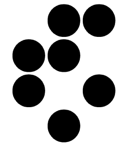
Institut "Jožef Stefan", Ljubljana, Slovenija



IJS Delovno poročilo
DP-12505

Reduction of Diffraction Edges in RTX

Igor Ozimek

Ljubljana, March 2018

Contents

1. INTRODUCTION 1

2. RADIO PROPAGATION SCENE DESCRIPTION 2

2.1. WALLS 2

2.2. DIFFRACTION EDGES 4

3. EDGE-ONLY APPROACH TO EDGE REDUCTION 6

3.1. REMOVE TOP AND BOTTOM EDGES 8

3.2. EDGE SUBDIVISION 9

3.3. EDGE REDUCTION 10

3.4. REMOVING SHORT EDGES 13

3.5. CLEANING UP 15

3.6. IMPLEMENTATION AND PROCESSING TIMES 15

4. EDGE–WALL APPROACH TO EDGE REDUCTION 17

4.1. REDUCEEDGES CLASS – USAGE AND DATA STRUCTURES 17

4.2. VISUALIZATION 18

4.3. EDGE REDUCTION ALGORITHM 20

4.4. EXECUTION TIME 21

Figures

Figure 1: Test scene (with ceiling removed) 2
Figure 2: Basic wall 3
Figure 3: Wall's edges and side vectors 5
Figure 4: Diffraction edge angle definition 5
Figure 5: Eight long edges of each wall and hole as diffraction edges (in red) 6
Figure 6: Redundant edges – enlarged view (some edges are visible only as thin red lines) 7
Figure 7: Redundant edges – enlarged view 2 7
Figure 8: The result of edge-only approach reduction 8
Figure 9: Edges after removal of two levels of top and bottom edges 9
Figure 10: An example of subdivision of partly overlapping edges (coinciding, but shown with horizontal displacement for visibility) 10
Figure 11: An example of edge reduction of partly overlapping edges (coinciding, but shown with horizontal displacement for visibility) 10
Figure 12: Two walls – ground plan 11
Figure 13: Two walls, normally perpendicularly aligned 11
Figure 14: Two walls, aligned obliquely (overlapping) 12
Figure 15: Edge reduction with bottom & top preprocessing 12
Figure 16: Edge reduction without bottom & top preprocessing 13
Figure 17: Final result, with bottom & top preprocessing 14
Figure 18: Final result, without bottom & top preprocessing 15
Figure 19: Non-transparent rendering 19
Figure 20: Non-transparent rendering with ceiling removed from processing 19
Figure 21: Partially transparent rendering 20
Figure 22: Edge subdivision 21

Tables

Table 1: Execution times of edge-only edge reduction, 'debug' version 16
Table 2: Execution times of edge-only edge reduction, 'release' version 16

1. Introduction

This report describes algorithms for reduction of diffraction edges, developed for use with the radio frequency ray tracing application developed in the ARRS project “L2-7664 - Advanced Ray-Tracing Techniques in Radio Environment Characterization and Radio Localization”. The ray tracing application makes use of Nvidia CUDA and OptiX engine and runs on Nvidia GPU cards. It performs radio propagation computation in a scene that has been designed with a custom developed scene design editor. The scene consists of walls (including floors and ceilings) with optional openings for windows and doors. Each wall has edges, which are included in the complete scene, however not all of these edges actually act as diffraction edges. The task of the diffraction-edge-reduction algorithm presented in this report is to remove superfluous edges and keep only those that actually cause radio wave diffraction.

In the following three chapters, radio propagation scene description will be explained first, followed by presentation of two different edge reduction approaches, their corresponding algorithms and impementations.

2. Radio Propagation Scene Description

The scene is built of walls (rectangular cuboids or boxes), which optionally contain openings (holes) for windows or doors. Floors and ceilings are also designed as walls that are positioned horizontally, as well as some other structures (e.g. tables). Complex wall shapes (floor, ceiling …) can be created by suitably placing openings, which are rectangles themselves, but can be rotated arbitrary within the wall, can overlap each other, and can extend past the wall borders.

Figure 1 shows a scene that has been used as the test scene during the development of the diffraction edge reduction algorithms described in the next two chapters. The ceiling is part of the scene, however in the figure it is made transparent so that the interior space is visible. Diffraction edges are not shown in this picture.

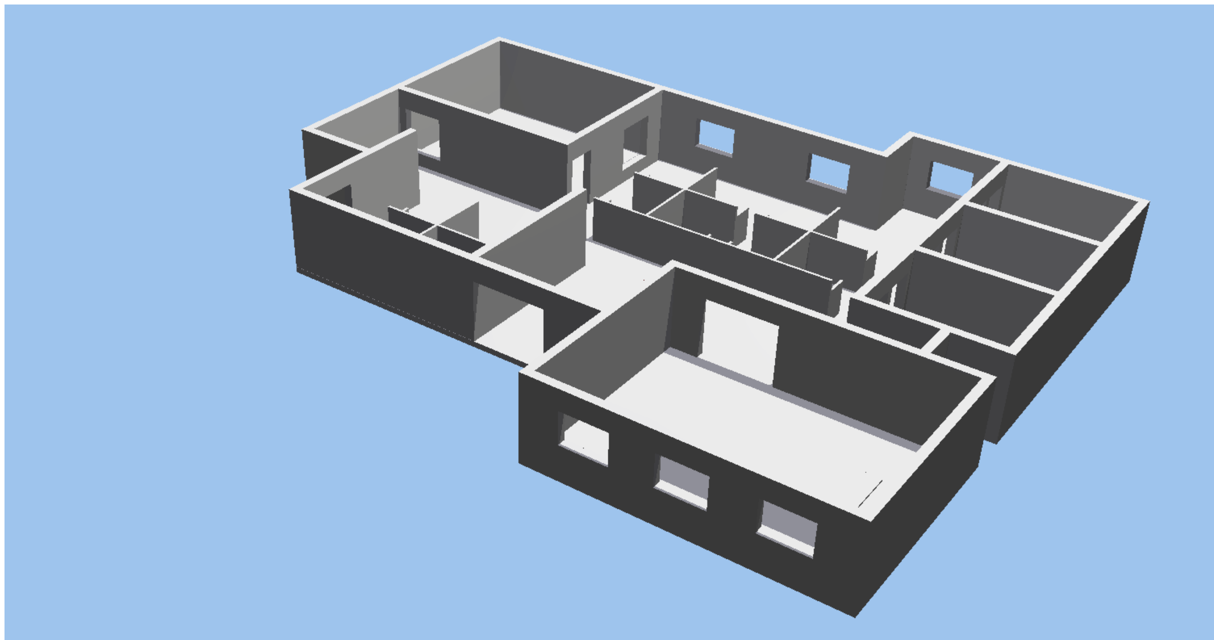


Figure 1: Test scene (with ceiling removed)

2.1. Walls

Figure 2 shows a basic wall (without openings). Each wall is defined by its length, thickness and height (in m). Its final position is defined by rotation and translation parameters, which are compatible with GLM (OpenGL Mathematics) processing. The numbers displayed are indices (sequential numbers) of the cuboid's faces as used by the edge reduction algorithm.

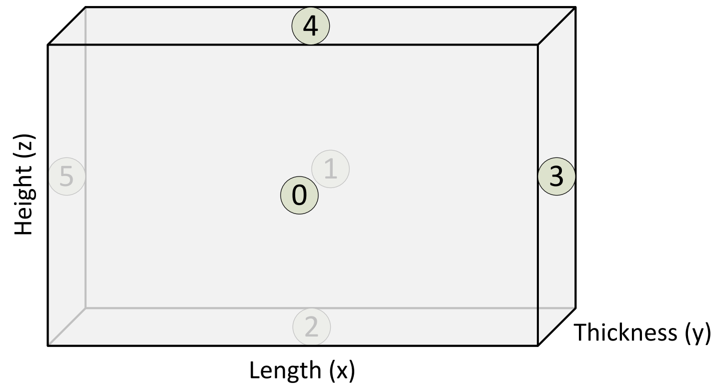


Figure 2: Basic wall

The scene editing application creates a scene description file in the XML format describing all the walls. Below is an example of this file for a simple scene containing only a single wall (2 m long, 10 cm thick and 1.6 m high), without rotation and with a small translation.

For each wall, the description contains also its triangulated surfaces (six of them for a simple wall). The surface ids in the XML file below correspond to the surface indices in Figure 2. Each surface is represented by a number of triangles; in our simple example, there are only simple rectangular surfaces and each is represented by two triangles. Triangulated description becomes more elaborate for walls with openings.

Each triangle is represented by three consecutive points, so each such triplet of points can be considered an independent group with index wrap-around when appropriate (e.g. for definition of triangle's edges). A point is marked with *e='1'* if the edge from the previous point (using wrap around) coincides with the cuboid's edge; the rest are the surface's internal (triangles') edges generated by the triangulation algorithm.

```
<?xml version="1.0" encoding="UTF-8"?>
<architecture>
   <scene>
         <wall>
               <material> 3 1 </material>
               <dimension> 2.00000000 0.10000000 1.60000000 </dimension>
               <rotation> 0.00000000 0.00000000 0.00000000 </rotation>
               <translation> -0.33500000 0.00000000 0.00000000 </translation>
               <separating_floors> 0 </separating_floors>
               <floor> 0 </floor>
               <surfaces>
                     <surface id="0">
                           <point e="0"> 2.00000000 0.00000000 1.60000000 </point>
                           <point e="1"> 0.00000000 0.00000000 0.00000000 </point>
                           <point e="1"> 2.00000000 0.00000000 0.00000000 </point>
                           <point e="0"> 0.00000000 0.00000000 0.00000000 </point>
                           <point e="1"> 2.00000000 0.00000000 1.60000000 </point>
                           <point e="1"> 0.00000000 0.00000000 1.60000000 </point>
                     </surface>
                     <surface id="1">
                           <point e="0"> 0.00000000 0.10000000 1.60000000 </point>
                           <point e="1"> 2.00000000 0.10000000 0.00000000 </point>
                           <point e="1"> 0.00000000 0.10000000 0.00000000 </point>
                           <point e="0"> 2.00000000 0.10000000 0.00000000 </point>
                           <point e="1"> 0.00000000 0.10000000 1.60000000 </point>
                           <point e="1"> 2.00000000 0.10000000 1.60000000 </point>
                     </surface>
                     <surface id="2">
                           <point e="0"> 2.00000000 0.10000000 0.00000000 </point>
                           <point e="1"> 0.00000000 0.00000000 0.00000000 </point>
                           <point e="1"> 0.00000000 0.10000000 0.00000000 </point>
                           <point e="0"> 0.00000000 0.00000000 0.00000000 </point>
                           <point e="1"> 2.00000000 0.10000000 0.00000000 </point>
                           <point e="1"> 2.00000000 0.00000000 0.00000000 </point>
                     </surface>
```

```
                    <surface id="3">
                          <point e="0"> 2.00000000 0.10000000 0.00000000 </point>
                          <point e="1"> 2.00000000 0.00000000 1.60000000 </point>
                          <point e="1"> 2.00000000 0.00000000 0.00000000 </point>
                          <point e="0"> 2.00000000 0.00000000 1.60000000 </point>
                          <point e="1"> 2.00000000 0.10000000 0.00000000 </point>
                          <point e="1"> 2.00000000 0.10000000 1.60000000 </point>
                    </surface>
                    <surface id="4">
                          <point e="0"> 2.00000000 0.00000000 1.60000000 </point>
                          <point e="1"> 0.00000000 0.10000000 1.60000000 </point>
                          <point e="1"> 0.00000000 0.00000000 1.60000000 </point>
                          <point e="0"> 0.00000000 0.10000000 1.60000000 </point>
                          <point e="1"> 2.00000000 0.00000000 1.60000000 </point>
                          <point e="1"> 2.00000000 0.10000000 1.60000000 </point>
                    </surface>
                    <surface id="5">
                          <point e="0"> 0.00000000 0.10000000 1.60000000 </point>
                          <point e="1"> 0.00000000 0.00000000 0.00000000 </point>
                          <point e="1"> 0.00000000 0.00000000 1.60000000 </point>
                          <point e="0"> 0.00000000 0.00000000 0.00000000 </point>
                          <point e="1"> 0.00000000 0.10000000 1.60000000 </point>
                          <point e="1"> 0.00000000 0.10000000 0.00000000 </point>
                    </surface>
              </surfaces>
        </wall>
  </scene>
</architecture>
```

When a wall contains one or more holes, there are additional tag(s) ‘hole’ for each hole, defining its parameters – 2D size rotation and translation (position) relative to the wall:

```
...
        <wall>
              <material> 0 0 </material>
              <dimension> 8.50000000 0.30000000 2.80000000 </dimension>
              <rotation> -0.00000000 0.00000000 3.14159265 </rotation>
              <translation> -3.33500000 -6.20000001 0.00000000 </translation>
              <separating_floors> 0 </separating_floors>
              <floor> 0 </floor>
              <hole>
                    <dimension> 1.60000000 1.10000000 </dimension>
                    <rotation> 0.00000000 </rotation>
                    <translation> 1.05000000 1.00000000 </translation>
              </hole>
              <surfaces>
...
```

The ‘surfaces’ information is modified as necessary: for each non-overlapping hole, four surfaces are added (penetrating the wall), and the main two wall’s surfaces (id “0” and “1”) are modified accordingly and described with a more complex triangle pattern (instead of only two symmetric triangles generated for a rectangle).

The list of wall edges (which are candidates for diffraction edges) could be generated directly from the cuboids representing the wall and its holes. However, this would become a complex task in case of overlapping holes or holes extending past the wall boundaries, therefore the edges are rather extracted from the triangulated surfaces’ description.

2.2. Diffraction edges

Each wall (or a non-overlapping hole) has twelve edges, which are candidates for diffraction edges in the final scene. However, the four edges representing thickness are (usually) short, their diffraction effect is therefore considered negligible and they are ignored, simplifying further processing. Figure 3 shows the remaining eight edges colored in red. For diffraction calculations, not only the edge itself is important but also position of the cuboid’s

planes (faces) creating the edge. In Figure 3 they are marked with arrows (vectors) and also with two red bands along each edge.

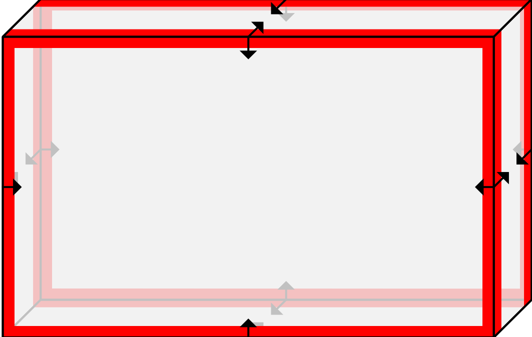


Figure 3: Wall's edges and side vectors

In principle, a diffraction edge is only a line segments and not a vector. However, to properly define the edge's diffraction angle (the angle formed by its adjacent faces) the sequence of both points and both side vectors is important, therefore we define edges as vectors. Looking in the direction along the edge vector (with the edge vector directed away from the observer, from P0 to P1 in Figure 4), the diffraction angle is defined as the angle from the first side vector (S0) to the second one (S1) measured in the right (clockwise) direction – right-hand (screw) rule, Figure 4.

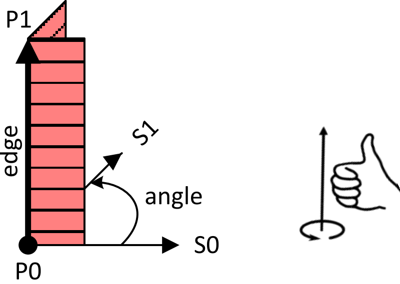


Figure 4: Diffraction edge angle definition

The above eight edges of each wall (and also from any optional holes in the wall) are only candidates for diffraction edges. In the final scene, not all of them actually act as diffraction edges, so some additional edge-reduction procedure is required to reduce the complete set of edges to only those actually causing diffraction. This will be described in chapters 3 and 4.

3. Edge-only approach to edge reduction

Figure 5 shows the diffraction edges (in red) as defined by the scene design editor (eight edges of each wall and each hole, as described previously). Not all of them actually cause diffraction. If diffraction is computed on all these edges, the computation time is increased, and the results are not accurate. Edges are shown as 'L' profiles, similarly as in Figure 3, but with a small displacement away from the edge for better visibility. When an edge of one wall aligns with a face of another wall, the 'L' shape is mostly buried inside the second wall, with only a small portion visible outside the wall as a red line. Traces of edges can also be seen on top of the table (created as a thin horizontal wall) in the large room since the table is thin enough for the displaced L shapes to just reach its surface.

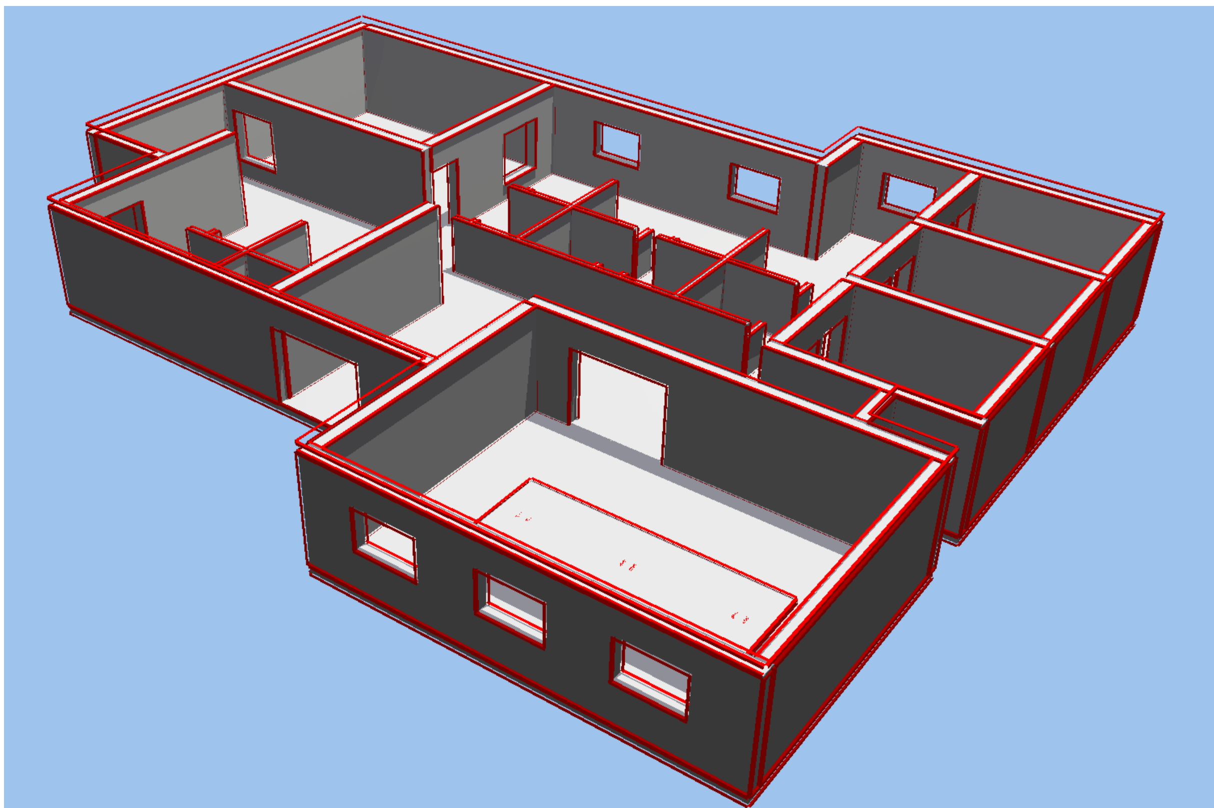


Figure 5: Eight long edges of each wall and hole as diffraction edges (in red)

Figures 6 and 7 show redundant edges in more detail. Generally, redundant edges (that do not cause diffraction) fall into two categories:

- Pairs of edges (theoretically also triplets, etc.), where edges of two cuboids (or theoretically more cuboids) are aligned and the resulting angle of the cuboids' faces is $\leq 180°$. In Figures 5, 6 and 7, we can see such vertical edges with corresponding cuboid face angles of exactly 180°.
- Cuboid edges that lie on a face of another cuboid (or possibly inside another cuboid – the existing scene editor application allows such design). Examples of such edges are horizontal edges of vertical walls on the floor and ceiling, and vertical edges in the corners of the rooms where a vertical wall connects to another wall, usually perpendicularly (see Figures 5, 6 and 7).

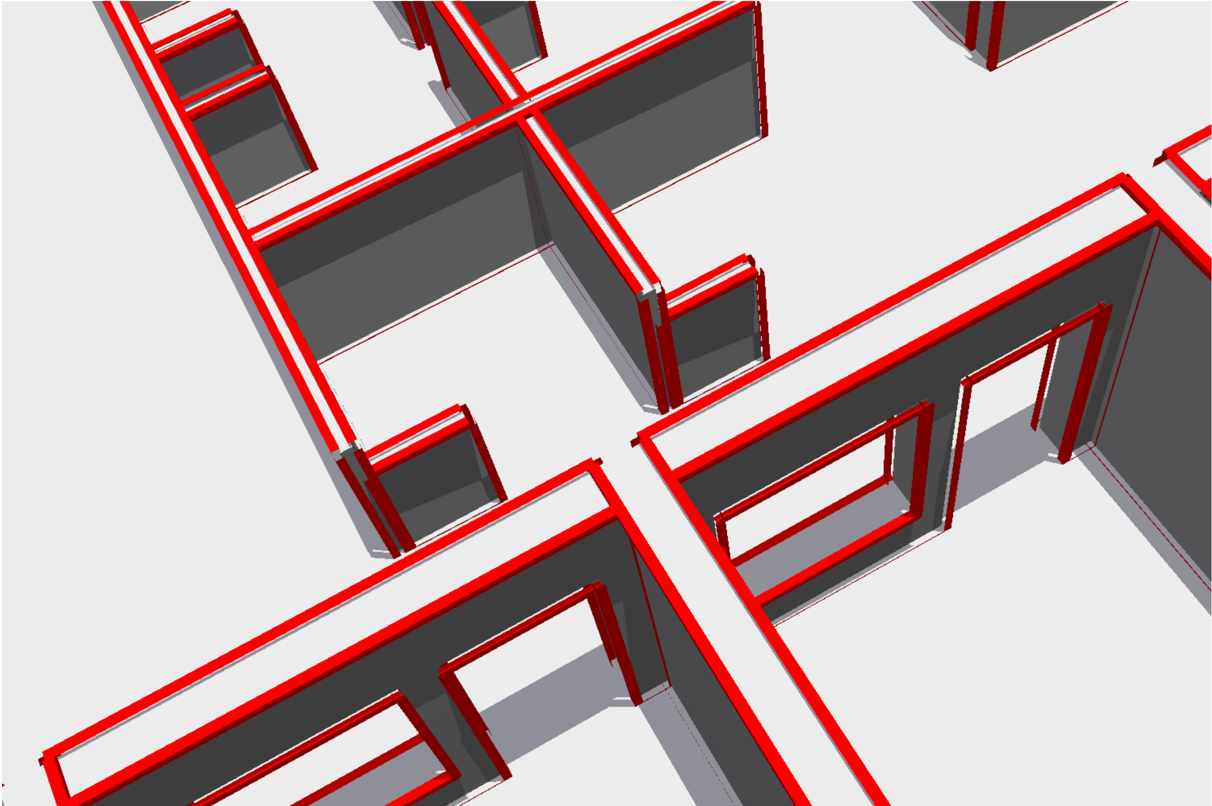


Figure 6: Redundant edges – enlarged view (some edges are visible only as thin red lines)

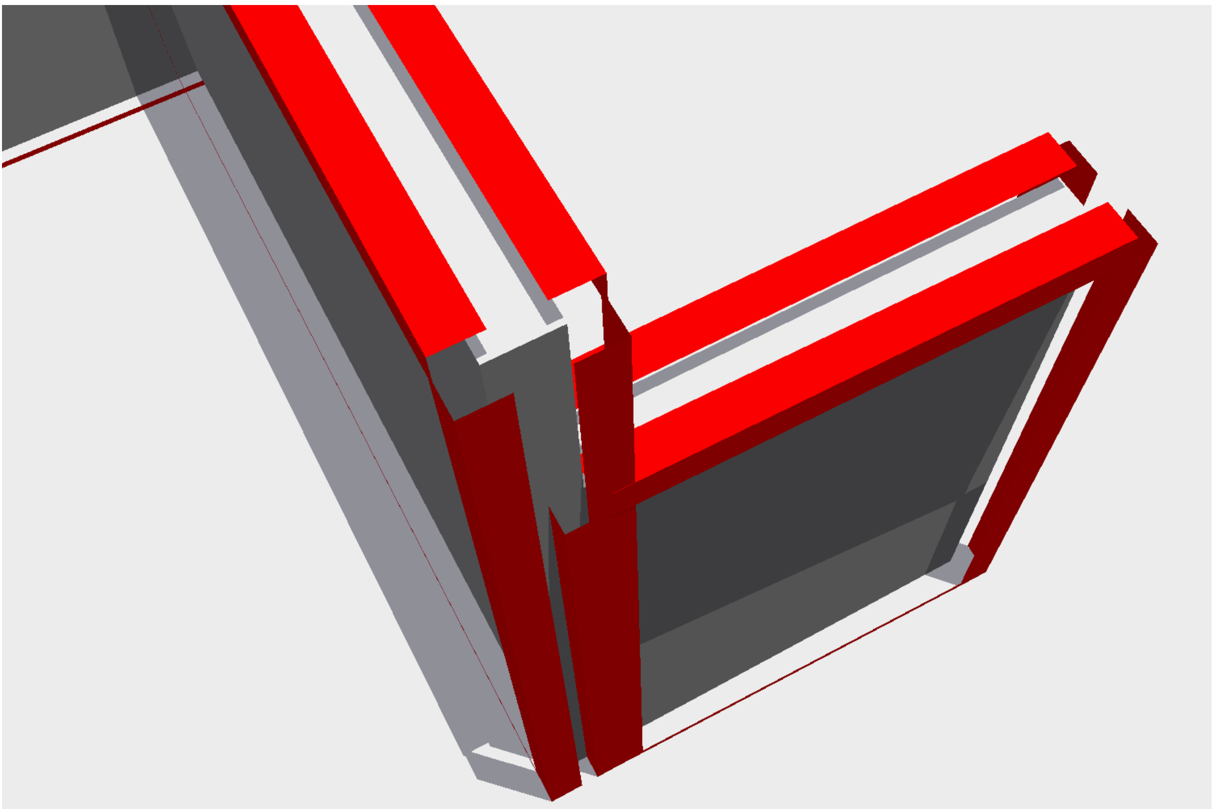


Figure 7: Redundant edges – enlarged view 2

Figure 8 shows the result of edge reduction that can be achieved by the edge-only reduction algorithm together with some additional (pre- or post-) processing, as described later in this chapter. The additional processing removes redundant horizontal edges but it cannot remove vertical edges in the corners of vertical walls, which can still be seen in Figure 8 as vertical red lines, as well as some other edges of this type.

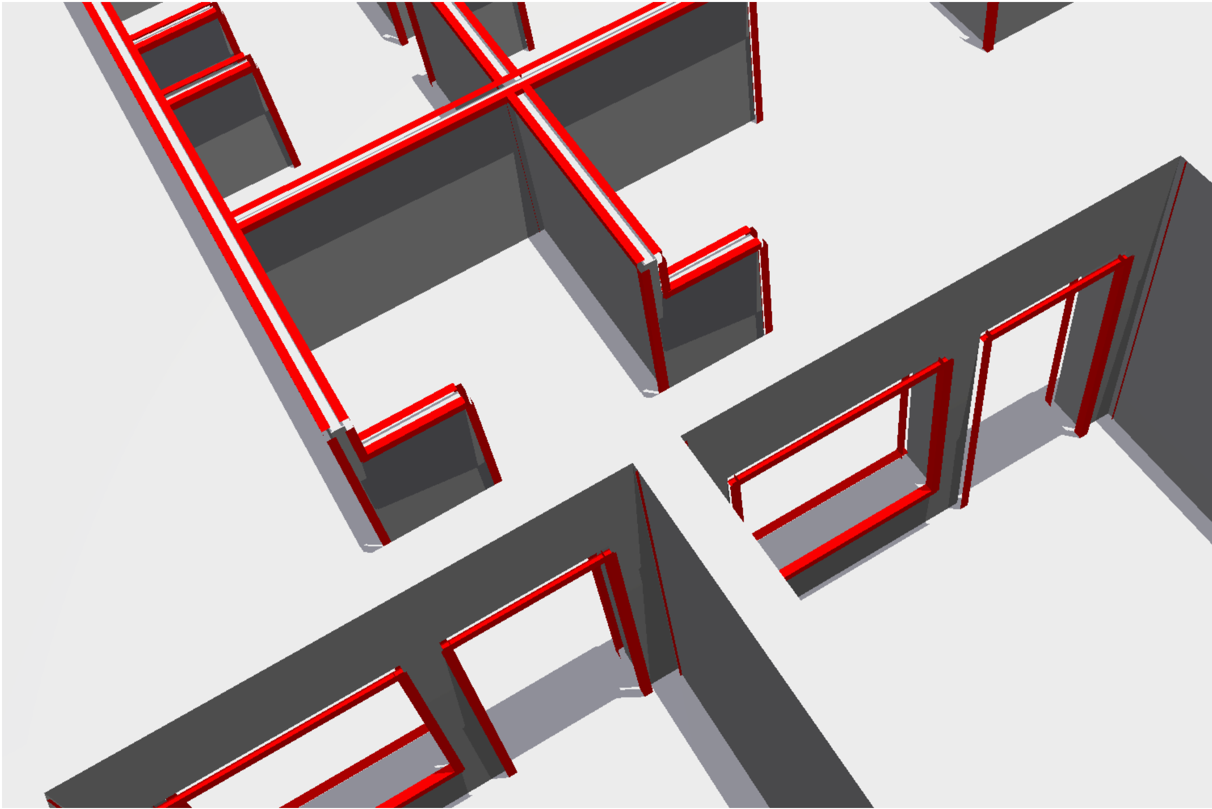


Figure 8: The result of edge-only approach reduction

The edge-only algorithm processes only the interactions between the edges and ignores information about the walls forming the edges (specifically, the wall's faces). This approach can eliminate many but not all of the non-diffraction edges. It can eliminate edges that are aligned to each other, but not those aligning with a face of another wall.

We envisioned this approach as a simple first step, which would simplify the task for the second step algorithm removing the rest of redundant edges. As it turned out, this first step is not necessary, as the second algorithm can do the job of removing all non-diffraction edges with comparable complexity and efficiency (execution time). The developed algorithm, implementation and results for the edge-only approach are nevertheless presented in this chapter.

In the following subsections, individual steps (implemented as C functions) of the edge-only approach are described. The last subsection gives basic information about the implementation and an analysis of processing times.

3.1. Remove top and bottom edges

These two functions remove top-most and bottom-most (horizontal) edges. Both are executed twice. The first run removes external floor or ceiling edges, which are strictly speaking correct diffraction edges but irrelevant for radio propagation inside the flat. The

second run removes horizontal edges at the interior floor and ceiling levels, which do not cause diffraction. This two functions are very simple and fast, with computation complexity $O(n)$ (where n is the number of edges) but are applicable only in simple cases with single-level horizontal floors and ceilings.

Figure 9 shows edges remaining after running these two functions twice:

```
RemoveBottomEdges();
RemoveBottomEdges();
RemoveTopEdges();
RemoveTopEdges();
```

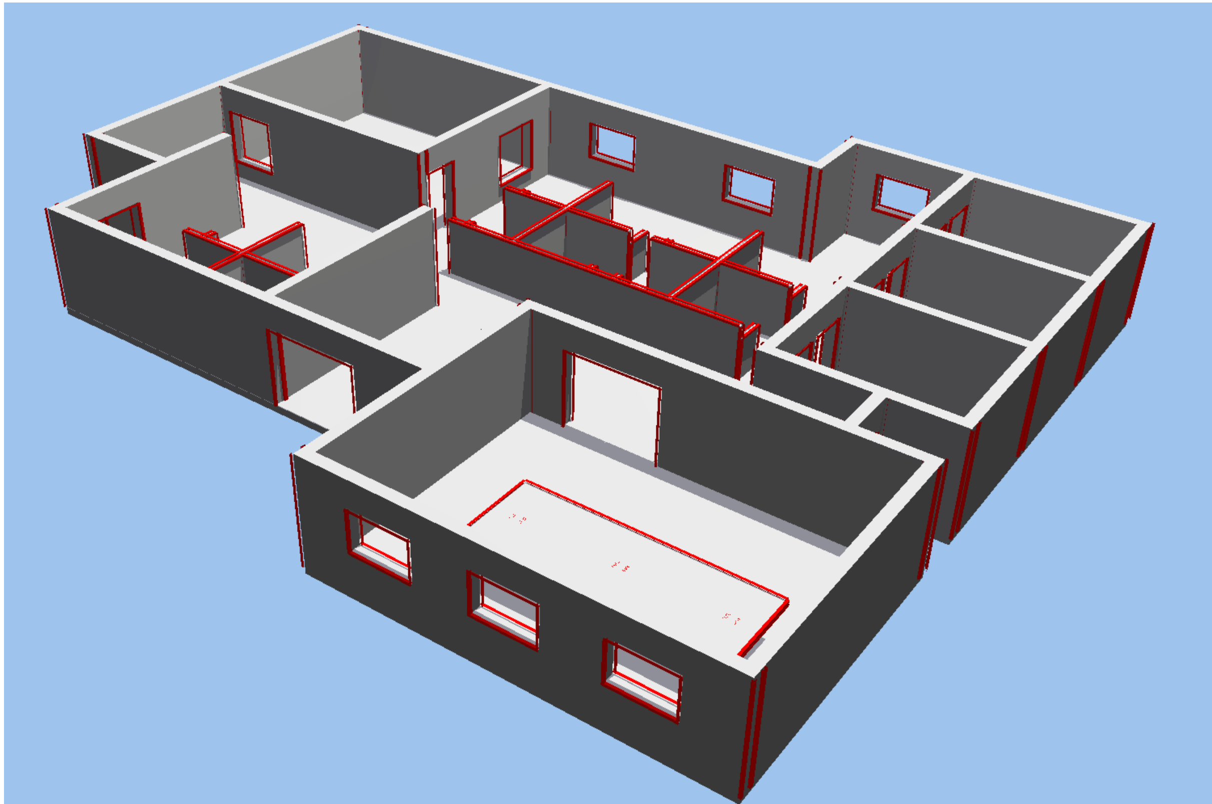


Figure 9: Edges after removal of two levels of top and bottom edges

3.2. Edge subdivision

This step is required before running the next step which actually performs edge reduction. Often, two edges overlap only partly due to their different lengths or a displacement (see Figure 7 for the first case). This function, `SubdivideEdges()`, subdivides edges to segments that completely coincide. Figure 10 shows how partly-overlapping edges must be subdivided before the next edge-reduction step. (Both edges are drawn with a small horizontal displacement for the sake of their visibility, but generally they are overlapping).

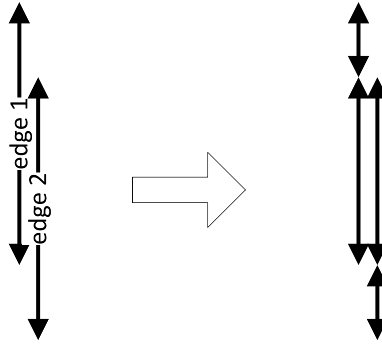


Figure 10: An example of subdivision of partly overlapping edges
(coinciding, but shown with horizontal displacement for visibility)

This subdivision is made pair-wise with one edge being the dividing edge and the other the divided edge. The computational complexity is $O(n^2)$.

A small mutual displacement of edges is actually allowed and supported – this is necessary to accommodate computational errors due to the limited computational precision (of the *double floating point* type), as well as minor scene design inaccuracies. Actually, any displacement much smaller than the radio signal wavelength can be considered unimportant and can be ignored. Similar tolerances are also used in other edge-reduction functions described in this document. A set of constants in the program specifies tolerances for various functions. These tolerances should actually be related to the radio signal wavelength, but in any case they should be larger than the computational errors caused by the limited computational precision.

This procedure changes the edges (subdivides some edges to shorter segments) but this is not visible in the visualization, which remains the same as in Figure 9.

3.3. Edge reduction

This procedure removes multiple coinciding edges, assuming that they belong to walls aligned to each other and there is actually no diffraction edge at that location, as shown by an example in Figure 11.

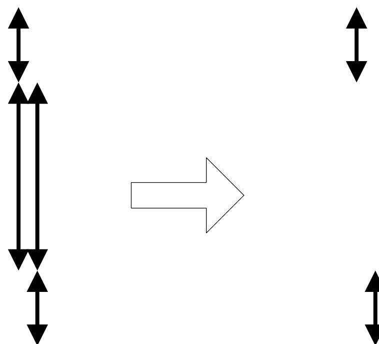


Figure 11: An example of edge reduction of partly overlapping edges
(coinciding, but shown with horizontal displacement for visibility)

Three versions of this functions have been developed, `ReduceEdges1()`, `ReduceEdges2()` and `ReduceEdges3()`, with different level of complexity:

Version 1 can remove pairs of coinciding edges.

Version 2 can remove multiple coinciding edges (pairs, triplets, etc.).

Version 3 can remove multiple coinciding edges but does this conditionally, taking into account the angles of the corresponding planes (cuboids faces) forming these edges. This is illustrated in the following three figures, which show ground plans of two vertical walls, with their vertical edges shown as red dots. Two walls (shown in Figure 12) are normally aligned perpendicularly as shown in Figure 13. The edge-only edge-reduction algorithm (all three versions) would correctly remove edges 2 and 3 (as already mentioned, it is not able to remove edge 4).

However, the scene design editor allows positioning a wall at a non-orthogonal angle and also overlapping with another wall (i.e. penetrating it). An example is shown in Figure 14. In this example, instead of removing both edges, version 3 correctly creates a single edge with its corresponding planes taken from both original edges. The implemented algorithm is able to correctly combine not only two but multiple walls placed obliquely to each other (with coinciding edges).

Edge reduction is performed pair-wise, the algorithm's computational complexity is $O(n^2)$.

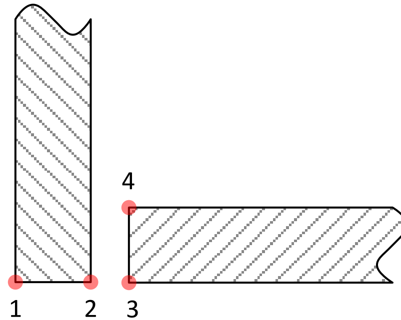


Figure 12: Two walls – ground plan

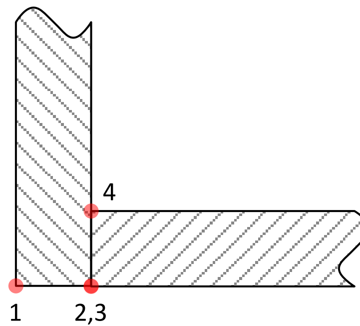


Figure 13: Two walls, normally perpendicularly aligned

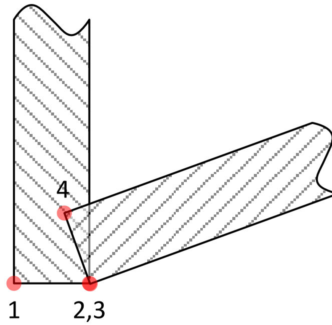


Figure 14: Two walls, aligned obliquely (overlapping)

Since our test scene consists of perpendicularly placed walls only (and without overlapping), all three versions of the edge reduction algorithm give the same results. Figure 15 shows edges remaining after running the following functions:

```
RemoveBottomEdges();
RemoveBottomEdges();
RemoveTopEdges();
RemoveTopEdges();
SubdivideEdges();
ReduceEdges3();
```

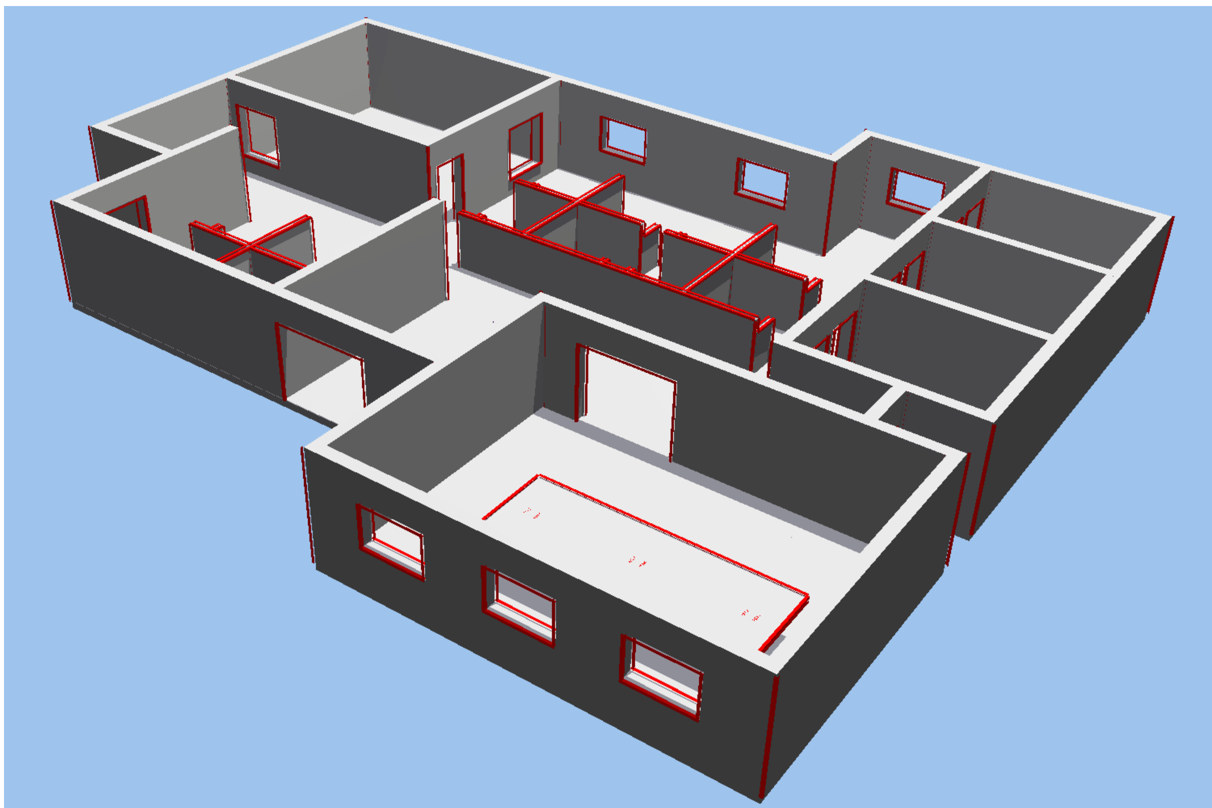


Figure 15: Edge reduction with bottom & top preprocessing

The removal of top and bottom edges could be performed after reduction (as postprocessing instead of preprocessing) with the same end result, but edge subdivision and reduction would then take longer due to a larger number of edges processed by these two steps (see Table 1).

Figure 16 shows the remaining edges in case that bottom and top edges are not removed, i.e. only the following two functions are executed:

```
SubdivideEdges();
ReduceEdges3();
```

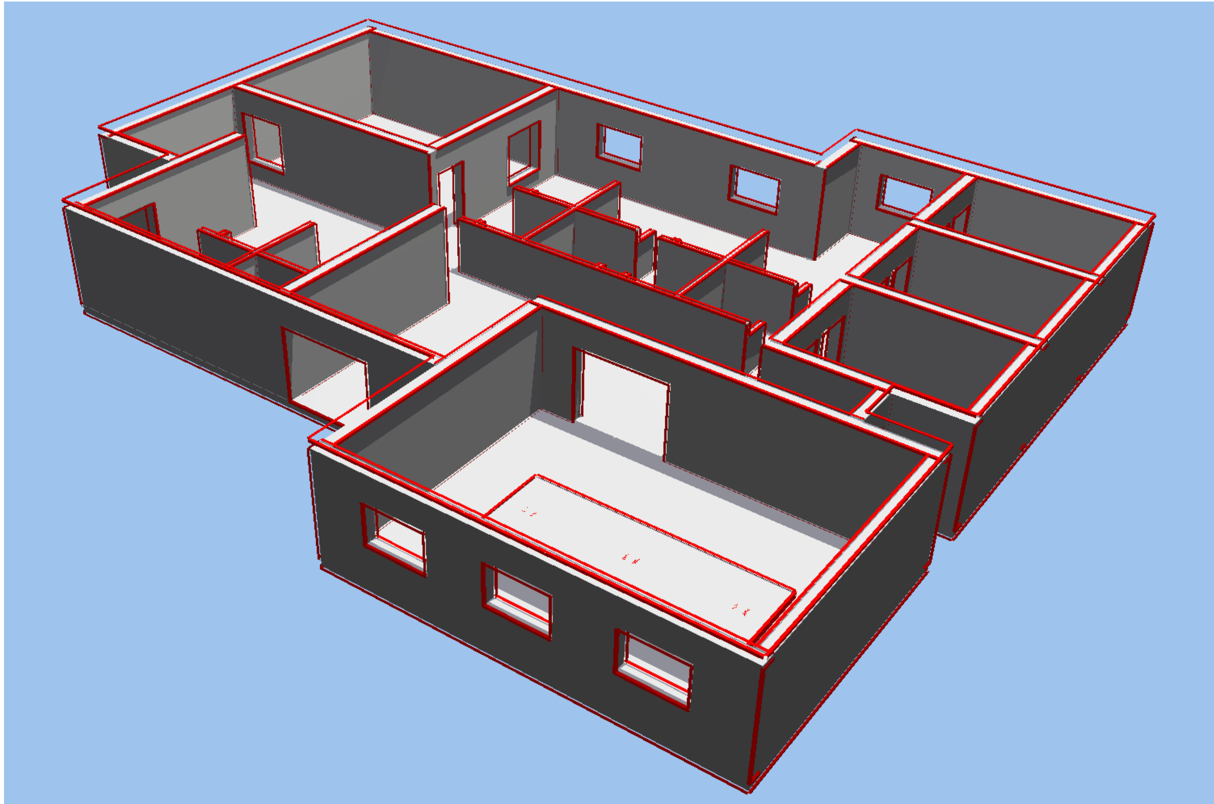


Figure 16: Edge reduction without bottom & top preprocessing

3.4. Removing short edges

A side effect of the described algorithms is that some short edges remain that should actually have been removed. One reason for this is that four short edges of each cuboid (those representing wall thickness) are not included as possible diffraction edges in the design. This results in unpaired subedges of coinciding edges not being removed. These subedges are of the size of wall thickness. Another reason is limited numerical precision, which can generate some very short edges (of the size of numerical errors).

In this step, edges shorter than a predefined length (a constant in the program) are removed. This function is simple and fast, with computation complexity $O(n)$. Figure 17 shows remaining edges after the execution of the following sequence of functions:

```
RemoveBottomEdges();
RemoveBottomEdges();
RemoveTopEdges();
RemoveTopEdges();
SubdivideEdges();
ReduceEdges3();
RemoveShortEdges();
```

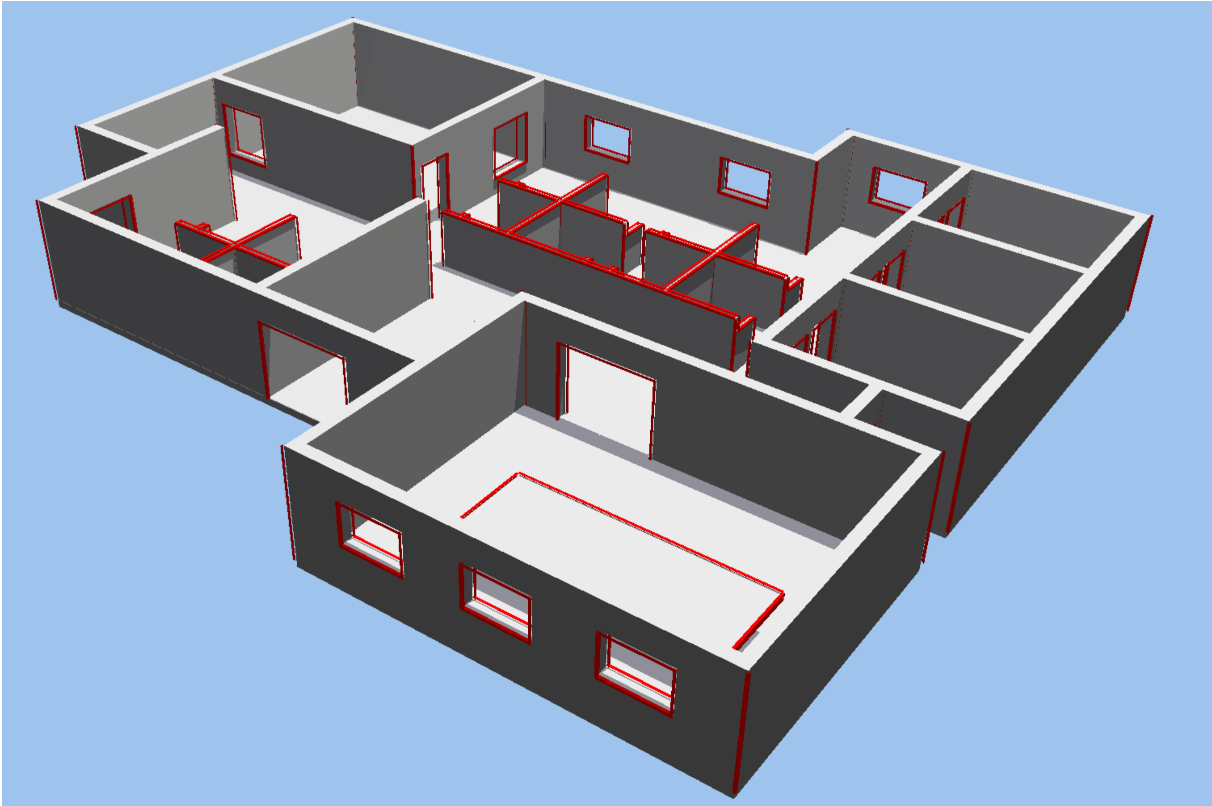


Figure 17: Final result, with bottom & top preprocessing

In the above case, only six short edges on the bottom table surface have been removed. If the bottom and top edges removal were not performed, a much larger number of short edges would remain, mostly at the second top and bottom levels, and would be removed in this step. Figure 18 shows the results for this case, with only the following sequence of functions executed:

```
SubdivideEdges();
ReduceEdges3();
RemoveShortEdges();
```

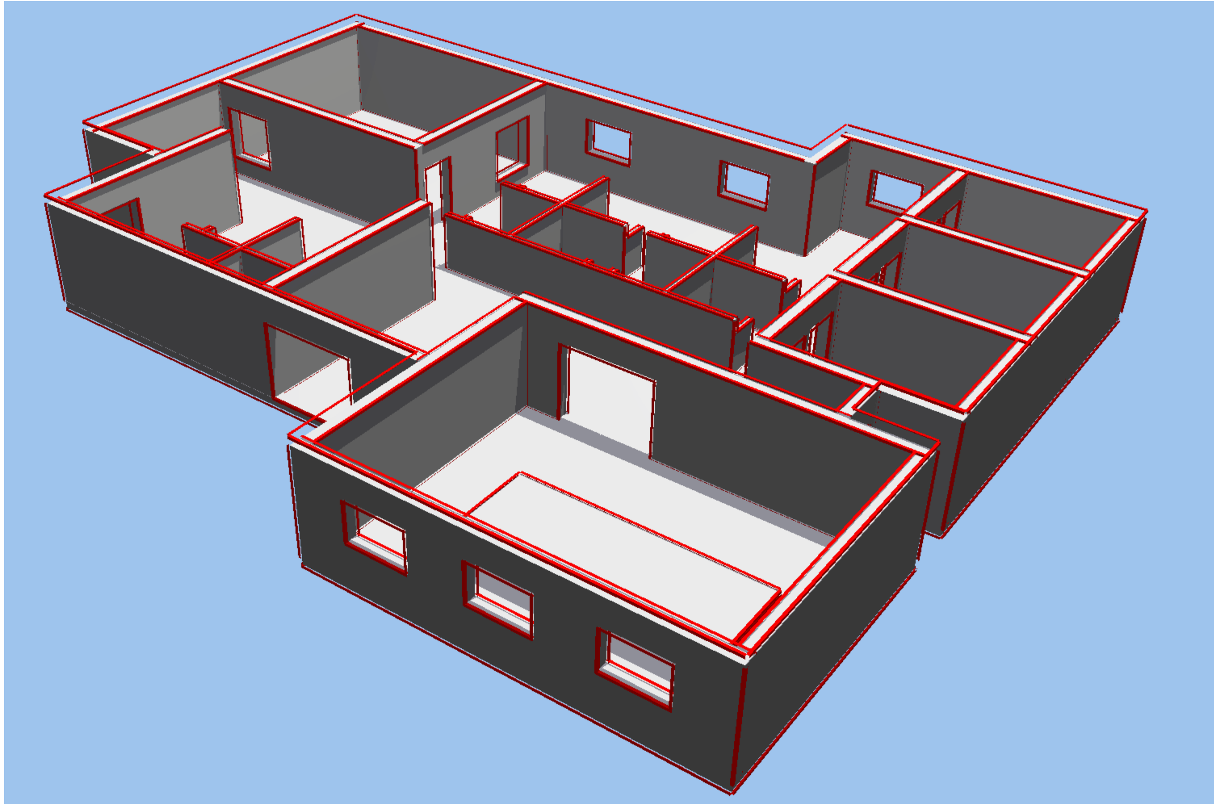


Figure 18: Final result, without bottom & top preprocessing

3.5. Cleaning up

After removing edges in previous steps, some points and side vectors become redundant (unused). `CleanupEdges()` removes theses redundant data before the new reduced edge list file is written. Edges themselves remain unchanged. This function is fast with computation complexity of $O(n)$.

3.6. Implementation and processing times

The complete processing consists of execution of the following chain of C functions:

```
RemoveBottomEdges();
RemoveBottomEdges();
RemoveTopEdges();
RemoveTopEdges();
SubdivideEdges();
// ReduceEdges1();
// ReduceEdges2();
ReduceEdges3();
RemoveShortEdges();
CleanupEdges();
```

The edge-reduction algorithm does not use GPU but runs entirely on CPU as a single threaded process. Its source code (C++) can be compiled either as a 'debug' or a 'release' version, which vastly influences the execution times. Table 1 shows execution times of each previously described function for the 'debug' version with time resolution of 10 ms. Besides the execution times, the numbers of processed (input) edges are also given in the table.

`CleanupEdges()` does not change the number of edges, so its input number is also the final (output) number of edges after the edge reduction process has completed.

Table 1: Execution times of edge-only edge reduction, ‘debug’ version

	w/o RemoveBottom/TopEdges (execution time / input edges)	with RemoveBottom/TopEdges (execution time / input edges)
2x RemoveBottomEdges()	–	2x 0.00s .. 0.01s / 612, 598
2x RemoveTopEdges()	–	2x 0.00s .. 0.01s / 474, 460
SubdivideEdges()	2.19s / 612	0.89s / 398
ReduceEdges1()	0.82s / 675	0.31s / 410
ReduceEdges2()	0.93s / 675	0.34s / 410
ReduceEdges3()	1.11s / 675	0.41s / 410
RemoveShortEdges()	0.00s (<0.01s) / 543	0.00s (<0.01s) / 346
CleanupEdges()	0.02s / 490	0.01s / 340

The complete processing time is dominated by edge subdivision and reduction, with the resulting complexity of $O(n^2)$.

Table 2 shows execution times for the ‘release’ version. The execution times are much shorter (about 50×) and cannot be precisely measured due to the limited time resolution.

Table 2: Execution times of edge-only edge reduction, ‘release’ version

	w/o RemoveBottom/TopEdges (execution time / input edges)	with RemoveBottom/TopEdges (execution time / input edges)
2x RemoveBottomEdges()	–	2x 0.00s (<0.01s) (612, 598)
2x RemoveTopEdges()	–	2x 0.00s (<0.01s) (474, 460)
SubdivideEdges()	0.04s (612)	0.02s (398)
ReduceEdges1()	0.02s (675)	0.01s (410)
ReduceEdges2()	0.02s (675)	0.01s (410)
ReduceEdges3()	0.02s (675)	0.01s (410)
RemoveShortEdges()	0.00s (<0.01s) (543)	0.00s (<0.01s) (346)
CleanupEdges()	0.00s (490)	0.00s (<0.01s) (340)

4. Edge–wall approach to edge reduction

The edge–wall approach to the diffraction edge reduction described in this chapter was first planned as a second step of the reduction process (following the edge-only approach from the previous chapter). However, the edge–wall approach offers a complete solution making the edge-only approach unnecessary. Although one might expect the edge–wall approach to be more complex, its actual implementation is rather simple and its execution comparably fast and efficient. It was implemented as a C++ class for inclusion into the scene editing application. The complete source code includes a new internal optional MATLAB-based visualization and an optional demo (test) main program.

4.1. ReduceEdges class – usage and data structures

The ReduceEdges constructor, `ReduceEdges::ReduceEdges( Scene *scene)`, does it all: it transforms the scene description data from the original data format to the ReduceEdges format and performs edge reduction. The final result, i.e. the reduced set of diffraction edges, can be found in its data structure described below (each edge is stored with the wall to which it belongs).

All the walls are stored in a C++ vector (array) of walls.

```
std::vector<re_Wall> re_walls;
```

Each wall is a structure containing two C++ vectors (arrays): one for (rectangular) cuboids and one for (diffraction) edges. A simple wall without holes has exactly one cuboid and eight edges (the four short edges representing wall thickness are not included). Each wall's hole is represented by an additional cuboid. In other words: the first cuboid in the C++ vector of cuboids represents a wall, while any additional cuboids represent holes in this wall.

```
struct re_Wall
{
  std::vector<re_Cuboid> cuboids;
  std::vector<re_Edge> edges;
};
```

Each cuboid is represented by an array of eight vertices, and array of six normalized (unit) vectors representing normals to its faces, directed outwards. This description is redundant but it is suitable for efficient further processing. Each coordinate is defined in 3D space with three double precision floating point values (x,y,z), using dvec3 type from GLM (OpenGL Mathematics) library.

```
struct re_Cuboid
{
  glm::dvec3 p[8];
  glm::dvec3 n[6];
};
```

Each edge is described by two points, each defined in 3D space with three double precision floating point values (x,y,z).

```
struct re_Edge
{
  glm::dvec3 p[2];
};
```

Visualization (MATLAB-based, see below) can be enabled at compile time. If enabled, MATLAB engine is started after edge reduction has been performed, and the scene with the resulting diffraction edges is drawn and displayed (see Figure 21) until the user presses the 'Return' key.

4.2. Visualization

The edge–wall reducing approach has its own internal optional visualization implemented as part of its ReduceEdges class. Visualization is performed by MATLAB (which must be installed on the computer) using its ability to execute its statements directly from C++. The implemented visualization was tested with MATLAB version R2016b. Due to available libraries, visualization can only be linked and used with the 'debug' version of the program and not with the 'release' version.

Visualization uses MATLAB's 'patches' for drawing 3D polygons. Here, they are used to draw rectangular cuboids' faces (in light blue color, see Figures 19 to 21). Unfortunately, there is no simple way to make openings (holes) to cuboids. Instead, openings are drawn as non-transparent cuboids in light gray color (see Figures below).

For correct rendering, holes' cuboids are made a little thicker than the wall (currently 0,1 mm on each side, fixed in the code), so that their two main faces are located outside the wall. This thickening only happens when visualization is enabled.

The following additional information is visualized (besides cuboid's faces):

- Diffraction edges in red color, with their endpoints marked with small red circles.
- Cuboids faces' normal vectors, normalized and scaled to the length of 0,5 m, positioned in the centers of faces and directed outwards. They are drawn as arrows in blue color.
- Walls cuboids' sequential numbers in black color, positioned in the center of cuboids.
- Wall holes' cuboids numbers in the form <wall_number>–<hole_local_number> in black color, positioned in the center of cuboids.
- Edges' numbers in the form <wall_number>–<edge_local_number> in red, drawn at the edges' midpoints.

Figure 19 shows non-transparent virtualization, which hides most of the important details. The ceiling could be removed to show the interior as in Figure 20, however openings (doors, windows) are non-transparent, and some important information would still be hidden (e.g. numbers located within the walls). (Figure 20 has actually been produced by completely omitting the ceiling also from the edge reduction process, which is why the horizontal edges of vertical walls at the celling level have not been removed.)

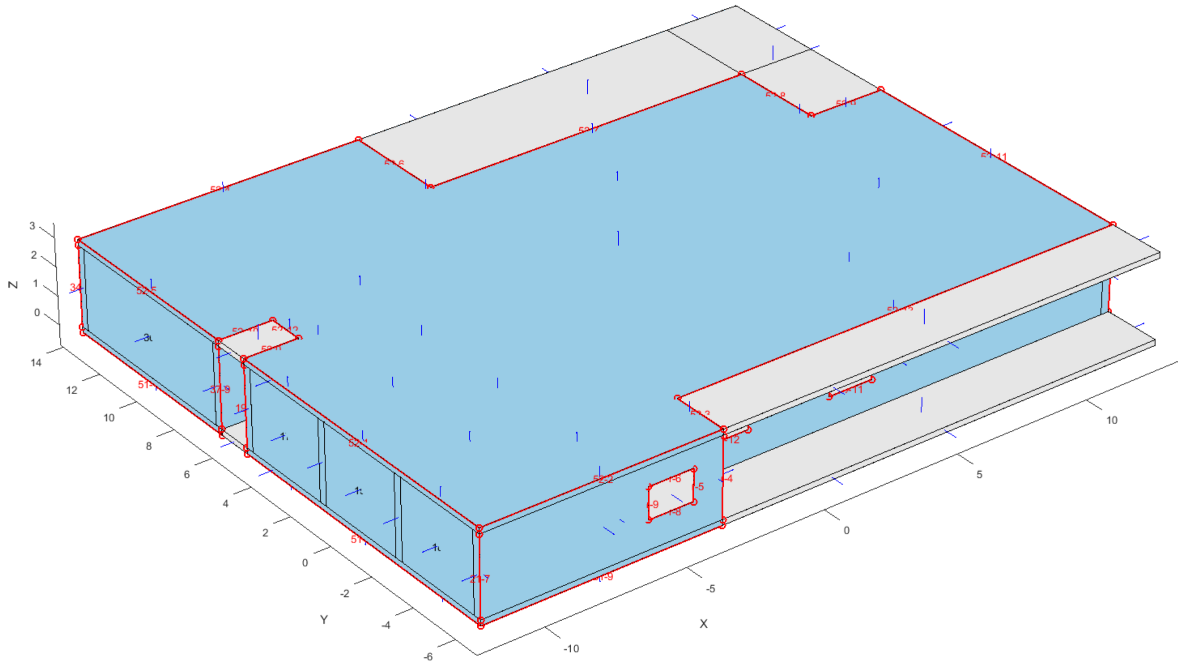


Figure 19: Non-transparent rendering

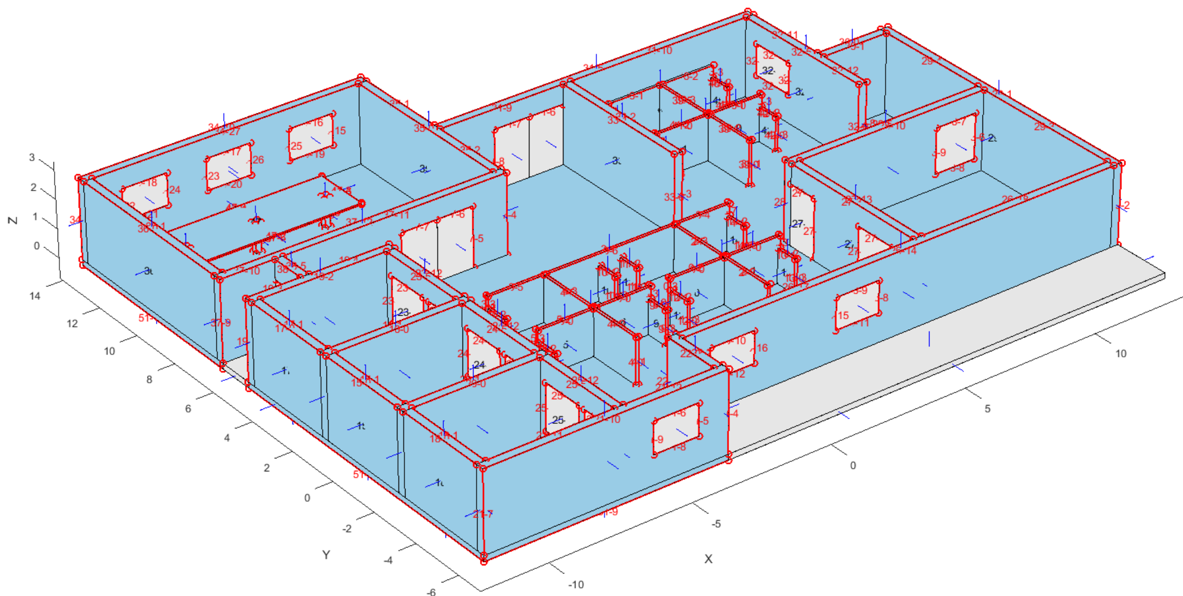


Figure 20: Non-transparent rendering with ceiling removed from processing

To avoid these visualization problems, drawings of patches can be performed partially transparently, as in Figure 21. A side effect is that MATLAB does not render wall openings consistently and with the correct color, but this is not really important for the intended test usage.

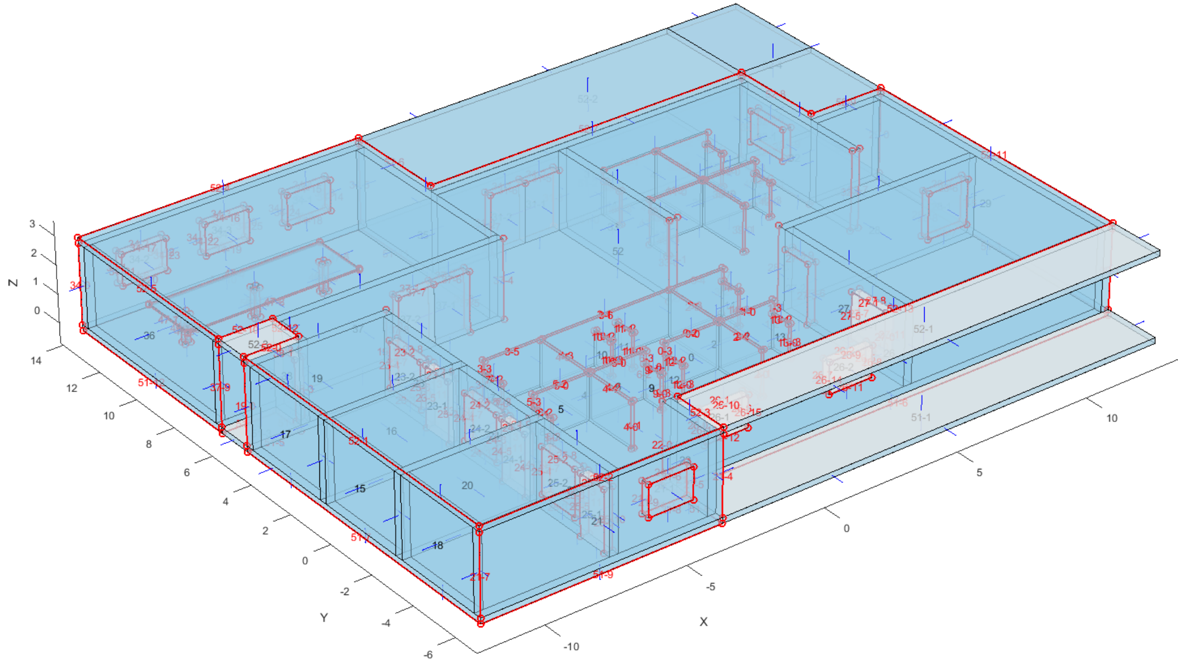


Figure 21: Partially transparent rendering

4.3. Edge reduction algorithm

The ReduceEdges constructor first loops through all the walls in the orginal data structure and transforms them to the ReduceEdges data structure (described above). This is a quick operation with $O(n)$ complexity (n being either the number of walls, cuboids or edges – these numbers can be assumed to be mutually proportional for a given scene design complexity).

Next, ReduceEdges subdivides all edges so that each edge segment lies entirely inside or outside of any cuboid (wall or hole). It does so by taking from each wall's edge list every edge (including those resulting from holes), finding its intersections with surfaces of any cuboid (walls and their holes), and splitting the edge accordingly. It then checks all newly generated edge segments and remove those lying inside a wall (the first wall's cuboid) and not inside any of the wall's holes (the remaining wall's cuboids).

The computation complexity of this operation is $O(n^2)$.

Figure 22 shows subdivision of an edge (belonging to an obliquely positioned overlapping wall not shown in the picture) to three segments as a result of intersections with another wall (the one displayed in the picture). The middle segment (located inside the displayed wall) is consequently removed.

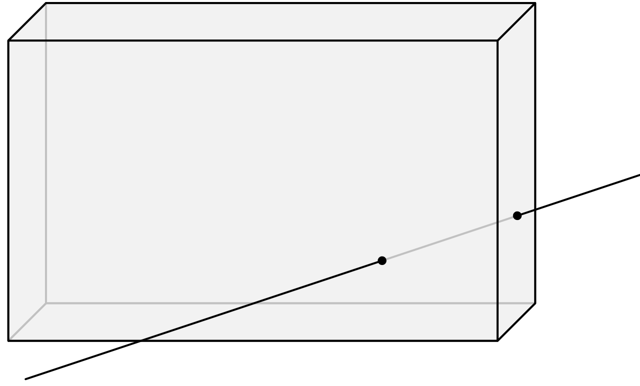


Figure 22: Edge subdivision

Some tolerances are necessary when testing if an edge lies (approximately) inside a wall or on its surface. This is currently accomplished by enlarging the wall's cuboids and reducing the size of its openings' cuboids for a small amount. The tolerances are necessary to accommodate not only the limited computational precision (double floating point) but also minor design inaccuracies. Any inaccuracies much smaller than the radio signal wavelength can be ignored. Allowed tolerances for various computations are defined as constants in the program and are currently set to 2 cm.

4.4. Execution time

The algorithm reduces the number of diffraction edges of the test scene (shown in previous Figures) from the original 616 to the final 304. (616 edges instead of 612 from Chapter 3 is due to additional four very short "phantom" edges generated by triangulation and included in the scene description XML file used in this chapter.)

The edge reducing algorithm executes on a single CPU core and does not use GPU processing. The following execution times are for the reduction algorithm only, excluding the optional visualization part (which can only be used with the 'debug' version and not with the 'release' version):

- 'release' version: 0.05 s
- 'debug' version: 2.54 s

The overall computational complexity of the algorithm (excluding visualization part) is $O(n^2)$.