrival at listener**. The resulting **image sources need to be tested to ensure that the path doesn't encounter obstacles** and can actually bounce on the right surfaces.

To obtain the final sound with reverb, reflections are converted into a train of impulse for the receiver, after the appropriate attenuation due to distance and reflections it approximates the Room Impulse Response (RIR).

The objective of this project

This project omits this last step to obtain the RIR, as computing this filter in real-time and applying the correct attenuations are entirely different matters that require their own study.

The **focus is instead the computation of ISs and the validation of their reflection paths**, with special attention to **implementing different optimizations** and allowing to enable or disable them independently for benchmarking.

IS generation and sound rays

Setup

The reverb simulation of this project is based on a set of elements that recreate a room in which sound can travel, along with the elements needed for sound to exist and be heard.

Specifically, there needs to be a **sound source**, a **listener** and a set of **sound-reflective surfaces**.

Source

The **object emitting sound**, given a position inside the scene. In this project, sound is assumed to be emitted by its source without specific orientation, meaning that sound rays may originate from any angle of the sound source.

Listener

The **object receiving sound**, with a position inside the room.

Surfaces

The set of surfaces defines the **boundaries on which sound is reflected**. They can used to approximate the dimensions of a detailed room without need for extreme accuracy.

In this project, **all surfaces are assumed to be convex and planar polygons**.
The following definitions are going to be useful later:

Convex polygon: a polygon that defines the boundary of a convex set, meaning that a line connecting any two points of the polygon is entirely included in the polygon itself.

Planar polygon: a polygon whose vertices and edges are entirely contained in a single plane.

More complex surfaces can be approximated by combining a set of surfaces that respect the requisites listed above.

For graphical simplicity, **this document always considers rectangular surfaces**. This comes without loss of generality, since all the following considerations easily apply to more complex polygons, as long as they retain the properties of convexity and planarity.

First order image sources

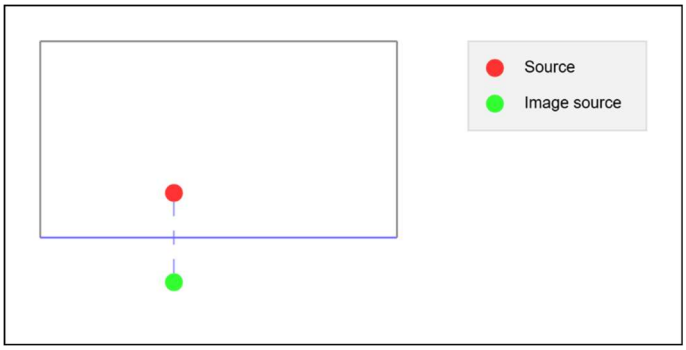
Image Sources (ISs from now on) are **mirror images of the original source**, they are used to **define the path followed by sound rays after they are reflected on a specific sequence of surfaces**.

> **Order of reflection**: number of boundaries inside the room on which a sound ray reflects before it reaches the listener.

Each IS is defined by its order of reflection, and the **complexity of the IS generation algorithm** is in turn **bounded by a maximum order for ISs**.

Creating first order ISs

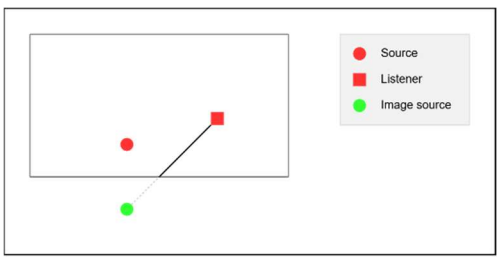
First order ISs are used to simulate the sound reaching the listener after the original signal is reflected on a single surface. The **position of a first order IS is obtained by mirroring the original source's position onto one of the reflective surfaces** of the room:



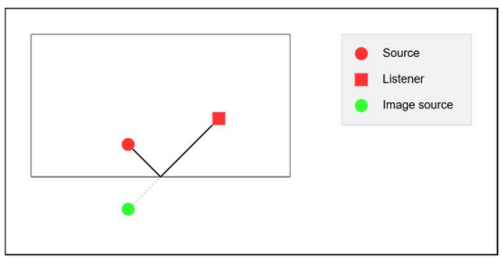
Intuitively, in an environment with N reflective boundaries, there are exactly N first order ISs, one relative to each surface. It's also important to notice how the **positions of image sources are entirely independent from the listener's position**, this means that it's **sufficient to determine their placement once for static sources** that don't move during the simulation.

First order sound rays

The **sound ray**, approximating the behaviour of the soundwave relative to each first order IS, is **obtained by tracing a path going backwards**, from listener to original source:



1 - A ray is traced from listener to IS**. It is valid if the ray intersects the IS's surface (on the front) without encountering obstacles.**



2 - **From the previous intersection**, a ray is traced **to the original source**. It is valid when it reaches the source without any obstruction.

Higher order image sources

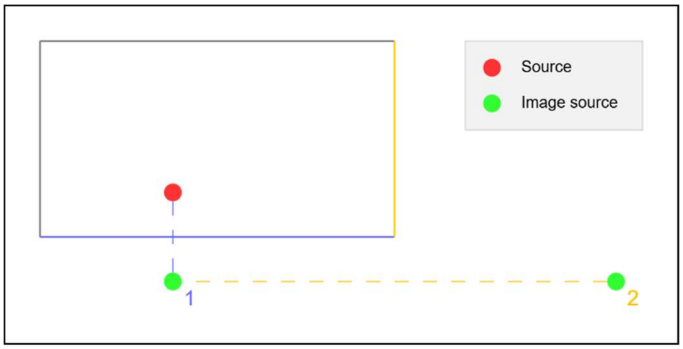
Higher order ISs represent ray paths that bounce off of a sequence of boundaries; without optimizations, there's one IS for each possible sequence of surfaces, excluding any sequences with consecutive repetitions. This is because surfaces, with the assumption of being flat, cannot reflect sound rays back onto themselves.

IS surface sequence: the sequence of reflections against boundaries represented by the IS, used to compute the path for its sound ray.

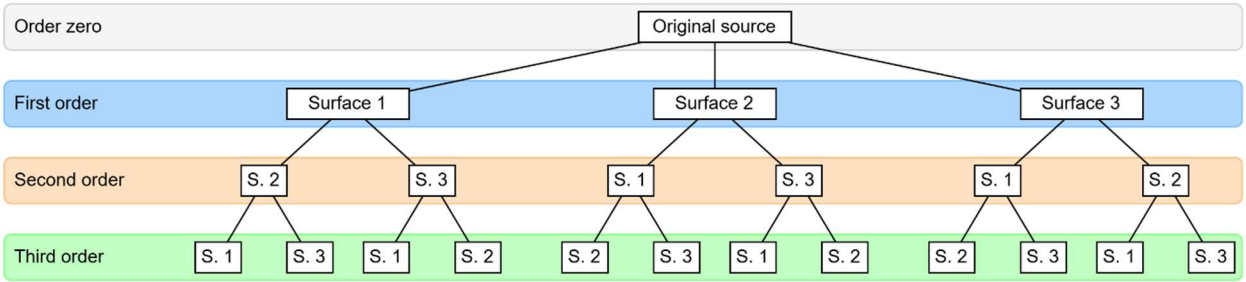
IS's surface: the last surface in the sequence of the IS, the one on which the sound ray bounces before reaching the listener.

Creating higher order ISs

ISs of order $o + 1$ are created using ISs of order o: given **an IS of order o**, it's **mirrored with respect to a reflective surface s, creating in that position a new IS of order $o + 1$**, whose sequence of reflective boundaries is the same as the father IS's + the s surface. This is done for every IS and for every surface, while avoiding consecutive repetitions of surfaces.



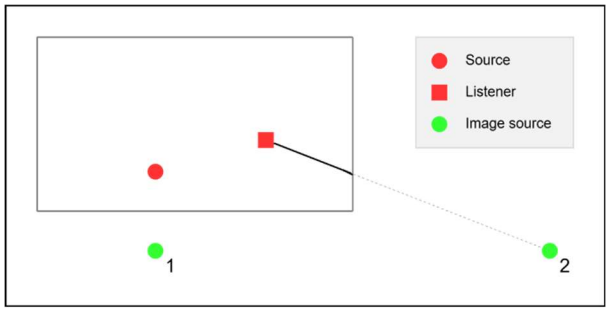
This creates a **tree structure**, where each path from root to node identifies an IS with a specific sequence of surface reflections:



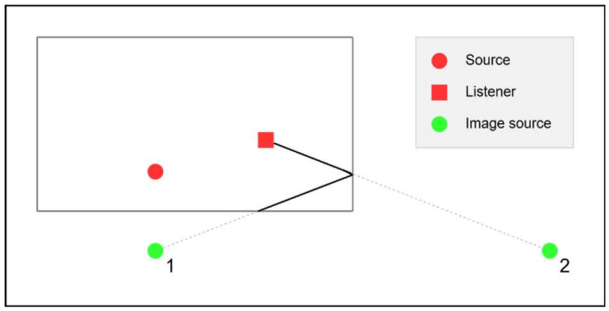
As an example: the first node on the left for the second order of reflections identifies a sound ray reflecting on surface 1 and 2, in that specific order.

Higher order sound rays

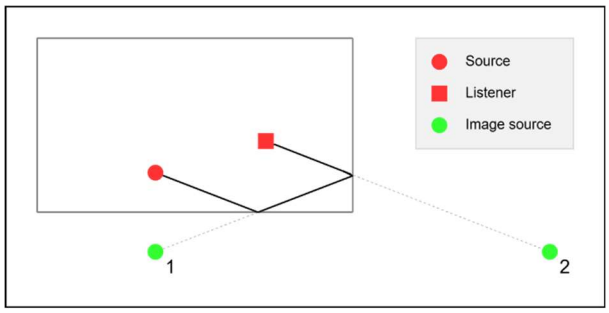
The **sound ray**, approximating the behaviour of the soundwave relative to each IS, is **obtained similarly to the first order ISs by tracing a path going backwards**, from listener to original source:



1 - A **ray is traced from listener to the IS**.
It is valid if the ray intersects the IS's surface (on the front side) without encountering obstacles.



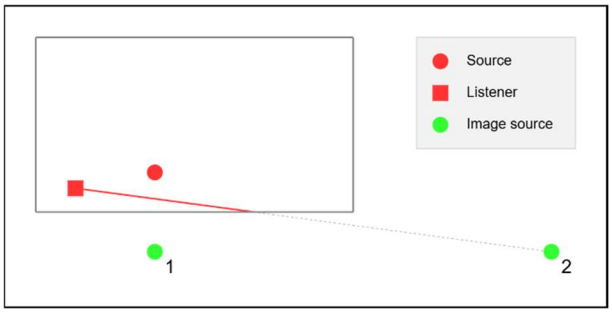
2 - **From the previous intersection**, a ray is traced **to the father IS**, following the image source tree from leaf to root. It is valid when it collides without any obstruction with the IS's corresponding surface (on its front side).



3 – If the path is valid for all ISs in the sequence, then **one last ray is traced from the last intersection to the source**.

Invalid sound ray paths

It's common for an IS's sound ray to be invalid, this means that **the IS would not generate any sound received by the listener**. It's important to note that an IS with no reflection path is **still needed** by the simulation, since it's used **to generate higher order ISs** that may very well have a complete path from source to listener.



On the right is an example of an invalid path, where the first ray traced from listener to IS doesn't intersect the correct surface.

IS generation and sound path tracing in Unity

All the concepts discussed above are applied in Unity, in a 3-dimensional sample scene.
The **sample scene is populated by**:

- **Reflective surfaces** – rectangular and planar, forming a completely closed shoebox room, the back-facing boundaries are not rendered to allow visibility.
- A **listener** – the red cylinder.
- A **sound source** – the red sphere.

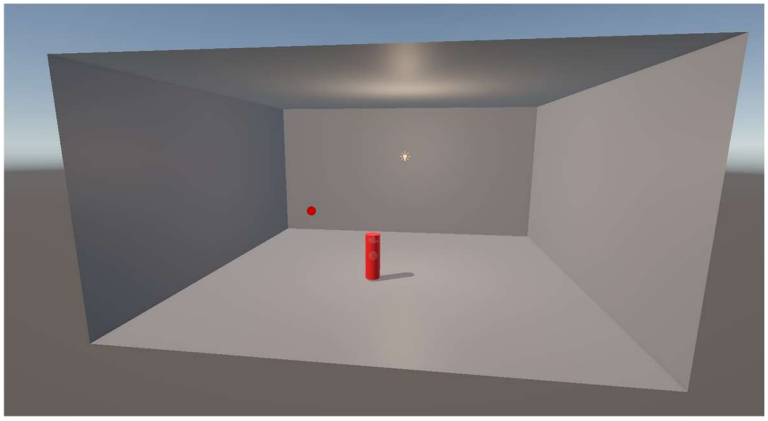


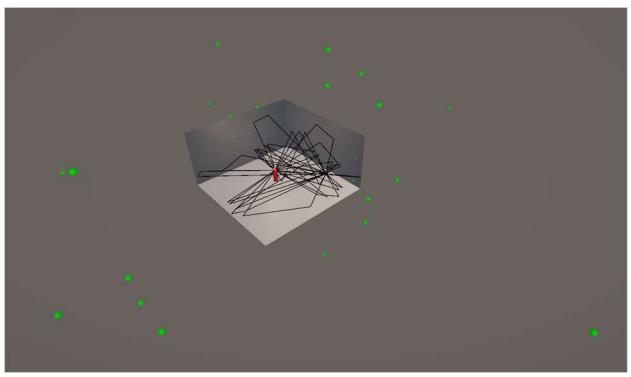
Image sources are created exactly as stated in the previous section, by mirroring the position of their father IS with respect to a surface. **All ISs are saved in a list**, with **each of them keeping**:

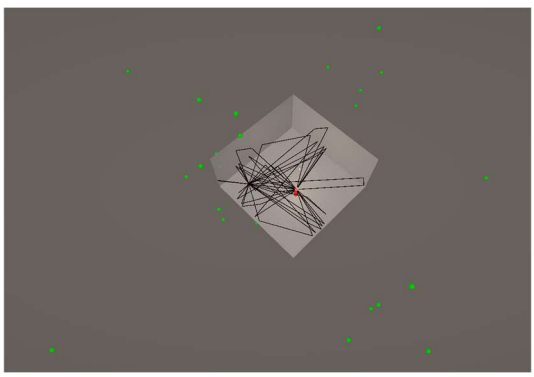
1. Its **index** in the list.
2. The **index of its father IS**.
3. Its **order** of reflection.
4. Its **surface** (the one around which the father IS's position was mirrored).

This is the **ISs tree structure**, allowing to traverse the list upwards towards the root from any IS.

Once all ISs have been generated, each one goes through a **path validation process**: using raycasts, the path is traced going backwards in the IS tree, like shown in the previous section.

The two images below show all ISs of first and second order (green spheres), along with all valid sound ray paths (shown as black lines) in the shoebox room from different viewpoints.





Advantages and disadvantages of image sources

The image sources approach has many interesting features, but they come with some drawbacks.

Advantages

Each IS is positioned exactly at the same angle as the sound ray that would reach the listener and its distance from the listener is the same length covered by the sound ray.

This means that, given that the IS has a valid path, **reverb can be recreated by simulating a sound identical to the original one**, but **with some adjustments**:

- **Energy** lowered and added **delay** due to distance.
- **Auralization** altered according to angle of arrival (the IS's position).
- **Attenuation** filters applied to simulate the loss of energy upon reflection of the sound ray on the sequence of surfaces.
- **Low pass filtering** to account for air absorption.

With respect to other methods that use a static Room Impulse Response filter, this approach **adapts dynamically** to the scene for a **more realistic and responsive reverb**.

Disadvantages and workarounds

Of course, the added value of dynamic reverb comes at a price. The **computational cost** for placing ISs and tracing sound ray paths hardly fits the real-time constraints of a simulation, but there's a few considerations that allow to partially work around the heavy computational intensity.

IS generation is only dependent on the room's boundaries and source location, while the path followed by rays is heavily dependent on the listener too.
This means that, **for static sound sources** and assuming the room's boundaries to also be static, **ISs can be created and positioned just at the beginning of the simulation** without any need to be generated again. Since IS generation is the most computationally expensive operation by far, this only leaves path tracing to be executed in real-time, a much more feasible operation.

The burdening process of IS creation remains an issue for dynamic sound sources and still affects the complexity of checking the sound rays. With a simulation **having N surfaces** and aiming at creating ISs up to the r **order**, the **total number of ISs will be in the order of $O(N^r)$**, the exponential growth of generated ISs not only makes handling dynamic sound sources incredibly difficult, but also leaves many paths to be checked constantly for static sound sources without any positive results.

The following section will cover **optimizations for the IS generation algorithm**, which **aim at culling the amount of ISs generated**, reducing the exponential growth of the problem with respect to maximum order of reflection.

Optimizations for IS generation

The complexity of the IS approach, generating image sources for each possible sequence of reflectors, is what **allows it to be so accurate**.

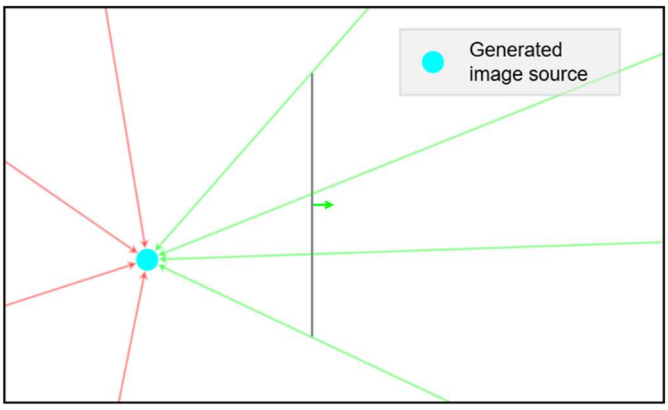
Intuitively, however, **many ISs and their children will never generate a valid sound ray**, independently of the listener's position. Optimizations aim at identifying all these ISs and avoid generating them, together with their entire subtree of ISs.

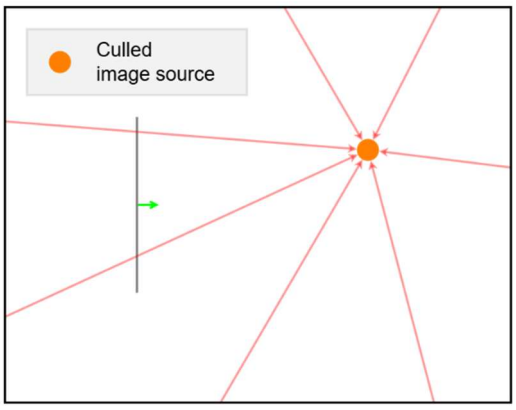
Specifically, these **optimizations revolve around the necessity of a ray to be traced towards the IS and collide with the front of its surface**: if it can be proven that, for a given IS, **no ray could be traced and hit its surface on the front**, then that **IS will never generate a valid path**. This also applies to all higher order ISs that would be generated from it, since for them to be valid the sound ray would need to respect the same condition.

Wrong side of reflector

First and simpler of the optimizations is the one **culling all ISs that would be positioned on the front side of their reflective surface**.

This is because, for a ray to hit the surface on the front while being traced to the IS, the latter would necessarily need to be positioned on the back side of the surface.

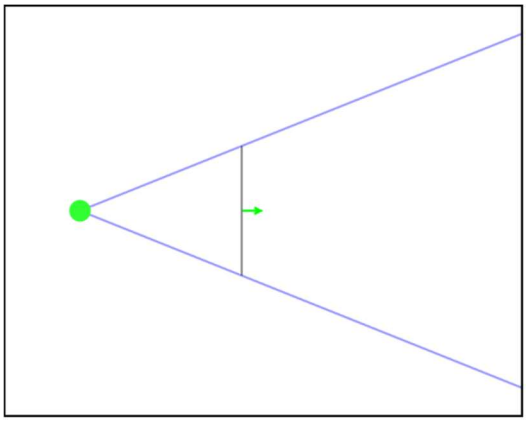




This way, by simply checking the IS's position with respect to the surface's normal, it's possible to prevent many ISs, along with their subtree, from being created.

Beam tracing

From the previous example, it can be noted that **there's only a limited range of directions from which a ray can be traced towards the IS and hit the correct surface on its front**.



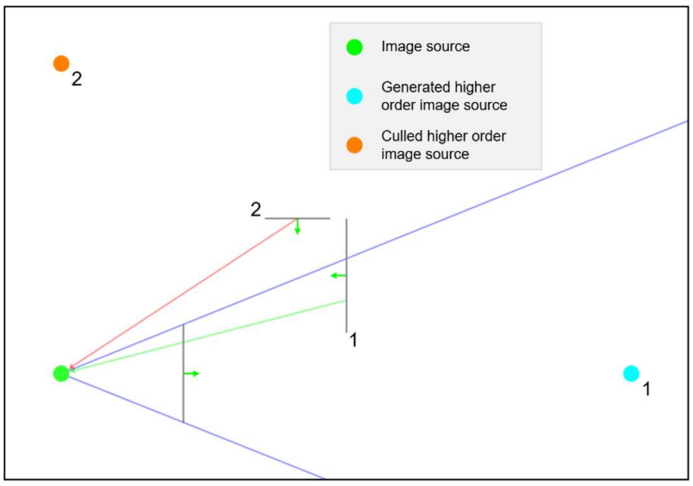
This range of directions is called beam, delimited in blue in the example on the left.

Again, for the sake of simplicity, every example up until now is in 2D. It is however important to note that, in a 3D simulation, a beam is identified by a frustum-like structure: a set of planes in space whose intersection encapsulates the beam. This will be discussed in more detail later, when writing about the implementation for these more advanced optimizations.

Given an image source i, for all its children ISs, during the tracing of the complete sound ray path, a part of the ray will need to be traced from the intersection with the child's surface all the way to the i image source and intersect with i's surface.

This means that **a valid ray will never be traced from a surface that falls outside of the beam of an IS**: there's no reason to generate a higher order IS by mirroring against that surface.

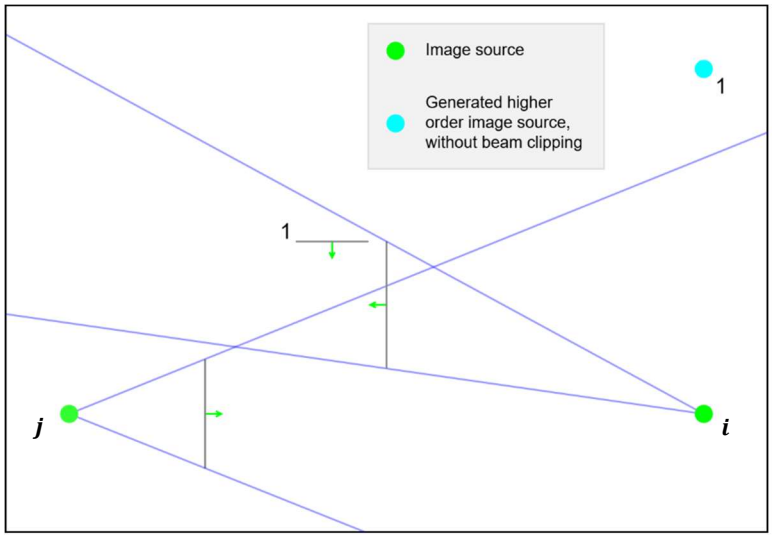
On the right, an example shows the utility of this optimization, as the second higher order IS surely doesn't need to be created, along with all its subtree of higher order ISs.



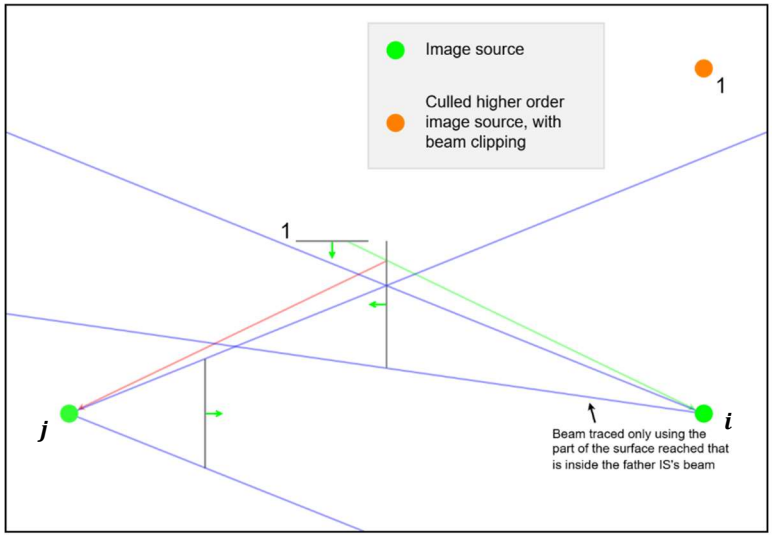
Beam clipping

When creating an IS *i*, from a lower order IS *j*, its beam is normally traced using its entire surface. This is actually not optimal because, for the final ray to be valid, the part of **ray traced towards *i* would need to intersect its surface on the portion that falls inside *j*'s beam**. Otherwise, the next ray traced towards j would surely fall outside its beam, invalidating the ray.

This condition can be applied by **tracing *i*'s beam from its surface after it's clipped by *j*'s beam**, so that **new ISs are only formed against surfaces that may reflect the sound ray on j's surface**.



In the example on the left, without beam clipping, a higher order IS is generated because it falls inside the beam traced from the entire surface of the IS on the right.



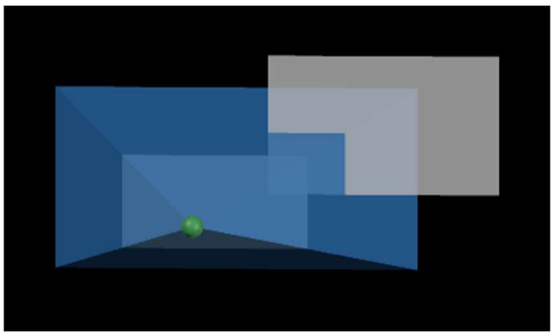
But it's easy to see here how all rays bouncing off surface 1 would never manage to reflect on all surfaces in the IS's sequence.

Beam clipping avoids the generation of such ISs, like in this example where surface 1 is outside the clipped beam.

Beam tracing and clipping in 3D

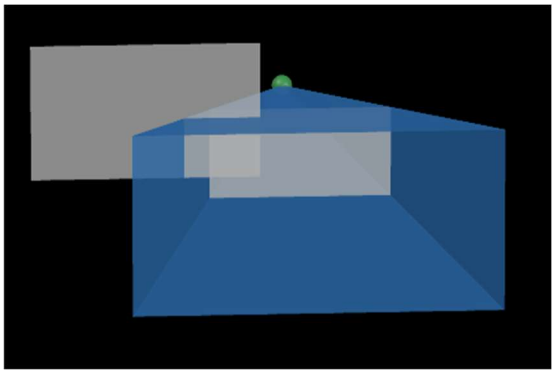
Adapting beam tracing to a 3D environment

In Unity, and in general in 3D, **a beam cannot be identified by a simple angle**.
In this project, in order to represent the set of directions that would form a valid ray, **a frustum-like structure is created for each IS**.



Planes are created by connecting the IS (the green sphere in the example) **to each edge of its surface**.

Each plane defines two semi-spaces: a valid one including the rest of the surface, and another excluding the surface.
The intersection of all valid semi-spaces forms the beam (displayed in blue in the example).



To implement the **beam tracing optimization**, from each IS, higher order ISs must only be generated against surfaces that fall inside this beam.

Furthermore, to implement **beam clipping**, the new IS needs to trace its beam only from the part of its surface that falls inside the previous IS's beam.
In the example shown on the right, where a second surface is intersected by the beam only in its corner, the new IS created from that surface should only use that section for its beam.

Convexity

For the frustum approach to work, **all surfaces to be projected need to be convex**: it would be impossible to represent the beam as an intersection of semi-spaces otherwise.
Thanks to the initial assumptions, **all the boundaries in the room are convex polygonal surfaces**, safe to work with using the frustum and perspective projection.

However, **beam clipping requires using a trimmed version of a surface for projection**, specifically the intersection of the original surface and the projection of the father IS's surface onto it.
To demonstrate the validity of the approach, it's **essential to prove that a clipped surface retains the property of convexity**.

Perspective projection preserves convexity (reference 1, section 4.2.4).

Thanks to this property, the **perspective projection of an IS's surface on the plane of another surface is sure to be convex**.

An intersection of convex sets is also convex (reference 2, section 1).

A clipped surface is the result of the intersection between the original surface and the projection of another surface, both of which are now proved to be convex: this proves that **the result also enjoys the property of convexity**.
By extension, clipped surfaces created by projecting another clipped surface are also convex.

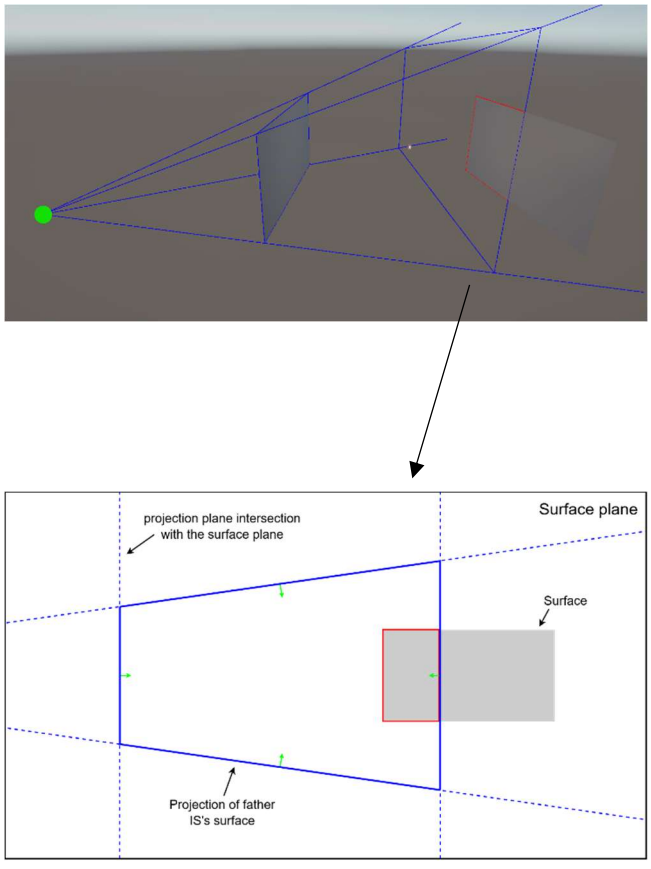
Geometrical implementation of beam tracing & clipping

All operations mentioned in the first sections of this document, like mirroring ISs along a surface or tracing a ray to check for a sound path, are easily performed operations that require basic vector algebra and built-in Unity methods.

This last part, however, is not so immediate: deciding whether a surface falls inside a series of semi-spaces requires some effort, as well as cutting out the parts that are outside them.
This section breaks down the process of **implementing these optimizations**, **reducing them to a series of simple geometrical operations**.

Perspective projection

Below is the **final result of beam tracing and clipping in Unity**, using perspective projection:



From the IS and its surface (the sphere and surface on the left), the frustum is created (the blue lines coming from the IS).

The frustum planes intersect with the opposing surface plane: the lines resulting from these intersections divide the surface plane into semi-planes, whose intersection defines the **perspective projection of the first surface onto the second** (the blue outline on the right).

The part of the new **surface inside the projection** (outlined in red) is going to be **used to trace the beam for the new IS**.

From now on, to better visualize and explain the process, **graphical examples will have this format: displaying the plane of the surface considered for generating a new IS**.

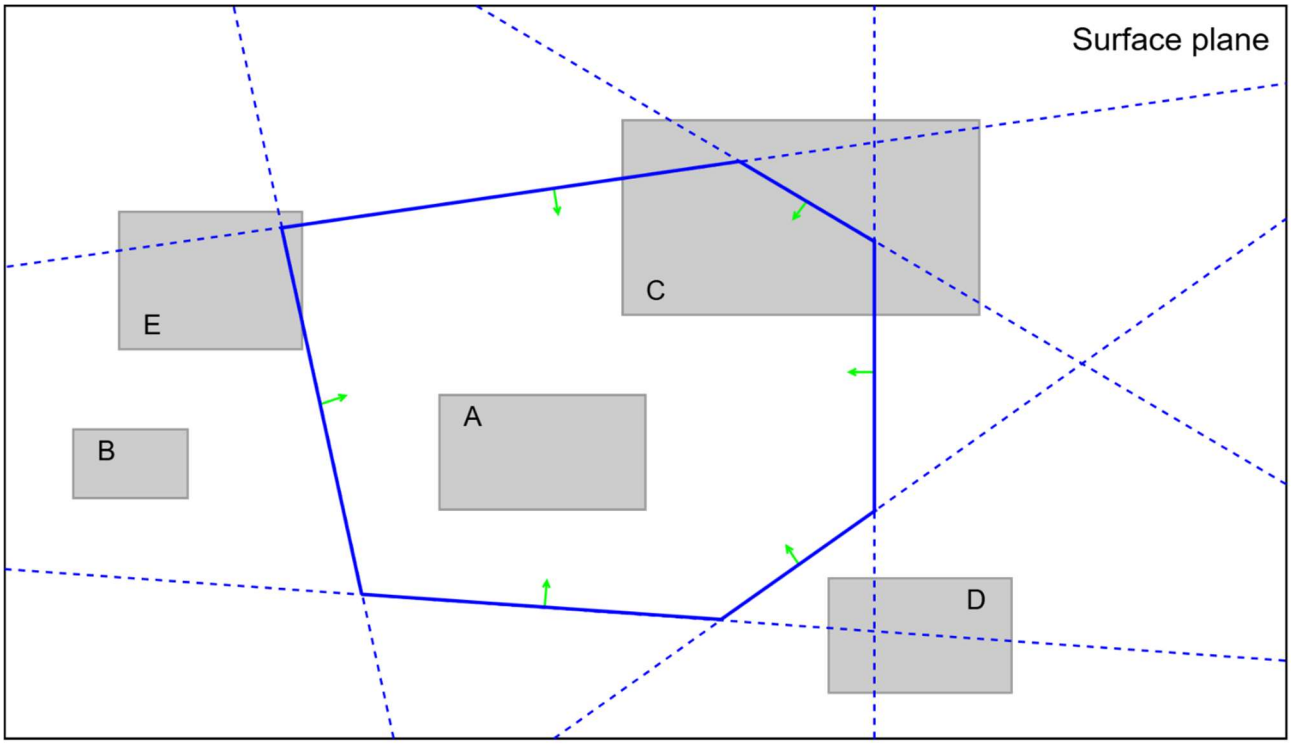
The intersections of the frustum planes with the surface plane are shown as blue dotted lines, while the full surface projection is shown as a blue polygon.

Finally, the surface section inside the projection is highlighted in red.

Inside or outside?

The seemingly simple problem of **understanding whether a surface falls inside the projection of another on its plane** is actually quite nuanced.

The example below, a single projection on four different surfaces, showcases notable situations:

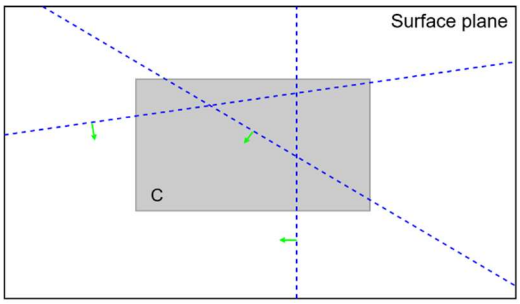


It's clear that **simply checking if at least one vertex of the surface is inside the frustum** (on the valid semi-plane of all the intersection lines in blue) **wouldn't suffice**: surface E is inside the frustum, but all of its vertices are outside.

In the same way, **checking for intersections with the projection lines is not accurate either**: surfaces A and B have no intersections, but one is inside the frustum, while the other outside. Similarly, surface D and C have intersections, but the former is excluded and the latter is not.

In this project, **the final algorithm to decide whether a surface is included in a beam is tied directly to the implementation of beam clipping**: the surface is first clipped by all projection lines, then a final check is carried out to see if it's inside the full beam's projection.

Clipping a surface

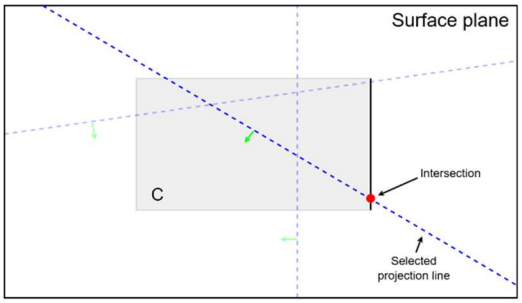


To show the process of clipping a surface, surface C from the example in the previous page is taken as a reference.

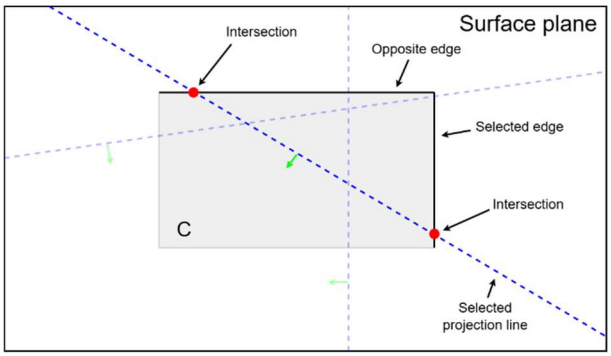
The final objective is to **reduce the surface to the portion included in all the valid semi-spaces**, indicated by the green vector on each line.

To fully clip the surface, the algorithm **cycles on all [projection line – surface edge] combinations**, **looking for intersections between edge and line**.
On the right is an example.

In case of intersection, the surface needs to be cut to discard the parts outside of the projection.



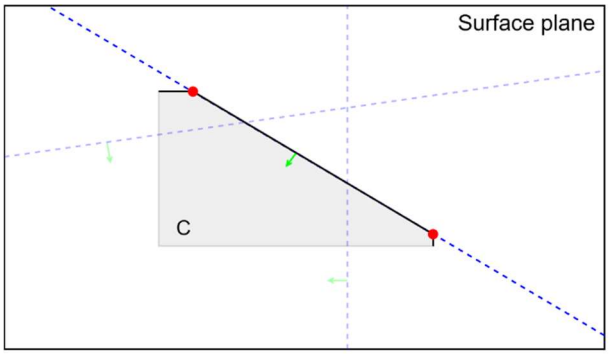
A line can intersect the edges of a connected and convex polygon in at most two points, if it were to intersect the edges in three or more points, it would mean that the line inside the polygon is not entirely included in it, contradicting the property of convexity. The exception where the line only touches the polygon on its edge is ignored and not considered an intersection by the algorithm.
As a result, the **possible cases for an intersection are reduced to two:**



1. Going back to the example above, once an intersection is detected on an edge, the edge that is connected to the vertex outside the valid semi-plane is also checked for collision against the same projection line.

This edge is referred to as "*adjacent edge*".

In case 1, **the line also intersects the adjacent edge and both intersection points are found**.

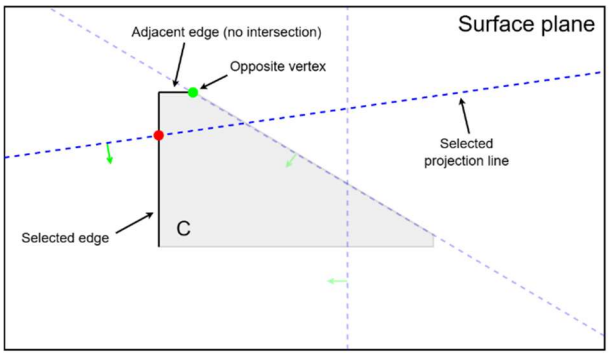


The resolution in case 1 is to **cut both edges up to the intersections**, then **create a new edge that connects them**.

This resolution **completely removes the surface section outside the projection line**: if this wasn't true, there would be more intersections, but there can only be two.

2. Picking up from the end of the first case, if **no intersection is found on the adjacent edge**, then it means the other intersection point is further along the polygon's boundary.

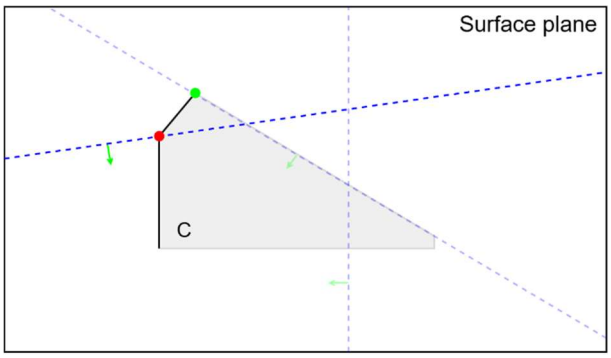
In this situation, the algorithm considers the vertex on the adjacent edge that isn't also connected to the selected edge, referring to it as "**opposite vertex**".



The resolution in case 2 is to **cut the selected edge up the intersection**, **discard the adjacent one**, then **create a new edge that connects intersection and opposite vertex**.

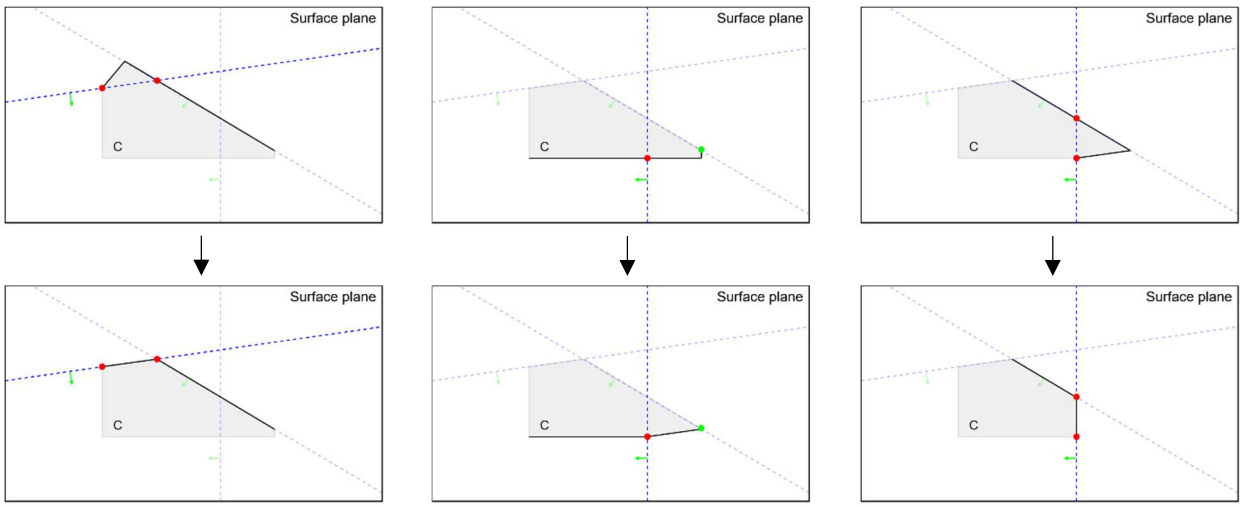
This resolution only **partially removes the surface section outside the projection line**: intersections with the projection line remain and will be handled by the next iterations.

The number of edges separating the two intersections is lowered by one, repeating the process ensures that **case 2 will eventually be reduced to case 1 and fully resolved**.

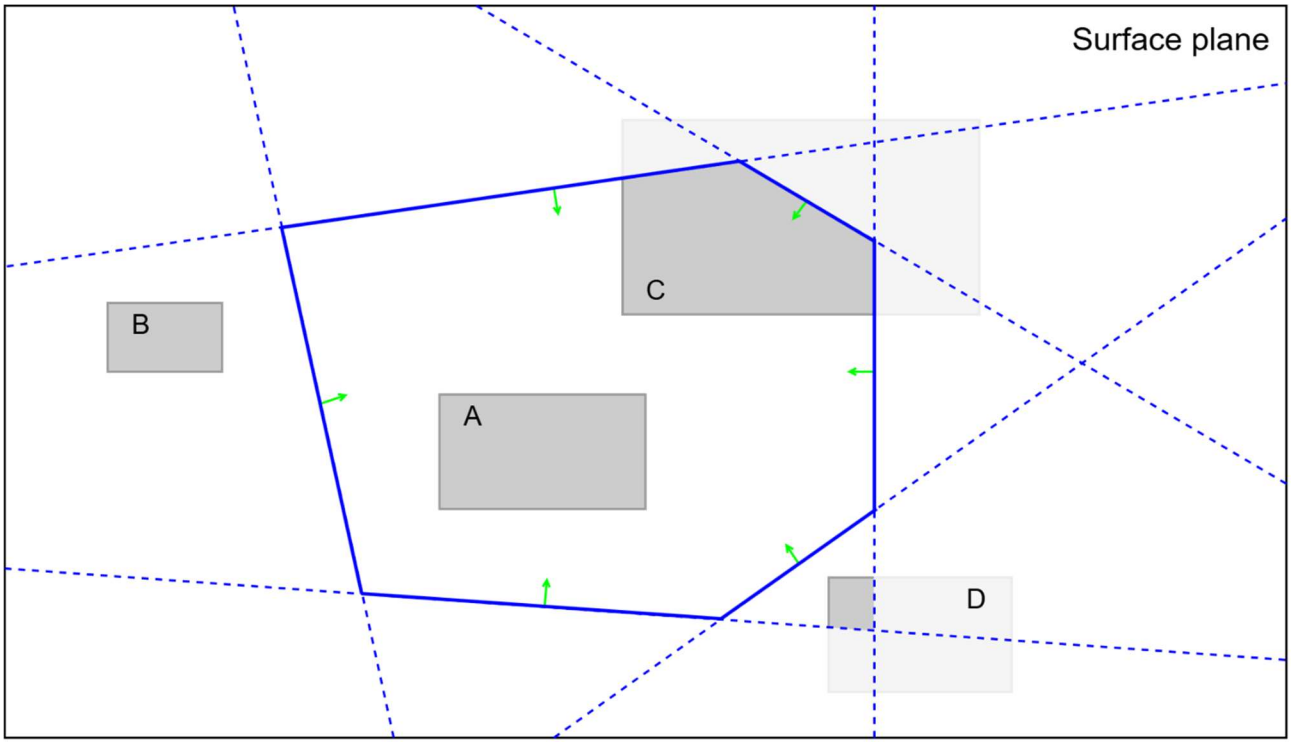


The algorithm keeps checking all possible intersections between line and edge, until all intersections have been handled. **When there are no more intersections**, the **remaining polygon is the clipped surface**, falling inside all the projection lines that were intersecting it.

Below, the remainder of the full clipping process of C is shown:



Deciding inclusion after clipping

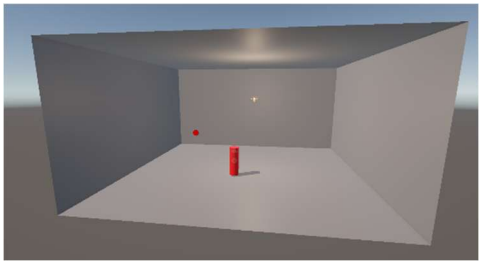


By applying clipping, like shown above, **every [projection – surface] configuration is reduced to one of the four cases above:**

1. **No intersections found in clipping**. Since the surface is connected and convex, it would be impossible for two points inside the surface to be on different sides of the frustum: if one was outside and the other inside, the line connecting them would be contained in the surface (because of convexity) and this line would at some point transition from outside to inside (or vice versa), creating an intersection and contradicting the original statement. This leaves two options:
    - **A. Surface entirely inside frustum**.
    - **B. Surface entirely outside frustum**.
2. **Intersections found and clipped**. After clipping, the situation is exactly the same described in point 1: no intersections are left, which again leaves two options:
    - **C. Remainder of the surface is entirely inside frustum**; the outside sections were cut.
    - **D. Remainder of the surface is entirely out of frustum**; the entire surface also was.

This means that after clipping, independently of case 1 or 2, **options a-b or c-d can be verified by checking a random point in the surface**, one of its vertices for example: **if that point is inside the frustum** (i.e. on the right side of all projection lines), then the **surface is inside the beam** and **its clipping was already computed** and ready to be used for the next beam tracing.
Otherwise, the **surface is completely outside the beam**.

Benchmarks



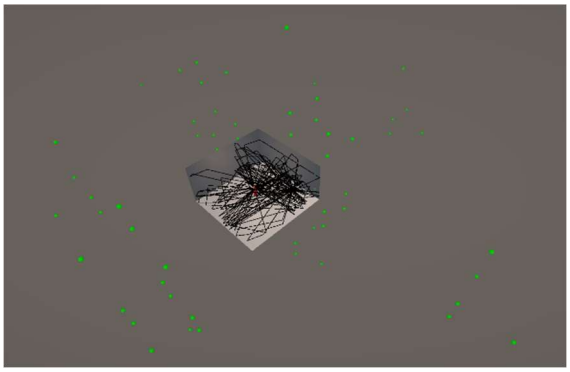
Given the shoebox room shown on the right, the **following are performance comparisons for different maximum orders of reflection and enabled optimizations**.
The number of valid sound rays from source to listener is only dependent on the chosen order of reflection, as optimizations don't reduce accuracy.
Time measurements are obtained by running the generation algorithms five times and reporting their average duration:

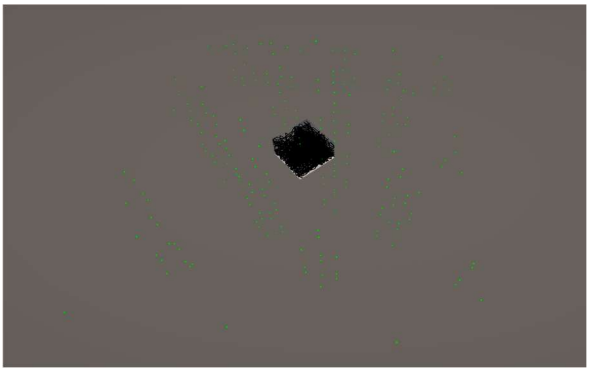
Maximum order of reflection: 3
Number of valid rays: 62

- **No optimizations**
  Generated ISs: 186
  Time to generate ISs: 0.58 ms
  Time to compute sound rays: 0.67 ms
- **Wrong side of reflector**
  Generated ISs: 162
  Time to generate ISs: 0.54 ms
  Time to compute sound rays: 0.66 ms
- **Wrong side of reflector + beam tracing**
  Generated ISs: 150
  Time to generate ISs: 1.54 ms
  Time to compute sound rays: 0.58 ms
- **Wrong side of reflector + beam tracing + clipping**
  Generated ISs: 150
  Time to generate ISs: 1.51 ms
  Time to compute sound rays: 0.62 ms

Maximum order of reflection: 5
Number of valid rays: 230

- **No optimizations**
  Generated ISs: 4.686
  Time to generate ISs: 12.15 ms
  Time to compute sound rays: 12.12 ms
- **Wrong side of reflector**
  Generated ISs: 2.190
  Time to generate ISs: 8.36 ms
  Time to compute sound rays: 7.05 ms
- **Wrong side of reflector + beam tracing**
  Generated ISs: 1.546
  Time to generate ISs: 18.71 ms
  Time to compute sound rays: 5.26 ms
- **Wrong side of reflector + beam tracing + clipping**
  Generated ISs: 1.214
  Time to generate ISs: 16.93 ms
  Time to compute sound rays: 4.34 ms



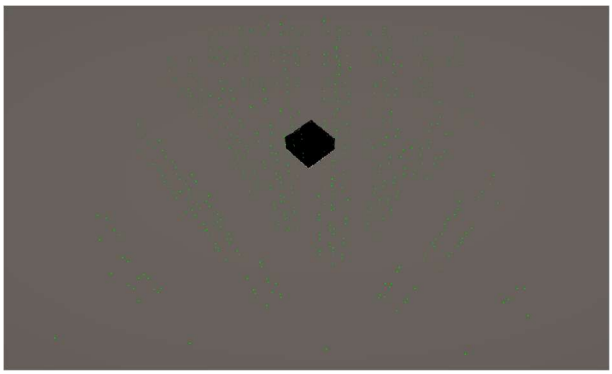


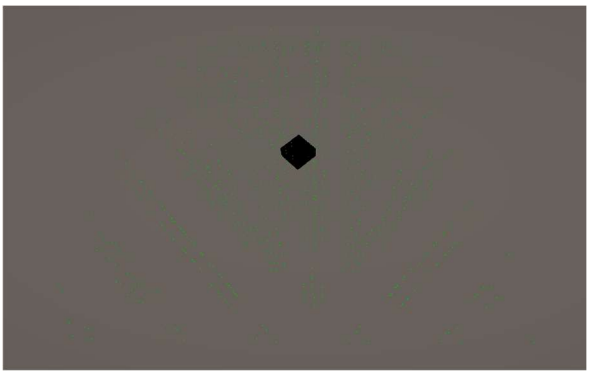
Maximum order of reflection: 7
Number of valid rays: 574

- **No optimizations**
  Generated ISs: 117.186
  Time to generate ISs: 0.43 **s**
  Time to compute sound rays: 0.28 **s**
- **Wrong side of reflector**
  Generated ISs: 23.226
  Time to generate ISs: 101 ms
  Time to compute sound rays: 69 ms
- **Wrong side of reflector + beam tracing**
  Generated ISs: 11.754
  Time to generate ISs: 157 ms
  Time to compute sound rays: 38 ms
- **Wrong side of reflector + beam tracing + clipping**
  Generated ISs: 5.397
  Time to generate ISs: 98 ms
  Time to compute sound rays: 20 ms

Maximum order of reflection: 10
Number of valid rays: 1.560

- **No optimizations**
  Generated ISs: 14.648.436
  Time to generate ISs*: 71.7 **s**
  Time to compute sound rays*: 36 **s**
- **Wrong side of reflector**
  Generated ISs: 684.084
  Time to generate ISs: 3.53 **s**
  Time to compute sound rays: 1.94 **s**
- **Wrong side of reflector + beam tracing**
  Generated ISs: 198.624
  Time to generate ISs: 3.13 **s**
  Time to compute sound rays: 0.66 **s**
- **Wrong side of reflector + beam tracing + clipping**
  Generated ISs: 29.137
  Time to generate ISs: 0.63 **s**
  Time to compute sound rays: 108 ms





It can be noted that, for high orders of reflections, the optimizations provide an essential cut to the exponentially growing set of generated ISs, reducing time by a significant margin.

As expected, the ***wrong side of reflector*** **optimization is incredibly cheap and always provides a benefit to the execution time of both algorithms** for all configurations.

The **two beam optimizations** are far more complex: while they always reduce the time for sound ray computation, at low orders of reflection (<= 5) they **can slow down the IS creation algorithm**.

This problem is negligible, however, since **the benefits at high orders easily outweigh the small drawbacks at low orders**: in the configurations with maximum order 10, **time to create ISs is reduced by** a stunning **99.1%** with all three optimizations enabled, while **ray evaluation** fares even better with its computation **time cut by 99.7%**.

* For the sake of my pc, these configurations were only run once, instead of five times

Final considerations

The ISs reverb method is surely an interesting approach to geometrical acoustics:
allowing to define reverb based on the configuration of the room allows simulations to be **dynamic and adaptive**, greatly enhancing immersion.

As seen with the final benchmarks, the algorithms required for IS generation and the tracing of sound rays can be quite **challenging to fully implement for real-time simulations**.
The approach proves with no doubt, however, to be a more than **fitting solution for reverb with static sound sources and small environments**: the computational complexity to verify sound rays grows much slower than that of IS generation and, for orders < 7, already displays an acceptable *time / number of reflections* ratio.

Given the option to only generate ISs at the beginning of the simulation, the strength of this method for reverberation is undeniable; even more so if **a hybrid approach is created**:
using a more lightweight reverb method to cover for complex environments in a large area, arguably the ISs reverb's greatest weakness, while **using ISs for the local area around the listener**.

References

1) Lecture on convexity - Carnegie Mellon University

2) Properties of convex sets - Computer Science & Engineering IIT Kanpur