The Transmission Line Matrix Method with Android Implementation

by

Mengfei Jiang
B.Sc., Nanjing University of Posts and Telecommunication, 2014

A Dissertation Submitted in Partial Fulfillment
of the Requirements for the Degree of

MASTER OF ENGINEERING

in the Department of Electrical and Computer Engineering

The Transmission Line Matrix Method with Android Implementation

by

Mengfei Jiang
B.Sc., Nanjing University of Posts and Telecommunication, 2014

Supervisory Committee

Dr. Poman P.M. So, (Department of Electrical and Computer Engineering)

Supervisor

Dr. Xiaodai Dong, (Department of Electrical and Computer Engineering)

Departmental Member

Supervisory Committee

Dr. Poman P.M. So, (Department of Electrical and Computer Engineering)

Supervisor

Dr. Xiaodai Dong, (Department of Electrical and Computer Engineering)

Departmental Member

Abstract

The Transmission Line Matrix (TLM) method is one of the time-domain numerical techniques to simulate electromagnetic field behaviour. We have implemented this method in a TLM App with field animation and Discrete Fourier Transform (DFT) capabilities. This report describes the theory of the two-dimensional shunt-connected TLM method and the software architecture of the implemented TLM App. The App can run in Android OS with API 23 or higher.

Table of Contents

List of Acronyms

Apps	applications.
CAD	Computer Aided Design.
CPUs	Central Processing Units.
GPUs	Graphic Processing Units.
GUI	Graphical User Interface.
Java VM	Java Virtual Machine.
OpenGL	Open Graphic Library.
OS	Operating System.
SDK	Software Development Kit.
TLM	Transmission Line Matrix.
UI	User Interface.

Android and OpenGL Terms

Fragment — A fragment behaves like a nested activity that can define its own layout for different screen size and manage its own lifecycle. When a fragment specifies its own layout, it can be configured in different combinations with other fragments inside an activity to create a dynamic and multi-pane user interface on Android.

ViewPager & FragmentPagerAdapter — FragmentPagerAdapter is to generate and manage fragment instances. The view for the current fragment is displayed in ViewPager. When users swipe left and right through fragments, ViewPager also provides a convenient way to recycle or destroy the current fragment.

Button — A button represents a push-button widget. Push-buttons can be pressed, or clicked, by the user to perform an action.

ListView — A view that shows items in a vertically scrolling list.

TextView — A view that displays text to the user and optionally allows them to edit it. A TextView is a complete text editor, however the basic class is configured to not allow editing.

EditText — EditText is a thin veneer over TextView that configures itself to be editable.

Vertex Shader — Control how to draw a vertex on a screen in OpenGL.

Fragment Shader — Control how to draw a primitive on a screen in OpenGL.

VBO — A buffer object to store vertex data in OpenGL.

IBO — A buffer object to store the indices of vertex data in the corresponding VBO.

List of Tables

List of Codes

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

My parents, for supporting me in the low moments.

Professor Poman So, for mentoring, support, encouragement, and patience.

I believe I know the only cure,
which is to make one's center of life inside of one's self,
not selfishly or excludingly,
but with a kind of unassailable serenity-to decorate one's inner house
so richly that one is content there,
glad to welcome any one who wants to come and stay,
but happy all the same in the hours when one is inevitably alone.

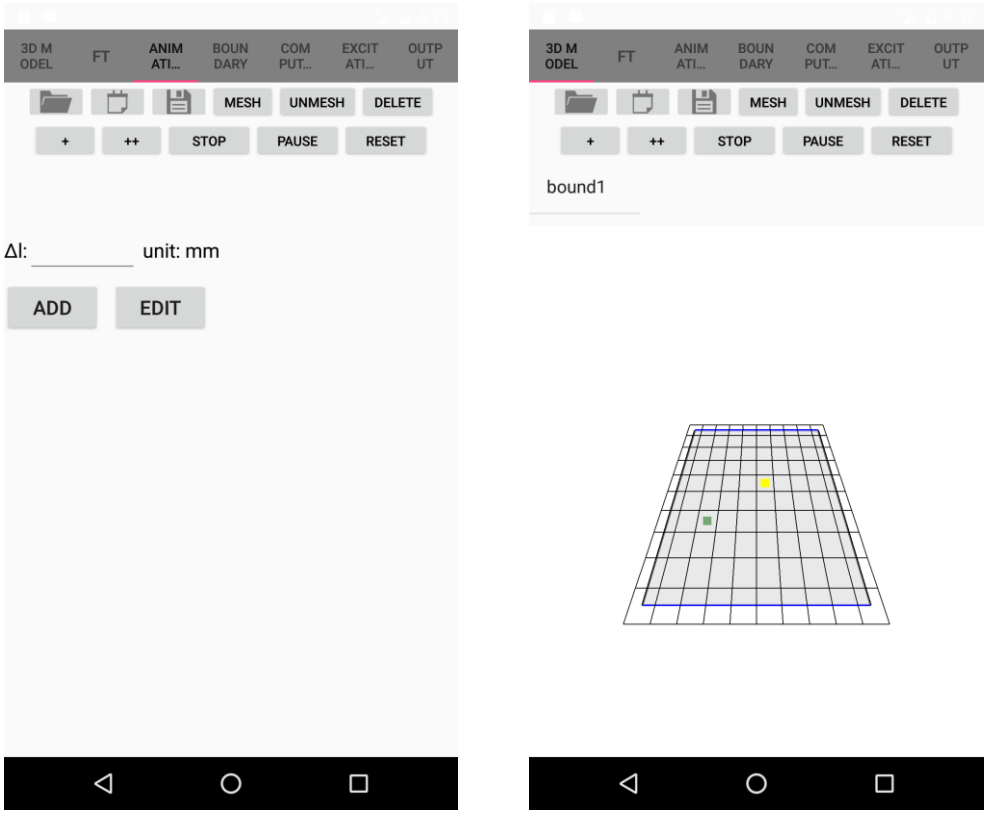
Mengfei Jiang

Chapter 1. Introduction

The Transmission Line Matrix (TLM) method is one of the time-domain numerical techniques used by the high-frequency engineering industry to simulate electromagnetic field behaviour. The method has three core computing procedures namely scattering, transfer, and reflection of voltage impulses. The underlying numerical operations can be executed sequentially in traditional Central Processing Units (CPUs) or in parallel in modern Graphic Processing Units (GPUs). Although most of today's mobile devices have sufficient processing power to execute a TLM program, due to software incompatibility between desktop and mobile operating systems, existing TLM programs for desktop computers cannot run on mobile devices.

From the hardware point of view, a critical difference between desktop computers and mobile devices is in their power sources. The former class of devices are connected to continuous power supplies and the latter group are powered by rechargeable batteries. As a result, computer programs for desktop computers are designed to maximize processing performance at the expense of power consumption whereas applications (apps) for mobile devices are designed to minimize power consumption at the expense of processing performance. Another characteristic difference between these two classes of hardware is their screen sizes. Screens for mobile devices are relatively small when compare to typical computer displays. Hence, traditional Graphical User Interface (GUI) designed for desktop computers are not suitable for use in mobile devices.

Android's Activity Templates provide many useful features for implementing apps for the mobile environment. We have implemented a TLM app using Android's Tabbed Activity template; each tab in the TLM app provides a view similar to a window in a traditional desktop program. The TLM app’s simulation and visualization modules are realized using Java and Open Graphic Library (OpenGL). Figure 1 depicts a few screen shots of the TLM app running under an Android phone emulator. As can be seen in the figure that the TLM app has a row of buttons at the top of the app. These buttons are for selecting one of the available tabs in the app. The features in each of the tabs in the TLM app are described in the ensuing chapters.



(a) (b)

Figure 1: Two screen shots of the developed Android TLM app.
(a) The structure editing tab for entering Δl.
(b) The structure input tab for viewing the created structure.

1.1 Functions of the Application

The developed Android app has the following functions:

i. Read a TLM network file from external storage;
ii. Save the TLM structure to external storage;
iii. Reset the app by clearing all data or read in a new TLM structure;
iv. Single click on a boundary to select it.
v. Double click on a selected boundary to display an edit menu.
vi. Drag a selected boundary horizontally or vertically to move it.
vii. Use the Start, Stop, Pause, Reset, + and ++ buttons to control TLM simulation.

1.2 Overview

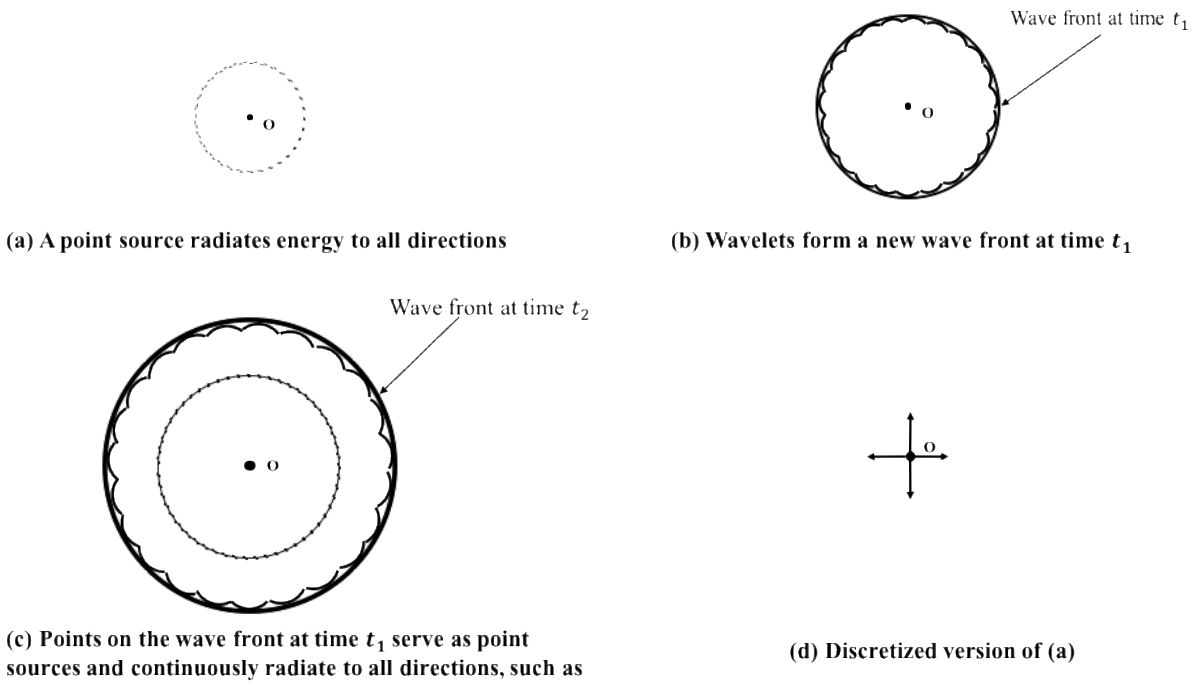
This report has seven chapters and they focus on:

- **Chapter 1** — introduction to the project.
- **Chapter 2** — theory and algorithms of the TLM method.
- **Chapter 3** — graphical interface design and software architecture. Android SDK objects and concepts such as fragment, listener, button, gesture, and object inheritance are discussed.
- **Chapter 4** — OpenGL rendering mechanism. The concepts of graphic pipeline, vertex buffer, index buffer, and coordinate mapping are covered.
- **Chapter 5** — cooperation between the field simulation and DFT calculation threads. The relationship between data generation and signal processing is explained.
- **Chapter 6** — validation of modeling results. The cut-off frequencies of a few rectangular waveguide computed using the TLM app and MEFiSTo are compared with the theoretical values.
- **Chapter 7** — conclusion and discussion. Suggestions on possible improvements are given before the conclusion.

Chapter 2. Two-dimensional TLM Method

Christen Huygens stated in his 1690 paper that wave propagation could be considered as "*all points on a wave front serve as point sources of spherical secondary wavelets. After a time T, the new position of the wave front will be the surface of tangency to these secondary wavelets*" [3]; this is illustrated in Figure 2(a)-(c). In Figure 2(a), a point source radiates energy into all directions. After time t_1, wavelets in all directions form a wave front. All points on this wave front can act as point sources and continuously produce new wave fronts. Figure 2(d) represents a discretized version of Figure 2(a) that restricts to four propagation directions only. This discretized version is the basis of the TLM method developed by P.B. Johns and R.L. Beurle[5].

Huygens' wave model can be discretized into finite space points. Figures 3(a) and (b) show an example for the two-dimensional situation. This discretized wave model can in turn be modeled using a mesh of transmission lines as illustrated in figures 3(c) and (d).



(a) A point source radiates energy to all directions

(b) Wavelets form a new wave front at time t_1

(c) Points on the wave front at time t_1 serve as point sources and continuously radiate to all directions, such as another new wave front at time t_2.

(d) Discretized version of (a)

Figure 2: Huygens' model of wave propagation.

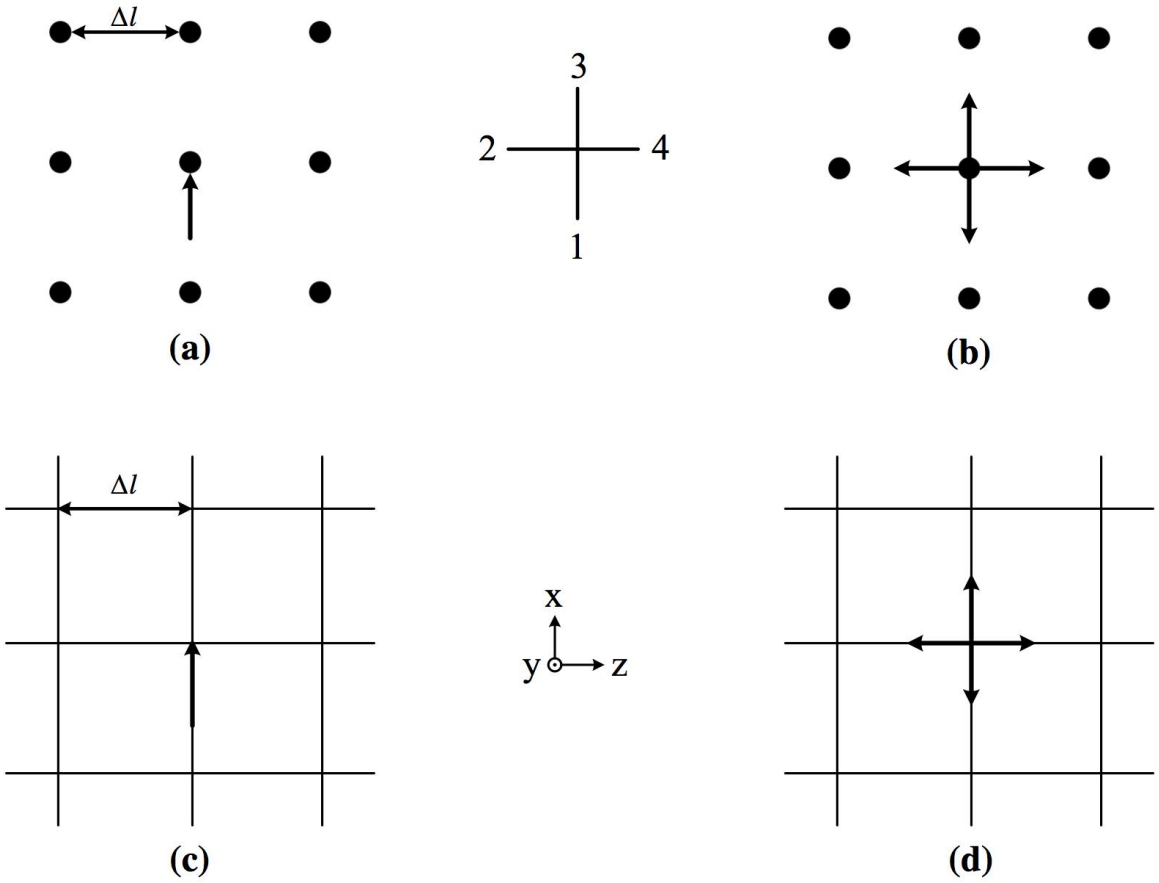


Figure 3: Discretized Huygens' wave model (a and b) and the equivalent TLM mesh (c and d).
(a) Incidence of a Dirac impulse is launched into the central space point.
(b) Scattered impulses at four directions.
(c) An incident voltage impulse is launched into the central shunt node.
(d) Scattered impulse at four link lines.

Figure 3(a) shows an impulse approaching the central space point in the discretized model from below (i.e. traveling along the positive x direction). Different from scattering isotropically into all directions in the continuous space, this impulse gets scattered into impulses into four directions only. The scattered impulses will propagate to the neighbouring space points, figure 3(b), and then act as new incident impulses to trigger a new set of scattering processes. In the discretized wave model, the time for an impulse to propagate from one space point to another is $\Delta t = \Delta l/c$, where Δl and c are the distance between to discrete space points and the velocity of light in free space, respectively.

Figures 3(c) and (d) illustrate a similar concept using a transmission line matrix (TLM). The transmission line junctions are called nodes which have scattering properties similar to the discrete space model. When an incident voltage impulse reaches a node, it will get scattered into all four transmission lines and propagate to the neighbouring nodes. When the scattered impulses arrive at the adjacent nodes, they will trigger a new set of scattering processes.

The above is a brief description of the TLM method. The remaining sections in this chapter will explain the details of voltage scattering, transfer and reflection algorithms for the TLM method. Implementation details for these algorithms is also discussed.

2.1 Scattering Process in a Shunt-connected TLM Network

The discussion on TLM so far does not target any specific transmission line connections. In general, a transmission line node can be created by either a shunt or a series connected transmission junction as shown in Figure 4. A shunt-connected TLM network contains only shunt nodes. The TLM theory and algorithms to be presented in this report are for the shunt-connected network made of air-filled two-wire transmission lines.

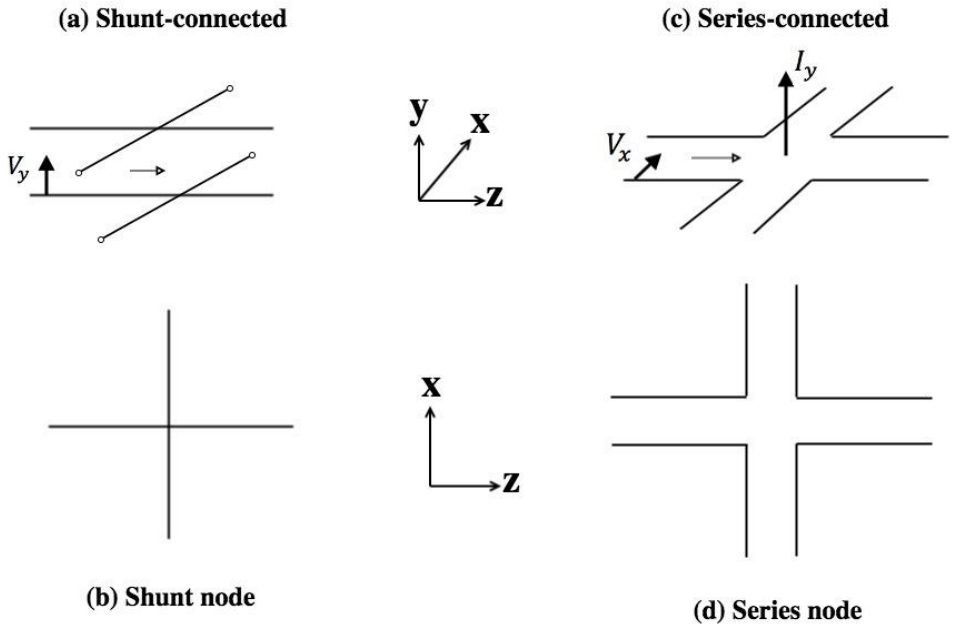


Figure 4: Two kinds of two-dimensional TLM node[4].

To facilitate our discussion and software implementation, node positions in the TLM mesh are numbered using a familiar *i-j* matrix index notation. Furthermore, transmission line segments connected to a node are called link-lines and numbered from 1 to 4 in the clockwise direction as shown in Figure 4. Hence link-line 1 of $node_{i,j}$ is connected to link-line 3 of $node_{i,j-1}$. Since the distance between two adjacent nodes is Δl, the time for a voltage pulse to travel through that distance is equal to $\Delta t = \Delta l/c$ where $c = 3 \times 10^8 \mathrm{m/s}$ is the speed of light in the air-filled transmission line.

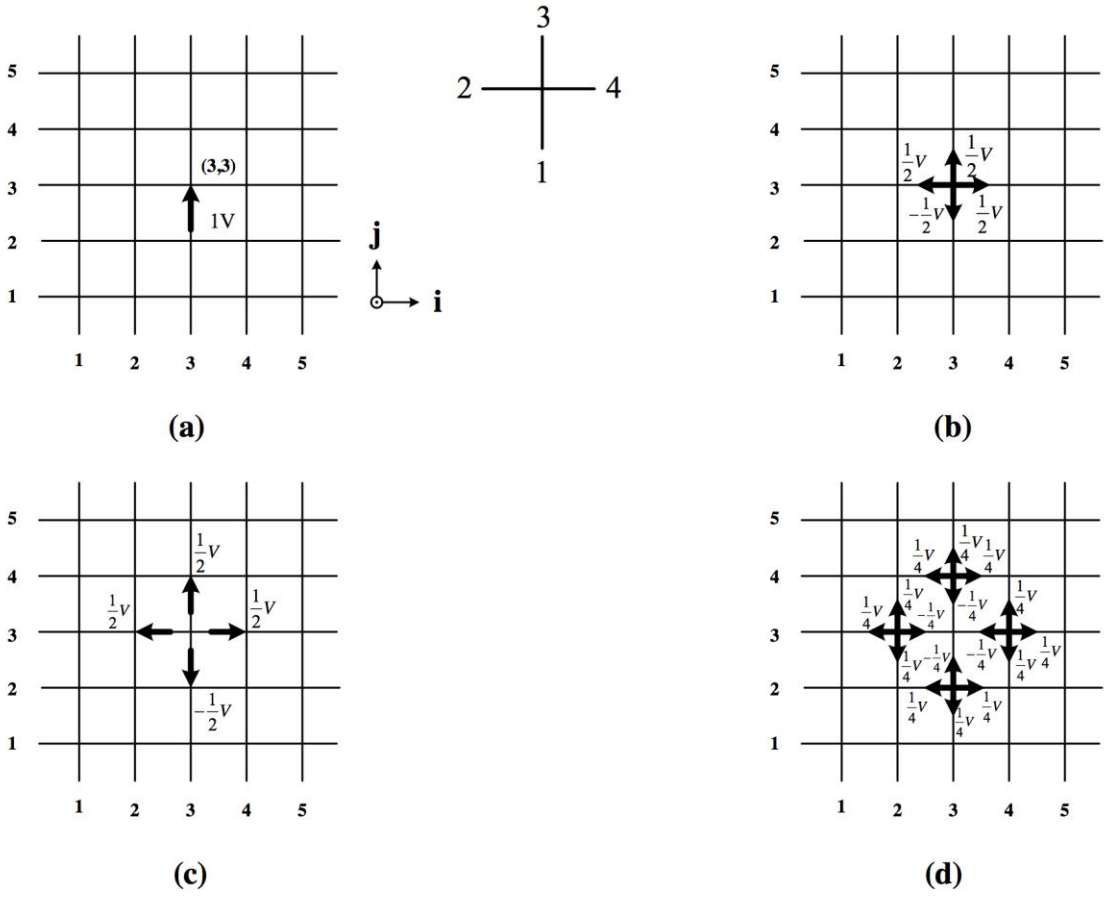


Figure 5: The scattering-transfer process in two time steps.
(a) An incident $1V$ impulse is launched into $node_{3,3}$ in the first time step.
(b) The incident impulse gets scattered into link lines 1-4 in the first time step.
(c) Scattered impulses reach the neighbouring nodes and trigger a set of new scattering process in the second time step.
(d) Impulses are scattered into four link lines in the second time step.

When an impulse incident on a shunt node from link line 1, the impedance, Z_L, seen by the impulse is the parallel combination of the impedances of the other three link lines, i.e. link lines 2 to 4. The reflection and transmission coefficients seen by the incident impulse are therefore equal to [4]:

$$\Gamma = \frac{Z_L - Z_0}{Z_L + Z_0} = \frac{\frac{Z_0}{3} - Z_0}{\frac{Z_0}{3} + Z_0} = -\frac{1}{2}, \qquad \tau = 1 + \Gamma = \frac{1}{2}$$

Applying the above formulas to the $1V$ incident impulse in Figure 4(a), the reflected impulse at $node_{3,3}$ in link-line 1 is $-\frac{1}{2}V$ and the transmitted impulses in link lines 2 to 4 are $\frac{1}{2}V$. These scatter impulses will propagate away from $node_{3,3}$ into its neighbouring nodes in four directions. Once the impulses reach the neighbouring nodes they will trigger a new set of scattering process. The elapsed time between two scattering-transfer operations is equal to Δt as this is the transit time for the impulses to move from nodes to nodes. In TLM literature, time is measured as an integer increment of Δt called time step, i.e. $1\Delta t$, $2\Delta t$, …$k\Delta t$, and so on. With this idea of time step, algorithms in this chapter mark the incident voltage impulses of $node_{i,j}$ at link-line n in time step k^{th} as $\boldsymbol{V}_n^i[i][j]_k$ and the scattered impulses at $node_{i,j}$ in link-line n in time step k^{th} as $\boldsymbol{V}_n^r[i][j]_k$, where $\boldsymbol{V}_n[i][j]$ is an element in a two-dimensional array. When all elements in the array are updated according to the TLM procedure, one iteration is completed.

Applying this concept to the events in Figure 4, at time step 1 $\boldsymbol{V}_1^i[3][3]_1 = 1V$, $\boldsymbol{V}_1^r[3][3]_1 = -\frac{1}{2}V$ and $\boldsymbol{V}_{2-4}^r[3][3]_1 = \frac{1}{2}V$. To simplify the notation further, subscript k and indices $[i][j]$ may be omitted when including them in an expression is not necessary. For instance, the scattering operation applies to all TLM nodes at every time step can be written as $\boldsymbol{V}^r = \boldsymbol{S} \times \boldsymbol{V}^i$; or in the expanded form:

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix}^r = \frac{1}{2} \cdot \begin{bmatrix} -1 & 1 & 1 & 1 \\ 1 & -1 & 1 & 1 \\ 1 & 1 & -1 & 1 \\ 1 & 1 & 1 & -1 \end{bmatrix} \times \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix}^i \qquad (2.1)$$

where $\left([v_1 \quad v_2 \quad v_3 \quad v_4]^i\right)^T$ and $([v_1 \quad v_2 \quad v_3 \quad v_4]^r)^T$ are the vectors that contain all incident and scattered voltages at the same node in the same time step.

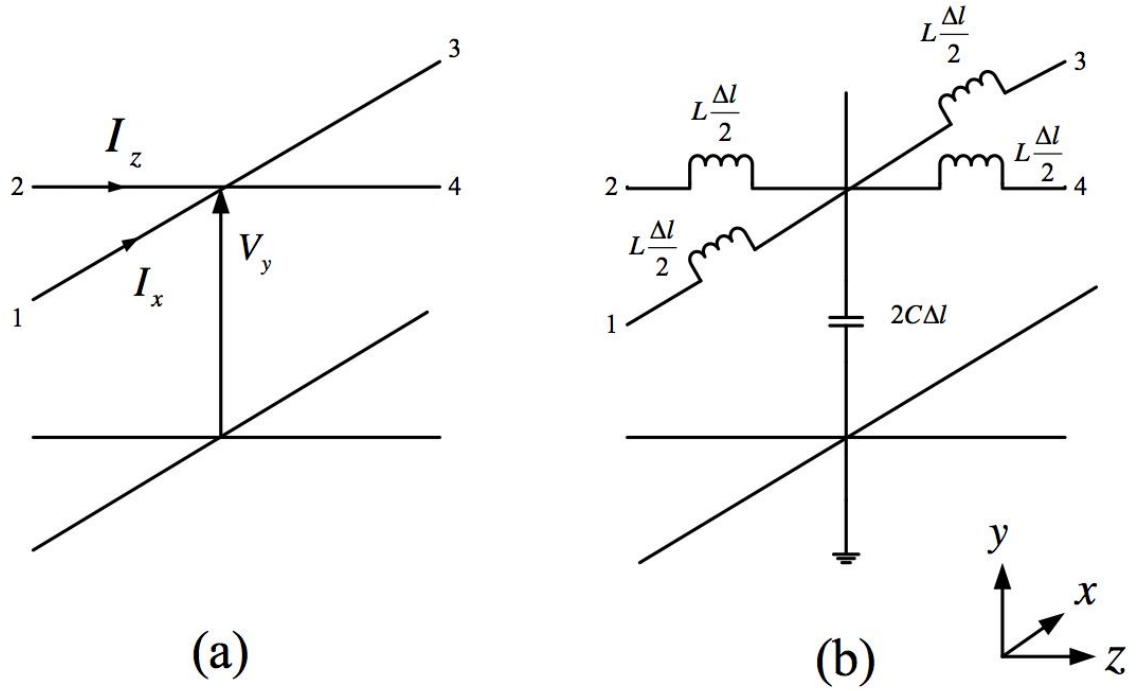


Figure 6: The unit cell of the two-dimensional shunt node TLM network.
C and L are the per-unit-length capacitance and inductance, respectively.
(a) A transmission line model.
(b) An equivalent lumped circuit element model.

Akhtarzad and John used the lumped element model shown in Figure 5 to demonstrate the equivalence between the circuit variables for TLM and the field variables for free space[4]. Transmission line theory gives the following partial differential equations

$$\frac{\partial V_y}{\partial x} = -L\frac{\partial I_x}{\partial t} \qquad \frac{\partial V_y}{\partial z} = -L\frac{\partial I_z}{\partial t} \qquad \frac{\partial I_z}{\partial z} + \frac{\partial I_x}{\partial x} = -2C\frac{\partial V_y}{\partial t} \tag{2.2}$$

In the lumped element model, L and C are the per-unit-length inductance and capacitance. Since the distance between two nodes is Δl, the inductance of each lumped inductor is $\frac{L}{2} \times \Delta l$ and total capacitance in a node is the parallel connection of two capacitances which is $2C \times \Delta l$. In free space, *y*-polarized TE_{10} waves moving in the *z*-direction must satisfy the following partial differential equations:

$$\frac{\partial E_y}{\partial x} = -\mu \frac{\partial H_z}{\partial t} \qquad \frac{\partial E_y}{\partial z} = +\mu \frac{\partial H_x}{\partial t} \qquad \frac{\partial H_x}{\partial z} - \frac{\partial I_x}{\partial x} = +\epsilon \frac{\partial E_y}{\partial t} \qquad (2.3)$$

The equivalence between circuit variables and field variables can now be seen by comparing equations (2.2) to (2.3)

$$E_y = V_y, \quad H_z = I_x, \quad H_x = -I_z \qquad (2.4)$$
$$\mu = L, \quad \text{and} \quad \epsilon = 2C$$

Hence the voltage behaviour in a shunt-connected transmission line network is indeed a proper analogy of the electric field behaviour of a TE wave free space or equivalent guided structures.

Equations 2.1 to 2.4 are valid for lossless media. A more general shunt-connected TLM mesh that can model electromagnetic wave propagation in lossy inhomogeneous media has a permittivity stub and a loss stub connected to the node. The permittivity stub has a normalized admittance y_0 and the loss stub has a normalized conductance g_0. Then a voltage impulse for a stub-loaded shunt node would be scattered into six link lines, as shown in Figure 6. However, only five impulses would be computed in the scattering process, since the loss stub would absorb the sixth impulses.

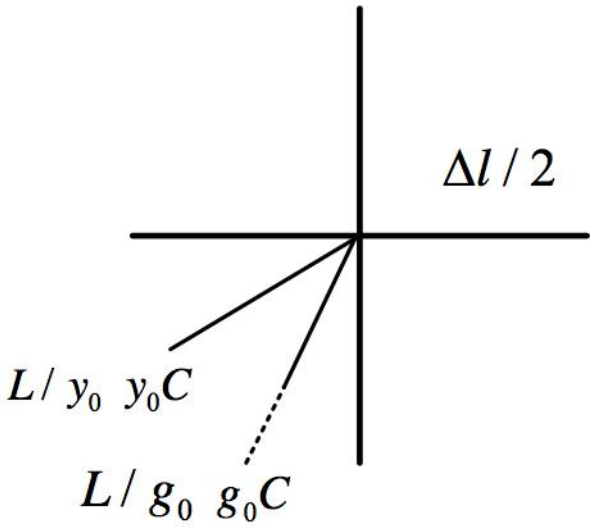


Figure 7: A shunt node with permittivity and loss stubs.
The length of the permittivity stub and the four link lines are the same, i.e. $0.5\Delta l$.
The length of the loss stub is infinite so that voltage pulse entered into that stub will leave the mesh for good.

Consequently, the revised scattering procedure for a stub-loaded shunt node is:

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix}^r = \frac{1}{y} \cdot \begin{bmatrix} -(y-2) & 2 & 2 & 2 & 2y_0 \\ 2 & -(y-2) & 2 & 2 & 2y_0 \\ 2 & 2 & -(y-2) & 2 & 2y_0 \\ 2 & 2 & 2 & -(y-2) & 2y_0 \\ 2 & 2 & 2 & 2 & 2y_0 - y \end{bmatrix} \times \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix}^i \quad (2.5)$$

where y and y_0 respectively are:

$$y = 4 + y_0 + g_0 = 4 + y_0, \quad y_0 = 4(\epsilon_r - 1), \quad \text{and} \quad g_0 = \sigma \Delta l \epsilon_0 / c \quad (2.6)$$

In Eq (2.6), ϵ_r is the relative permittivity, which is depends on the dielectric material, for vacuum, ϵ_r is 1.

Equation (2.5) can be rearranged to obtain a more compute efficient implementation as

$$v_a = \frac{2}{y}\left(v_1^i + v_2^i + v_3^i + v_4^i + y_0 v_5^i\right), \quad v_n^r = v_a - v_n^i \quad \text{for } n = 1 \ldots 5 \quad (2.7)$$

This is the scattering equation implemented in our TLM application. The program source code has five two-dimensional arrays to store the voltages in the link-lines and permittivity stubs.

Algorithm 1: Scattering Process of a stub-loaded shunt node

computationBox number = kc
for *nn* = 1 : *kc* **do**
  tmp = $hashmap_{computationBox}$.*get(nn)*
  for *j* = *tmp[3]*-1 : *tmp[4]*-1 **do**
    for *i* = *tmp[1]*-1 : *tmp[2]*-1 **do**
      $v_a = \frac{2}{y_0+4} \times \left(V_1^i[i][j] + V_2^i[i][j] + V_3^i[i][j] + V_4^i[i][j] + y_0 \times V_5^i[i][j]\right)$
      $V_1^r[i][j] = v_a - V_1^i[i][j]$
      $V_2^r[i][j] = v_a - V_2^i[i][j]$
      $V_3^r[i][j] = v_a - V_3^i[i][j]$
      $V_4^r[i][j] = v_a - V_4^i[i][j]$
      $V_5^r[i][j] = v_a - V_5^i[i][j]$
    end for
  end for
end for

Table 1: Algorithm of Scattering Process of a stub-loaded shunt node

2.2 Reflection Processes in a Shunt-Connected TLM Network

The algorithm given in equation (2.7) performs impulse scattering only. Scattered impulses emerging from all nodes must be transferred to their neighbouring nodes. In a TLM network that does not contain any boundary separating neighbouring nodes, transferring impulses to the neighbouring nodes is a straightforward data swapping. However realistic electromagnetic structures do have boundaries with arbitrary shapes, orientations, and locations. In the TLM mesh, boundaries must be orientated either horizontally or vertically and placed halfway between adjacent nodes in order to fulfill the underlying time synchronism condition.

2.2.1 Perfect Electric and Magnetic Boundaries

Perfect electric and magnetic boundaries reflect voltage impulses completely. Instead of propagating to a neighbour node, an impulse hitting these kinds of boundary will be reflected back to the original node. The reflection coefficient, $\Gamma = v_n^r / v_n^i$, depends on the boundary conductivity; $\Gamma = -1$ for perfect electric boundaries whereas $\Gamma = +1$ for perfect magnetic boundaries, Figure 8.

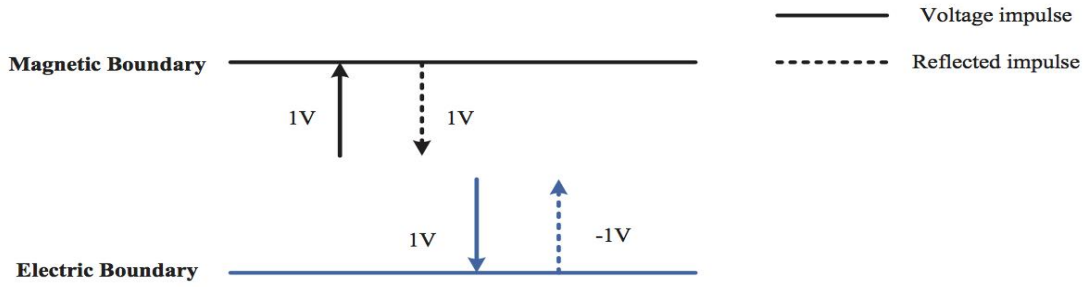


Figure 8: Reflection at perfect magnetic and electrical boundaries.

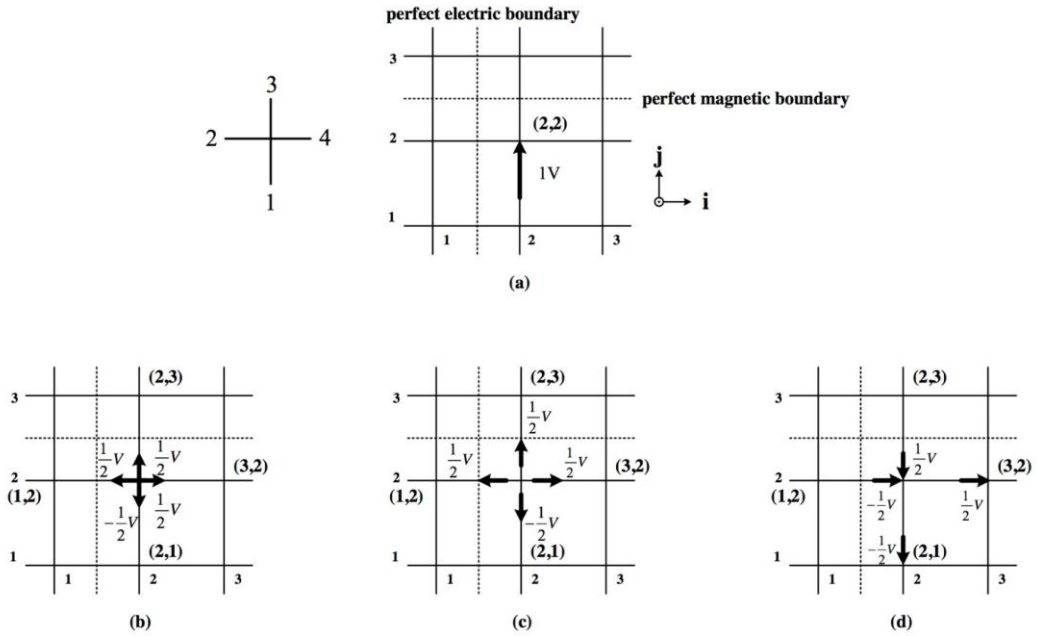


Figure 9: Apply the TLM method to the perfect boundaries scenario.
(a) An incident voltage impulse is launched into $node_{2,2}$.
(b) Impulse gets scattered at $node_{2,2}$.
(c) Scattered impulses are transferred to neighbouring nodes.
(d) Scattered impulses at link lines 2 and 3 hit the boundaries and get reflected.

Figure 9 shows an example of the scattered impulses hitting two perfect boundaries during the transfer process. In the figure, a perfect magnetic boundary at halfway between $node_{2,2}$ and $node_{2,3}$ and a perfect electric boundary at halfway between $node_{1,2}$ and $node_{2,2}$. A voltage impulse incident on to $node_{2,2}$ from link line 1 get scattered into four impulses. The scattered impulses in link line 2 hits the electric boundary and reflect back to $node_{2,2}$ with a value of $-\frac{1}{2}V$. On the other hand, the scattered impulses in link line 3 hits the magnetic boundary and reflect back to $node_{2,2}$ with a value of $\frac{1}{2}V$. The impulses on link lines 1 and 4 will propagate continuously to $node_{2,1}$ and $node_{3,2}$ since there is no boundary blocking the impulse propagation. The algorithm that handles impulse transfer and reflection in the TLM mesh is show in Figure 10.

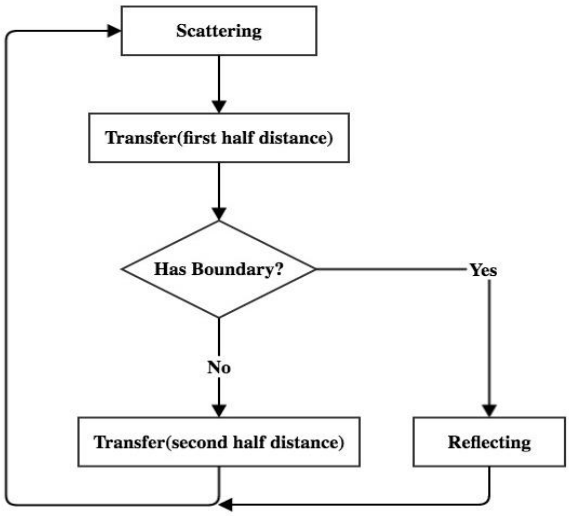


Figure 10: The TLM method with perfect boundaries in one time step.

2.2.2 Dielectric Boundaries for TM modeling

The equivalence material given in Eqs (2.2) to (2.4) are valid for TE wave modeling. Nevertheless, the shunt node TLM method may be used for TM wave modeling as well if the mesh voltages are used to represent magnetic field in the problem space. In that situation, the boundary condition across two dielectric regions must be corrected to ensure the continuity of magnetic field across the interface [Tatsuo]. Without going into another set of equivalence equations similar to that of Eqs (2.2) to (2.4) we just state the algorithm in Figure 11 and Eqs (2.8) to (2.11).

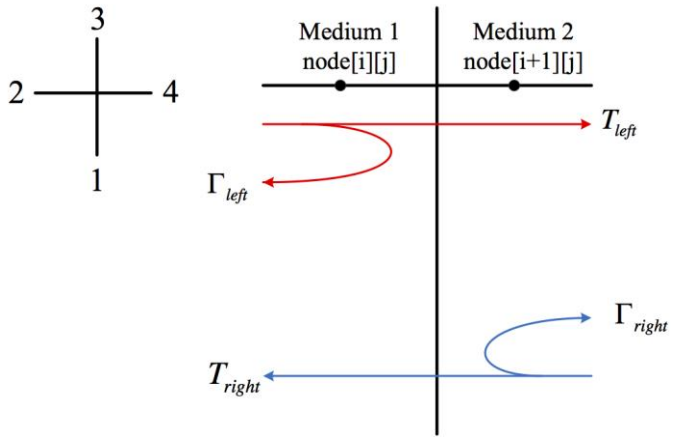


Figure 11: Transmission and reflection coefficients at a dielectric interface for modelling of TM wave behaviour.

The transmission (τ) and reflection coefficients (Γ) in medium 1 and medium 2 are [4]:

$$\Gamma_1 = \frac{1-r}{1+r}, \qquad \Gamma_2 = \frac{r-1}{r+1}, \qquad \tau_1 = \frac{2}{1+r}, \qquad \tau_2 = \frac{2r}{1+r}$$

where $\boldsymbol{r = \frac{\varepsilon_1}{\varepsilon_2} = \frac{instrinsic\ impedance\ of\ Medium\ 1}{instrinsic\ impedance\ of\ Medium\ 2}}$.

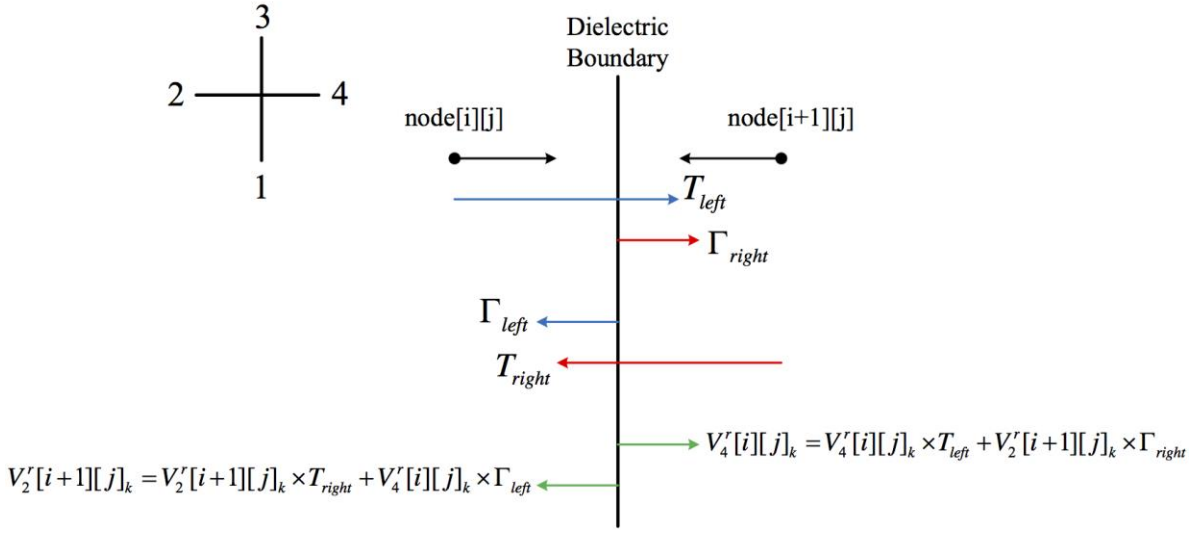


Figure 12: Impulses transmission and reflection after hitting the dielectric boundary.

Referring to Figure 12, the impulses in link line 2 at $node_{i+1,j}$ and link line 4 at $node_{i,j}$ in time step k^{th} are crossing a dielectric interface. The resulting voltages are:

$$V_4^r[i][j]_k = V_4^r[i][j]_k \cdot \tau_{1(left)} + V_2^r[i+1][j]_k \cdot \Gamma_{2(right)}. \tag{2.8}$$

$$V_2^r[i+1][j]_k = V_1^r[i+1][j]_k \cdot \tau_{2(right)} + V_4^r[i][j]_k \cdot \Gamma_{1(left)}. \tag{2.9}$$

The algorithm to modify scattered impulses at dielectric interfaces is shown in Figure 13.

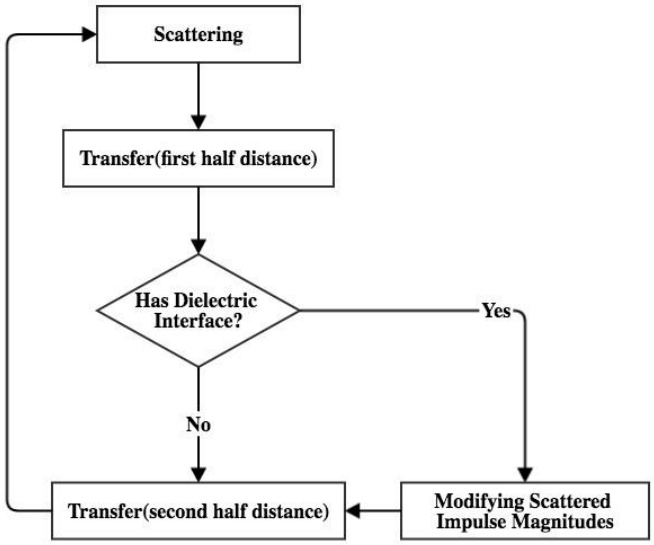


Figure 13: The TLM method with dielectric interfaces in one time step.

2.3 Transfer Processes in a Shunt-Connected TLM Network

The TLM impulse transfer process has been mentioned in Section 2.1 and Section 2.2 but the implementation detail is yet to be discussed. In a real transmission line network voltage impulses would naturally propagate to the adjacent nodes, or back to the scattering nodes if reflected by boundaries, and act as incident impulses in the ensuing scattering process. The following sections describe the algorithms developed to perform this impulse transfer process on a computer program.

2.3.1 Interchanging Impulse Values between Neighbouring nodes

Since each TLM node has four link lines and one permittivity stub, five two-dimensional arrays are used to store the impulse values in the TLM mesh. The state of these impulses change back and forth from incident to scattered as the simulation progress in time. At the beginning of a TLM iteration, say at time $t = k \Delta t$, all values in the arrays are considered as incident impulses for the scattering algorithm. Once the scattering process is completed, all values in the arrays become scattered impulses for the transfer process. Upon completion of the transfer process, the values in the arrays become incident impulses again for the next TLM iteration at time $t = (k + 1) \Delta t$.

The impulse transfer process applies the data interchange procedure shown in Figure 14 to each nodes in the TLM mesh. The impulse in link line 3 at $node_{i,j}$ is swapped with the one in link line 1 at $node_{i,j+1}$. Similarly, the impulse in link line 4 at $node_{i,j}$ is swapped with the one in link line 2 at $node_{i+1,j}$. This interchanging process can be expressed mathematically as:

$$V_3^i[i][j]_{k+1} = V_1^r[i][j+1]_k \quad , \quad V_4^i[i][j]_{k+1} = V_2^r[i+1][j]_k \tag{2.10}$$

$$V_1^i[i][j+1]_{k+1} = V_3^r[i][j]_k \quad , \quad V_2^i[i+1][j]_{k+1} = V_4^r[i][j]_k \tag{2.11}$$

Note that the impulse on the open-circuited permittivity stub is reflected at the open end becomes incident an impulse. Since the reflection coefficient for an open-circuited stub is $\Gamma = 1$, the value for $V_5^i[i][j]_{k+1} = \Gamma \times V_5^r[i][j]_k = V_5^r[i][j]_k$. Therefore the value in the permittivity stub requires no actual update.

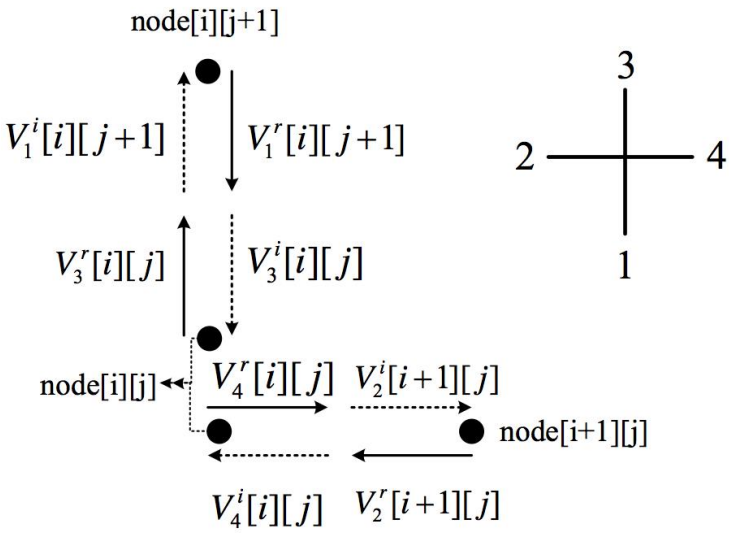


Figure 14: Interchanging among nodes at link lines 1-4.

Algorithm 2: Interchanging voltage impulses between neighbouring nodes

computationBox number = kc
for *nn* = 1 : *kc* **do**
  tmp = $hashmap_{computationBox}$.*get(nn)*
  for *j* = *tmp[3]*-1 : *tmp[4]*-1 **do**
    for *i* = *tmp[1]*-1 : *tmp[2]*-1 **do**
      $a = V_3^r[i][j]$
      $V_3^i[i][j] = V_1^r[i][j+1]$
      $V_1^i[i][j+1] = a$
      $a = V_4^r[i][j]$
      $V_4^i[i][j] = V_2^r[i+1][j]$
      $V_2^r[i+1][j] = a$
    end for
  end for
end for

Table 2: Algorithm of Interchanging voltage impulses between neighbouring nodes.

The impulse interchange procedure given in Eqs (2.10) and (2.11) must be applied to all nodes in the TLM mesh in order to complete the TLM impulse transfer process. The pseudo code in Algorithm 2 implements the transfer process for a boundary-free mesh.

2.3.2 Adjusting Impulse Values Adjacent to Boundaries

The algorithms to handle boundaries and dielectric interfaces in the TLM mesh have been presented in Section 2.2. These algorithms must be executed first with the reflected impulses placed in the neighbour nodes ahead of the anticipated impulse interchange operations in the impulse transfer process. The transfer process will then restore the reflected impulses to the intended link lines. The formulas for handling perfect reflecting boundaries, i.e. $\Gamma = 1$ or $\Gamma = -1$, are:

$$V_3^r[i][j] = V_1^r[i][j+1] \cdot \Gamma \quad V_1^r[i][j+1] = V_3^r[i][j] \cdot \Gamma \tag{2.12}$$

$$V_4^r[i][j] = V_2^r[i+1][j] \cdot \Gamma \quad V_2^r[i+1][j] = V_4^r[i][j] \cdot \Gamma \tag{2.13}$$

The pseudo code that applies the algorithm to the TLM mesh is:

Algorithm 3: Adjusting Impulse Values Adjacent to Perfect Boundaries

electric and magnetic boundary number = kb
for *nn* = 1 : *kb* **do**
  tmp = $hashmap_{boundary}$.*get(nn)*
  r = $hashmap_{reflectCoeff}$.*get(nn)*
  for *j* = *tmp[3]*-1 : *tmp[4]*-1 **do**
  for *i* = *tmp[1]*-1 : *tmp[2]*-1 **do**
  $V_{xy} = V_3^r[i][j]$
  $V_3^r[i][j] = V_1^r[i][j+1] \cdot r$
  $V_1^r[i][j+1] = V_{xy} \cdot r$
  end for
  end for
end for

Table 3: Algorithm of adjusting impulse values adjacent to perfect boundaries

Dielectric interfaces described in Eqs (2.8) and (2.9) are to keep the continuity of magnetic field across the interface, the input will be only $\Gamma_{1(left)}$ for vertical boundaries or $\Gamma_{1(below)}$ for horizontal boundaries. If the vertical boundary in Figure 11 is rotated 90° counter-clockwise, it would become a horizontal boundary. Therefore, Eq(2.14) to Eq(2.17) are applicable to $\Gamma_{1(left)}$ or $\Gamma_{1(below)}$ depending on the orientation of the boundary.

$$\Gamma_{1(left,below)} = \frac{1-r}{1+r} = \frac{1-\frac{\varepsilon_1}{\varepsilon_2}}{1+\frac{\varepsilon_1}{\varepsilon_2}} = \frac{\varepsilon_2-\varepsilon_1}{\varepsilon_2+\varepsilon_1}, \tag{2.14}$$

$$\tau_{1(left,below)} = \frac{2}{1+r} = 1 + \frac{1-r}{1+r} = 1 + \Gamma_{1(left,below)}, \tag{2.15}$$

$$\Gamma_{2(right,above)} = \frac{r-1}{r+1} = -\frac{1-r}{1+r} = -\Gamma_{1(left,below)}, \tag{2.16}$$

$$\tau_{2(right,above)} = \frac{2r}{1+r} = 1 - \frac{1-r}{1+r} = 1 - \Gamma_{1(left,below)}. \tag{2.17}$$

With the results in Eqs (2.14)-(2.17), Eq (2.8) and Eq (2.9) could be rewritten as:

$$V_4^r[i][j]_k = V_4^r[i][j]_k \cdot (1 + \Gamma_{left}) + V_2^r[i+1][j]_k \cdot (-\Gamma_{left}). \tag{2.18}$$

$$V_2^r[i+1][j]_k = V_1^r[i+1][j]_k \cdot (1 - \Gamma_{left}) + V_4^r[i][j]_k \cdot \Gamma_{left}. \tag{2.19}$$

The pseudo code for impulse modification at a dielectric interface is:

Algorithm 4: Impulse Modification at a Dielectric Interface

dielectric boundary number = kd
for *nn* = 1 : *kd* **do**
  tmp = $hashmap_{boundary}.get(nn)$
  r = $hashmap_{reflectCoeff}.get(nn)$
  for *j* = *tmp[3]*-1 : *tmp[4]*-1 **do**
  for *i* = *tmp[1]*-1 : *tmp[2]*-1 **do**
  $v_x = V_4^r[i][j]$
  $v_y = V_2^r[i+1][j]$
  $V_4^r[i][j] = v_x \cdot (1+r) + v_y \cdot (-r)$
  $V_2^r[i+1][j] = v_y \cdot (1-r) + v_x \cdot r$
  end for
  end for
end for

Table 4: Algorithm of impulse modification at a dielectric interface

The complete algorithms for the TLM method is shown in Figure 15:

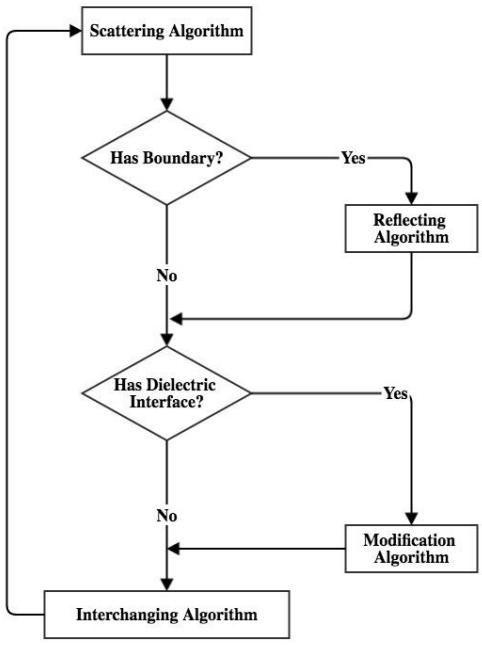


Figure 15: The complete algorithm of the TLM method.

Chapter 3. Application Layout and Widget Implementations

This application is implemented using Android's "*Tabbed Activity*" template. The default template offers three tabs, a navigation, and a floating application bar. To maximize the viewing area in each tab, the navigation and application bar are deleted. Each tab has a specific group of functions and output, such as editing the TLM network or displaying the simulation result.

3.1 Main Activity Initialization

Android maintains a stack which contains the *activity* instances from different applications. The instance at the top of the stack is the currently running application. Therefore, when launching the TLM app, Android instantiates the *MainActivity* class and adds the new instance to the top of the activity stack. The *activity* becomes active after executing the *onCreate*, *onStart*, and *onResume* methods. In the *onCreate* method, the *activity* creates a window for the User Interface (UI) via the *setContentView* method. The widget objects in this layout can be retrieved by the *findViewById* method. Once a view is found, more functions can be associated with that view. For example, when a button object is created in the *onCreate()* method, a *ClickListener* instance, which defines the response when clicking on the button, could be assigned to that view. Generally, the *onCreate* method only performs initialization work, other features should be implemented outside of the *onCreate* method. The pseudo code of the *onCreate* method in the *MainActivity* class is shown in Listing 1. The UI of this new *activity* instance has five widgets as shown in Figure 16. The widgets are respectively the tab layout, the button widgets, the list view, the text view and the view of a *fragment* instance.

```
import android.support.design.widget.TabLayout;
import android.widget.Button;
import android.widget.ListView;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {

      super.onCreate(savedInstanceState);
         setContentView(R.layout.activity_main);
         TabLayout tabLayout = (TabLayout) findViewById(R.id.tabs);
         mesh_btn = (Button) findViewById(R.id.mesh);
         wall_lv = (ListView) findViewById(R.id.lv);
         list_tv = (TextView) findViewById(R.id.lv_tv);
         ..........
  }
    ..........
}
```

Listing 1: The pseudo code of the *onCreate* method in the *activity* instance.

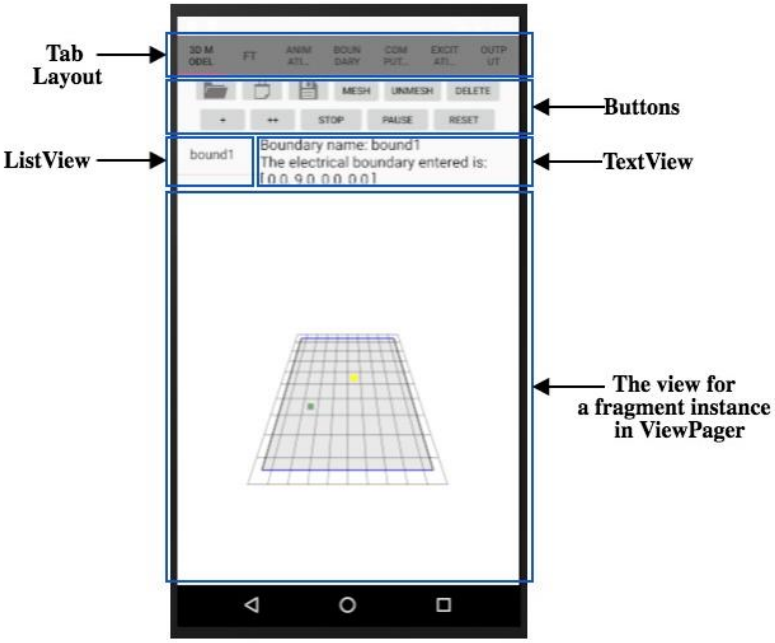


Figure 16: The UI when the application is brought up.

The tab layout in Figure 16 displays seven tab names in this app. As each tab has its own layout and independent functions, the tab is implemented by a *fragment* instance which could define the layout and manage its own lifecycle within the *activity* instance. Similar to the *activity*, a *fragment* instance becomes active after executing the *onAttach*, *onCreate*, *onCreateView*, *onActivityCreated*, *onStart* and *onResume* methods where the *onAttach* method nests the *fragment* instance to the *activity* instance and the *onCreateView* method

does the initialization work. Listing 2 uses an “Animation” tab as an example to explain the implementation of a *fragment* class.

```
package com.example.apple.neweditortab;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.Button;
import android.widget.EditText;


public class Animation extends Fragment{
    Button add_btn, edit_btn;
    EditText delta;

    public static Animation newInstance() {
        Animation fragment = new Animation();
        return fragment;
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle
                             savedInstanceState) {
        View rootView = inflater.inflate(R.layout.animationregion, container, false);

        add_btn = (Button) rootView.findViewById(R.id.add_reg);
        edit_btn = (Button) rootView.findViewById(R.id.edit_reg);
        delta = (EditText) rootView.findViewById(R.id.delta);
        ……

        return rootView;
    }
}
```

Listing 2: "Animation" *fragment* class.

Following the same implementation process, seven *fragment* classes are created to display the TLM simulation and edit the TLM network. The swiping effect among different *fragments* is implemented by a *ViewPager* widget. This widget has a standard adapter instantiated from a *FragmentPagerAdapter* class. The adapter is employed to generate and manage seven *fragment* instances by overriding the *getItem* method. This method returns the *fragment* instance if the *position* is valid. For example, when the third tab is swiped, the *position* is 2. The *ViewPager* gets this *position* and passes it to the adapter. As the adapter in this app defines the *getItem* method to return an *Animation fragment* instance at *position* 2, the *ViewPager* renders the view of an *Animation fragment* at the third tab. On the other hand, the *ViewPager* widget is also applied to the tab layout which displays *fragment titles* instantiated in the *getItem* method. To define titles, the adapter overrides the *getPageTitle* method which should keep consistent with the *getItem* method. For

example, the third tab is the *Animation fragment* instance. Then the title for the third tab is defined as "Animation". The complete code to implement *ViewPager*, *FragmentAdapter* and *TabLayout* in the *MainActivity* class is shown in Listing 3.

```
public class MainActivity extends AppCompatActivity {
    protected void onCreate(Bundle savedInstanceState) {
         mSectionsPagerAdapter = new SectionsPagerAdapter(getSupportFragmentManager());
         mViewPager = (ViewPager) findViewById(R.id.container);
         mViewPager.setAdapter(mSectionsPagerAdapter);
         TabLayout tabLayout = (TabLayout) findViewById(R.id.tabs);
         tabLayout.setupWithViewPager(mViewPager);
         ……
    }
    public class SectionsPagerAdapter extends FragmentPagerAdapter {
        public SectionsPagerAdapter(FragmentManager fm) {
            super(fm);
        }
        @Override
        public Fragment getItem(int position) {
            switch (position) {
                case 0:
                    return TLMGL.newInstance();
                case 1:
                    return FT_Display.newInstance();
                case 2:
                    return Animation.newInstance();
                case 3:
                    return Boundary.newInstance();
                case 4:
                    return Computation.newInstance();
                case 5:
                    return Excitation.newInstance();
                case 6:
                    return OutputSetting.newInstance();
            }
            return null;
        }
        @Override
        public CharSequence getPageTitle(int position) {
            switch (position) {
                case 0:
                    return "3D Model";
                case 1:
                    return "FT";
                case 2:
                    return "Animation";
                case 3:
                    return "Boundary";
                case 4:
                    return "Computation";
                case 5:
                    return "Excitation";
                case 6:
                    return "Output";
            }
            return null;
        }
    }
}
```

Listing 3: Implement *ViewPager*, *FragmentPagerAdapter* and *TabLayout*.

This section introduces the code segment that constructs and manages the *activity* and *fragment* instances. Co-operating between *activity* and *fragment* provides the fundamental behavior to the application. Next section describes the widget implementation in the *activity* and *fragment* modules to make the application functional.

3.2 Data Transmission and Gesture Detection

Section 3.1 introduces the structure built by *activity* and *fragment*. Based on the structure, this section explains the data transmission from *fragment* to *activity* and describes how to set gesture detector to respond to the action on the touch screen as well.

3.2.1 Data Transmission from UI to Back-end

As one of the main functions in this app is to edit the TLM networks, five tabs are designed to input the TLM component information. For example, in "Boundary" tab, users are supposed to enter the boundary position with real-world coordinates. In this case, the *fragment* class employs *EditText* widgets which allow users to edit the boundary coordinates. "Boundary" tab also has a button named "ADD". The button object sets a *ClickListener* instance to monitor the clicking action and the overridden *onClick* method in the listener defines the response for the button clicking. For "Boundary" tab, the *onClick* method reads the context edited in *EditText* objects and parses the context to numbers. These parsed numbers are transmitted back to the *MainActivity* via an interface. The *MainActivity* class implements the abstract method in the interface to process and store the transmitted data.

Listing 4 shows the *ClickListener* and interface implementation in "Boundary" tab. In fact, except for tabs to display the TLM simulation and the evolution of DFT, other tabs follow the same process to transmit the input data to the back-end.

```
import android.widget.Button;
import android.widget.EditText;

public class Boundary extends Fragment {
    Button add_bound_btn;
    EditText name, xmin, xmax, ymin, ymax, reflect_coeff;
    ……

    public interface returnBoundary{
        abstract void getBoundary(String name, double xmin, double xmax, double ymin,
                                  double ymax, double reflect_coeff);
    }
```

```
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle
                             savedInstanceState) {
        View rootView = inflater.inflate(R.layout.boundary, container, false);
        name = (EditText) rootView.findViewById(R.id.name_b);
        xmin = (EditText) rootView.findViewById(R.id.xmin_b);
        ……
        add_bound_btn = (Button) rootView.findViewById(R.id.add_b);
        add_bound_btn.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                String name_c;
                double xmin_c, xmax_c, ymin_c, ymax_c;
                double reflect_coeff_c;
                if (!name.getText().toString().isEmpty()){
                    name_c = name.getText().toString();
                } else {
                    name_c = "";
                }
                if (!xmin.getText().toString().isEmpty()) {
                    xmin_c = Double.parseDouble(xmin.getText().toString());
                } else {
                    xmin_c = -1.0;
                }
                ……
                returnBoundary returnboundary = (returnBoundary) getActivity();
                returnboundary.getBoundary(name_c, xmin_c, xmax_c, ymin_c, ymax_c,
                                           reflect_coeff_c);

                name.setText("");
                xmin.setText("");
                ……
            }
        });
        return rootView;
    }
}
```

```
public class MainActivity extends AppCompatActivity implements Boundary.returnBoundary{
    ……
    @Override
    public void getBoundary(String name, double xmin, double xmax, double ymin, double
                            ymax, double reflect_coeff) {
        double[] tmp_boundary;
        if (reflect_coeff == -1) {
            if (!e_boundary.containsKey(name)) {
                wall_name.add(name);
            }
            tmp_boundary = new double[]{xmin, xmax, ymin, ymax, 0.0, 0.0, 1.0, 1.0};
            e_boundary.put(name, tmp_boundary);
        } else if (reflect_coeff == 1) {
            if (!m_boundary.containsKey(name)) {
                wall_name.add(name);
            }
            tmp_boundary = new double[]{xmin, xmax, ymin, ymax, 0.0, 0.0, 0.0, 1.0};
            m_boundary.put(name, tmp_boundary);
        } else {
            if (!adi_boundary.containsKey(name)) {
                wall_name.add(name);
            }
            tmp_boundary = new double[]{xmin, xmax, ymin, ymax, 1.0, 0.0, 0.0, 1.0};
            adi_boundary.put(name, tmp_boundary);
            adi_coeff.put(name, reflect_coeff);
        }
    }
    ……
}
```

Listing 4: Implementation of the *ClickListener* and the interface.

3.2.2 Multi-Gesture Detection

"3D Model" *fragment* employs the multi-gesture detection to sense the multiple finger actions on the touch screen. The multi-gesture detection is implemented by a *GestureDetector* instance which works like a *Button* widget. To respond the gesture action, a *GestureListener* instance is also applied to the *GestureDetector*. The *GestureListener* implements two interfaces to respond the single-tapping and double-tapping actions via respectively overriding *onDown* and *onDoubleTap* methods. Listing 5 shows the code to implement the *GestureDetector* and *GestureListener*.

```
import android.view.GestureDetector;

public class TLMGL extends Fragment{
    protected GestureDetector tlmGestureDetector;
    ……
    public View onCreateView(LayoutInflater inflater,ViewGroup container,Bundle
                             savedInstanceState) {
        ……
        tlmGestureDetector = new GestureDetector(getActivity(), new
                                                 TLMGestureListener(tlmglRenderer));
        ……
    }
    ……
}
```

```
import android.view.GestureDetector;

public class TLMGestureListener implements GestureDetector.OnGestureListener,
                                           GestureDetector.OnDoubleTapListener {

    ……
    public TLMGestureListener(TLMGLRenderer tlmglRenderer){
        this.tlmglRenderer = tlmglRenderer;
    }
    ……
    @Override
    public boolean onDown(MotionEvent e) {
        float x = e.getRawX();
        float y = e.getRawY();

        tlmglRenderer.mLastTouchX = x;
        tlmglRenderer.mLastTouchY = y;
        tlmglRenderer.handleTouchDown(x - TLMGL.screenLocation[0],
                                      y – TLMGL.screenLocation[1]);
        return true;
    }
    @Override
    public boolean onDoubleTap(MotionEvent e) {
        tlmglRenderer.handleDoubelTap();
        return true;
    }
    ……
}
```

```
public class TLMGLRenderer implements GLSurfaceView.Renderer {
    public void handleTouchDown(float xevent, float yevent) {……}
    public void handleDoubelTap() {……}
    public void handleMove(float x, float y) {……}
    ……
}
```

Listing 5: Implement the *GestureDetector* and *GestureListener*.

In “3D Model” fragment, the *view* object is created by a *GLSurfaceView* instance instead of a layout template. This part is introduced in Chapter 4. Because the *GestureListener* only responds to tapping events, the *GLSurfaceView* instance in “3D Model” *fragment* is assigned with an *OnTouchListener* instance to detect the finger-moving events. The *OnTouchListener* overrides the *onTouch* method to define the response for different action types such as *ACTION_MOVE* or *ACTION_SCROLL*. This app only has the *ACTION_MOVE* response implemented as shown in Listing 6.

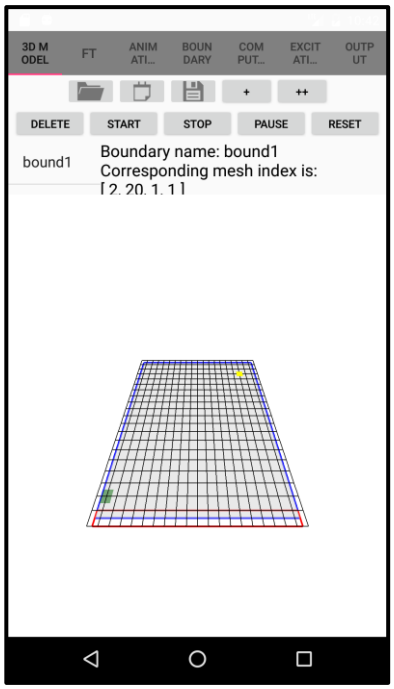
```
public class TLMGL extends Fragment{
    ……
    public View onCreateView(LayoutInflater inflater,ViewGroup containner,Bundle
                             savedInstanceState) {
        tlmglSurfaceView = new TLMGLSurfaceView(getActivity());
        ……
        tlmGestureDetector = new GestureDetector(getActivity(), new
                                                 TLMGestureListener(tlmglRenderer));
        tlmglSurfaceView.setOnTouchListener(new View.OnTouchListener() {
            @Override
            public boolean onTouch(View v, MotionEvent event) {
                v.getLocationOnScreen(screenLocation);
                tlmGestureDetector.onTouchEvent(event);

                if (event.getAction() == MotionEvent.ACTION_MOVE){
                    final float x = event.getRawX();
                    final float y = event.getRawY();
                    tlmglRenderer.handleMove(x, y);
                }
                return true;
            }
        });
        ……
    }
    ……
}
```

Listing 6: *OnTouchListener* implementation in “3D Model”.

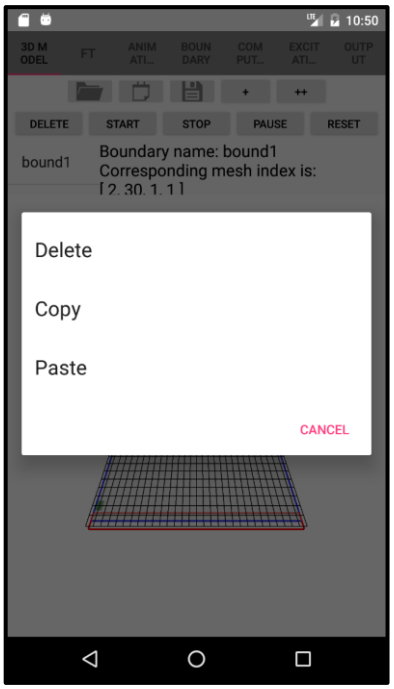
Codes in Listing 5 and 6 explain the implementation of the gesture detection in this app. This app can give feedback for the single tapping, double tapping and single-finger moving event. The effect is shown in Figure 17. Single tapping is to choose a boundary as shown in (a). The *onDown* method in the *GestureListener* is invoked if one finger presses down on the screen. This method detects the coordinate of the touch location which is applied to do the crossing test. The boundary is selected if passing the test and a red rectangular is added around the chosen boundary. Once a boundary is selected, double tapping on it alerts a dialog to provide an operation list (Figure 17 (b)). The operations are respectively deleting, coping or pasting the selected boundary. The moving event in this app is to move the chosen boundary as shown in (c) and (d). On moving the selected boundary, the

boundary coordinate in real world is also updated in the *tlmglRenderer.handleMove* method. The complete code is provided in Appendix.

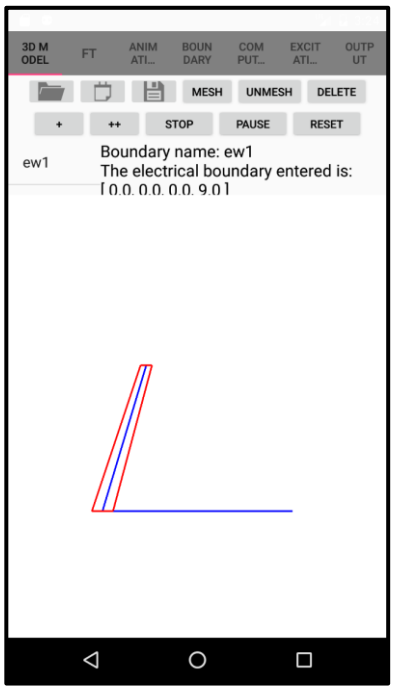
Implementing various types of widgets and multi-gesture detection gives a complete function of editing the TLM network. Users can save the TLM network by clicking the "SAVE" button. When re-launching the application, users could also import the file containing the TLM information via the "  " button.



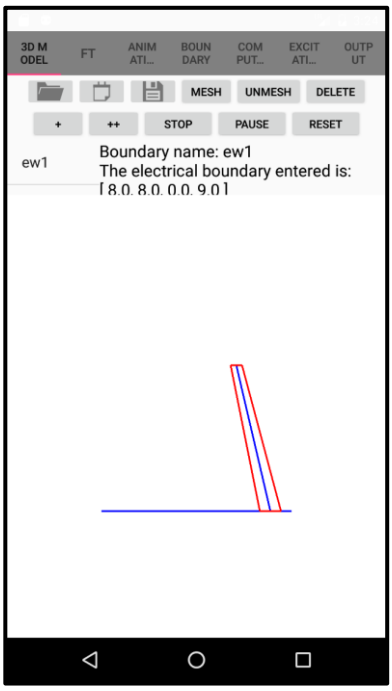
(a) Select a boundary by single tapping



(b) Alert a dialog by double tapping



(c) The selected boundary before moving



(d) The selected boundary after moving

Figure 17: Multi-gesture detection effect.

Chapter 4. OpenGL Rendering Mechanism

Different from the tabs for editing functions, the *view* object in "3D Model" *fragment* is from a *GLSurfaceView* instance instead of a layout template. The *GLSurfaceView* renders a view via the frame assembled by OpenGL. This chapter introduces the implementation of the app's "3D Model" *fragment* using OpenGL Embedded System 2.0.

4.1 Set the Renderer in GLSurfaceView

In "3D Model" fragment, the *GLSurfaceView* employs a *Renderer* instance to render the frame assembled by OpenGL. The *Renderer* has three overridden methods – *onSurfaceCreated*, *onSurfaceChanged* and *onDrawFrame*. The *onSurfaceCreated* method is executed when the *GLSurfaceView* instance is created or recreated. The initialization for the *view* instance is implemented in this method, such as setting the background color. The *onSurfaceChanged* method is executed when the screen orientation changes. In this method, the matrices applied to do the projection are created because they are determined by the height and width of the screen. In the *onDrawFrame* method, OpenGL assembles and renders the frames. Since OpenGL keeps assembling the frames when the app is running, the *onDrawFrame* method is executed repeatedly while the app is active. Listing 7 shows the initialization for a *GLSurfaceView* class and a *Renderer* class in "3D Model" *fragment*.

```
public class TLMGL extends Fragment{
    ……
    public View onCreateView(LayoutInflater inflater,ViewGroup container,Bundle
                             savedInstanceState) {
        tlmglSurfaceView = new TLMGLSurfaceView(getActivity());
        tlmglRenderer = new TLMGLRenderer(getActivity());
        tlmglSurfaceView.setEGLContextClientVersion(2);
        tlmglSurfaceView.setRenderer(tlmglRenderer);
        ……
        return tlmglSurfaceView;
    }
    ……
}
```

```
public class TLMGLSurfaceView extends GLSurfaceView {
    public TLMGLSurfaceView(Context context) {
        super(context);
    }
}
```

```
public class TLMGLRenderer implements GLSurfaceView.Renderer {
    @Override
    public void onSurfaceCreated(GL10 gl, EGLConfig config) {
        glClearColor(1.0f, 1.0f, 1.0f, 0.0f);
        ……
    }
```

```
    @Override
    public void onSurfaceChanged(GL10 gl, int width, int height) {
        glViewport(0, 0, width, height);
        ……
    }
    @Override
    public void onDrawFrame(GL10 gl) {……}
}
```

Listing 7: Initialize a *GLSurfaceView* class and a *Renderer* class in "3D Model" *fragment*.

4.2 Matrix Assignment

OpenGL employs matrix multiplications to project a 3D object on a 2D screen. Figure 18 shows four matrices applied to the projection process. These matrices are assigned in the *OnSurfaceChanged* method since they are determined by the aspect ratio of the screen. When implementing the *onSurfaceChanged* method in the *Renderer* class, the *glViewport* method gets *Viewport Matrix*; *android.opengl.Matrix* library provides the methods to directly assign values to *Projection Matrix* and *Model Matrix*. *View Matrix* is ignored in the *Renderer* class because *View Matrix* is combined with *Model Matrix* in the real implementation [1]. Figure 18 also illustrates the multiplication order and the effect order on the screen. Following the multiplication order in Figure 18, the matrices in the *OnSurfaceChanged* method are initialized as shown in Listing 8.

```
import static android.opengl.Matrix.*;

public class TLMGLRenderer implements GLSurfaceView.Renderer {
    private int[] viewportMatrix;
    private float[] projectionMatrix = new float[16];
    private float[] modelMatrix = new float[16];
    protected static float[] pmMatrix = new float[16];
    ……
    @Override
    public void onSurfaceChanged(GL10 gl, int width, int height) {
        glViewport(0, 0, width, height);
        perspectiveM(projectionMatrix, 0, 90f, (float) width / height, 1f, 10f);
        setIdentityM(modelMatrix, 0);
        translateM(modelMatrix, 0, 0f, 0f, -2f);
        multiplyMM(pmMatrix, 0, projectionMatrix, 0, modelMatrix, 0);
    }
    ……
}
```

Listing 8: Matrix initialization in the *Render* class.

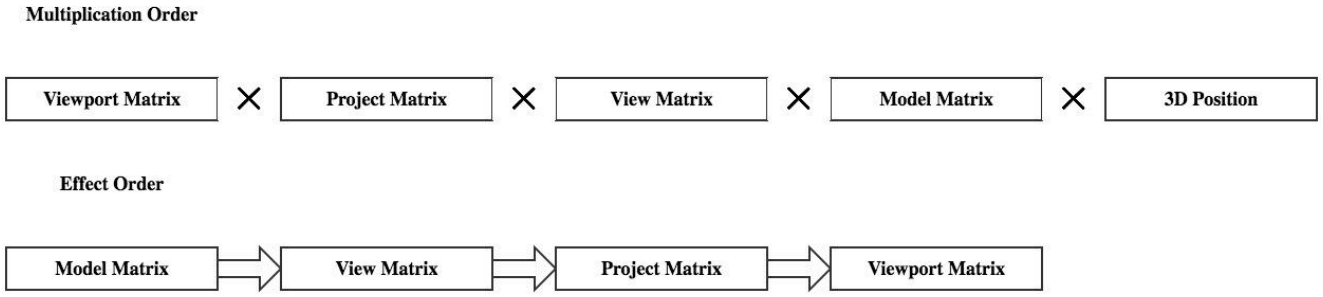


Figure 18: Multiplication order and effect order.

4.3 Create A Program to Compile Vertex and Fragment Shader

Referring to the OpenGL pipeline (shown in Figure 19), OpenGL assembles the primitives first before assembling a complete frame. Figure 20 employs the *ComputationGL* class as an example to introduce the implementation steps of assembling primitives and how to apply the *ComputationGL* object in the *renderer* class. This app also has an *AnimationGL* class for the mesh added among the TLM network, an *ExcitationGL* class for the excitation probes, and an *OutputGL* class for the output probes. The structures and methods for these three classes are similar as the *ComputationGL* class.

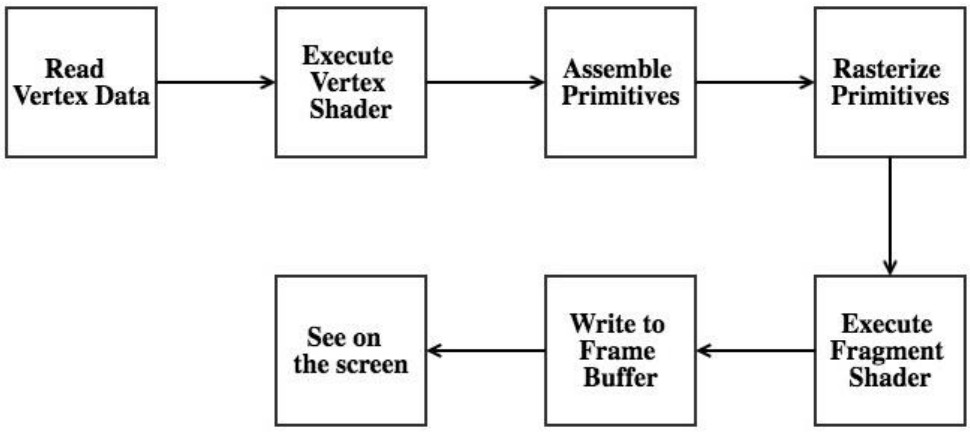
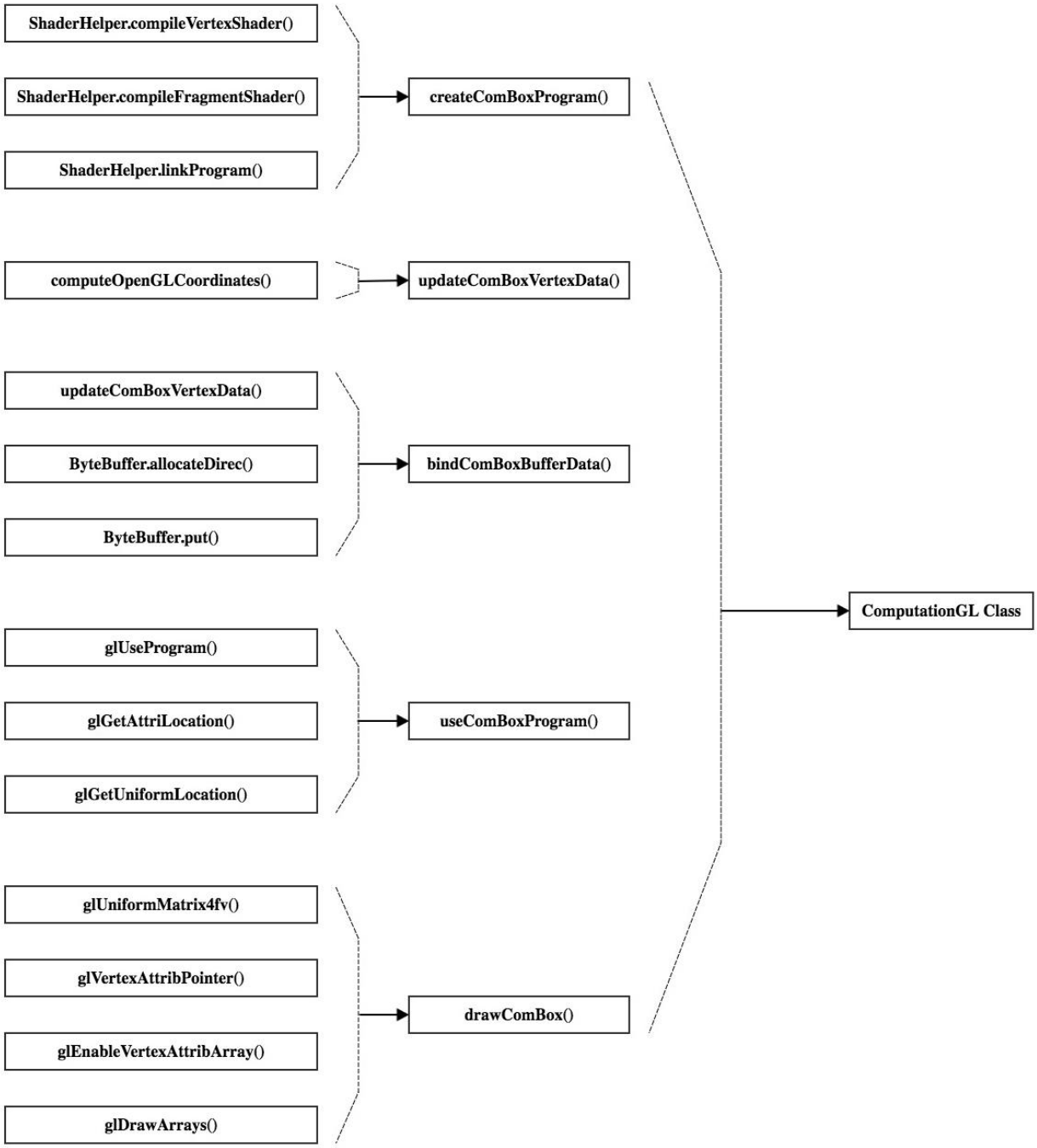
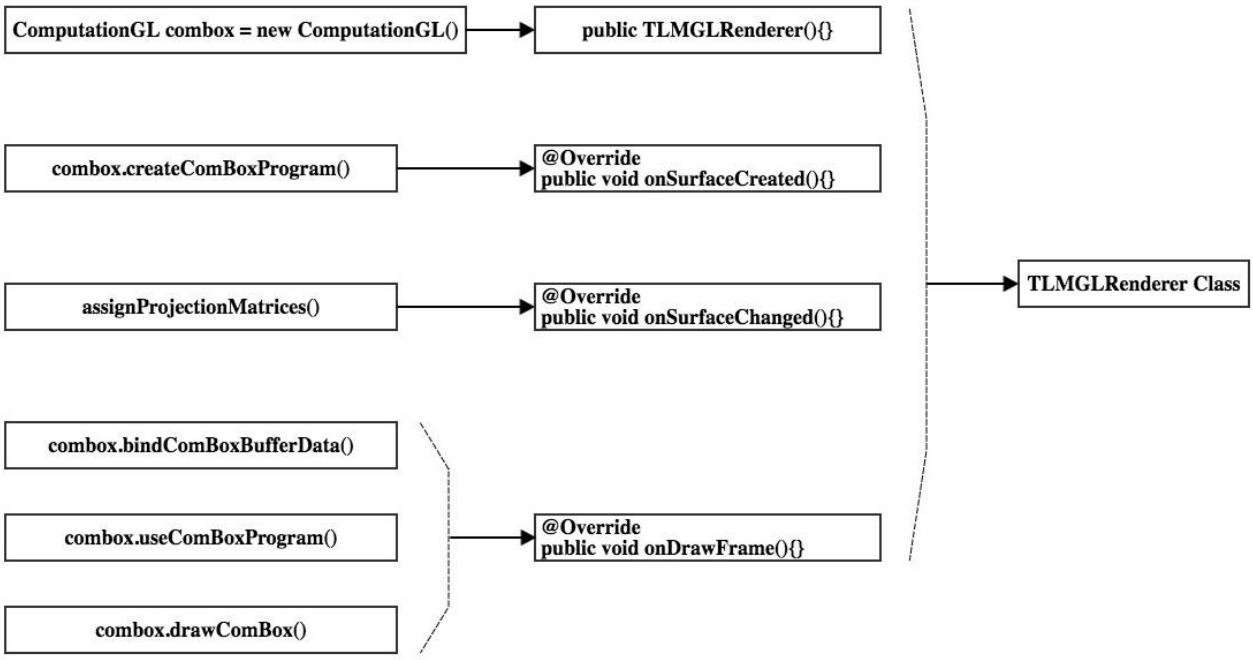


Figure 19: OpenGL ES 2.0 graphics pipeline[1].



(a) The *ComputationGL* class



(b) Apply the *ComputationGL* object to the *renderer* class.

Figure 20: The implementation of assembling primitives in the *ComputationGL* class

This section explains the *createComBoxProgram* method in the ComputationGL class and the rest parts are discussed in Section 4.4 and 4.5. As shown in Figure 19, vertex and fragment shaders are two programmable parts. Before getting vertex data ready, the program linked with vertex and fragment shaders is initialized first. A *ComputationGL* object is instantiated in the *Renderer* constructor. The *createComBoxProgram* method in the object is executed in the *onSurfaceCreated* method shown in Listing 9. Algorithms for other TLM components are encapsulated in different classes and follow the similar process to be initialized.

```
public class TLMGLRenderer implements GLSurfaceView.Renderer {
    public TLMGLRenderer(Context context) {
        this.context = context;
        combox = new ComputationGL();
        ……
    }

    @Override
    public void onSurfaceCreated(GL10 gl, EGLConfig config) {
        ……
        combox.createComBoxProgram();
        ……
    }
    ……
}
```

Listing 9: Initialize the *ComputationGL* to execute the *createComBoxProgram()* in the *renderer*.

The *createComBoxProgram* method implements two steps to create a program instance – compiling the vertex and fragment shaders, and linking the shaders to the program. These implementations are accomplished with the *android.opengl.GLES20* functions shown in Listing 10.

```
import static android.opengl.GLES20.*;
public class ComputationGL {
    ……
    public void createComBoxProgram() {

        String vertexShaderSource = "attribute vec4 a_Position;\n" +
                                    "uniform mat4 u_Matrix;\n" +
                                    "void main(){\n" +
                                    "gl_Position = u_Matrix * a_Position;\n" +
                                    "}";
        String fragmentShaderSource = "precision mediump float;\n" +
                                      "void main(){\n" +
                                      "gl_FragColor = vec4(0.825,0.825,0.825,0.5);\n" +
                                      "}";
        int vertexShader = ShaderHelper.compileVertexShader(vertexShaderSource);
        int fragmentShader = ShaderHelper.compileFragmentShader(fragmentShaderSource);
        com_program = ShaderHelper.linkProgram(vertexShader, fragmentShader);
    }
    ……
}
```

```
import static android.opengl.GLES20.*;

public class ShaderHelper {

    public static int compileVertexShader(String shaderCode){
        return compileShader(GL_VERTEX_SHADER, shaderCode);
    }

    public static int compileFragmentShader(String shaderCode){
        return compileShader(GL_FRAGMENT_SHADER, shaderCode);
    }

    private static int compileShader(int type, String shaderCode){
        final int shaderObjectId = glCreateShader(type);
        glShaderSource(shaderObjectId, shaderCode);
        glCompileShader(shaderObjectId);
       return shaderObjectId;
    }

    public static int linkProgram(int vertexShaderId, int fragmentShaderId) {
        final int programObjectId = glCreateProgram();
        glAttachShader(programObjectId, vertexShaderId);
        glAttachShader(programObjectId, fragmentShaderId);
        glLinkProgram(programObjectId);
        return programObjectId;
}
     ……
}
```

Listing 10: Compile shaders and link shaders to a program.

4.4 Mapping TLM Networks from the Real World to OpenGL

Since OpenGL has the coordinate in the form of $[x, y, z, w]$ and the coordinate should not exceed $[-1, 1]$ in x, y and z when w is $[1, 2]$, the TLM network in real world needs further mapping before doing the projection. This app sets *XMIN*, *YMIN* and *maxLen* variables to implement the mapping process from the real-world coordinates to the OpenGL coordinates. The first reason for employing these variables is the length of TLM components in the real world is dynamic. For example, the first input boundary has the length of 10mm. The second one could even be 100mm. It's necessary to check the maximal length and update *maxLen* accordingly whenever a new boundary is added. *maxLen* optimizes the usage of the screen by making sure that no normalized exceeds $[-1, 1]$. The second reason is that the input data is has absolute position in 2D TLM networks while the OpenGL coordinate for the networks is the relative position to *XMIN* and *YMIN*.

Listing 11 illustrates how *XMIN*, *YMIN* and *maxLen* are updated in the app. Figure 21 shows an example of the mapping process that computes the OpenGL coordinates. In this app, Δl has a default value of 1mm. The example uses the default Δl value, and the initial values for *XMIN*, *YMIN*, and *maxLen* are *Float.MAX_VALUE*, *Float.MAX_VALUE*, and *0.0f*, respectively.

```
TLMGLRenderer.XMIN = (float) Math.min(TLMGLRenderer.XMIN, xmin-delta_l/2.0);
TLMGLRenderer.YMIN = (float) Math.min(TLMGLRenderer.YMIN, ymin-delta_l/2.0);
TLMGLRenderer.maxLen = (float) Math.max(TLMGLRenderer.maxLen,
                                        Math.max(xmax-xmin, ymax-ymin)+delta_l);
TLMGLRenderer.GL_L = (float) 2 / TLMGLRenderer.maxLen;
TLMGLRenderer.GL_w = (float) 1 / TLMGLRenderer.maxLen;
```

Listing 11: Update *XMIN*, *YMIN* and *maxLen*.

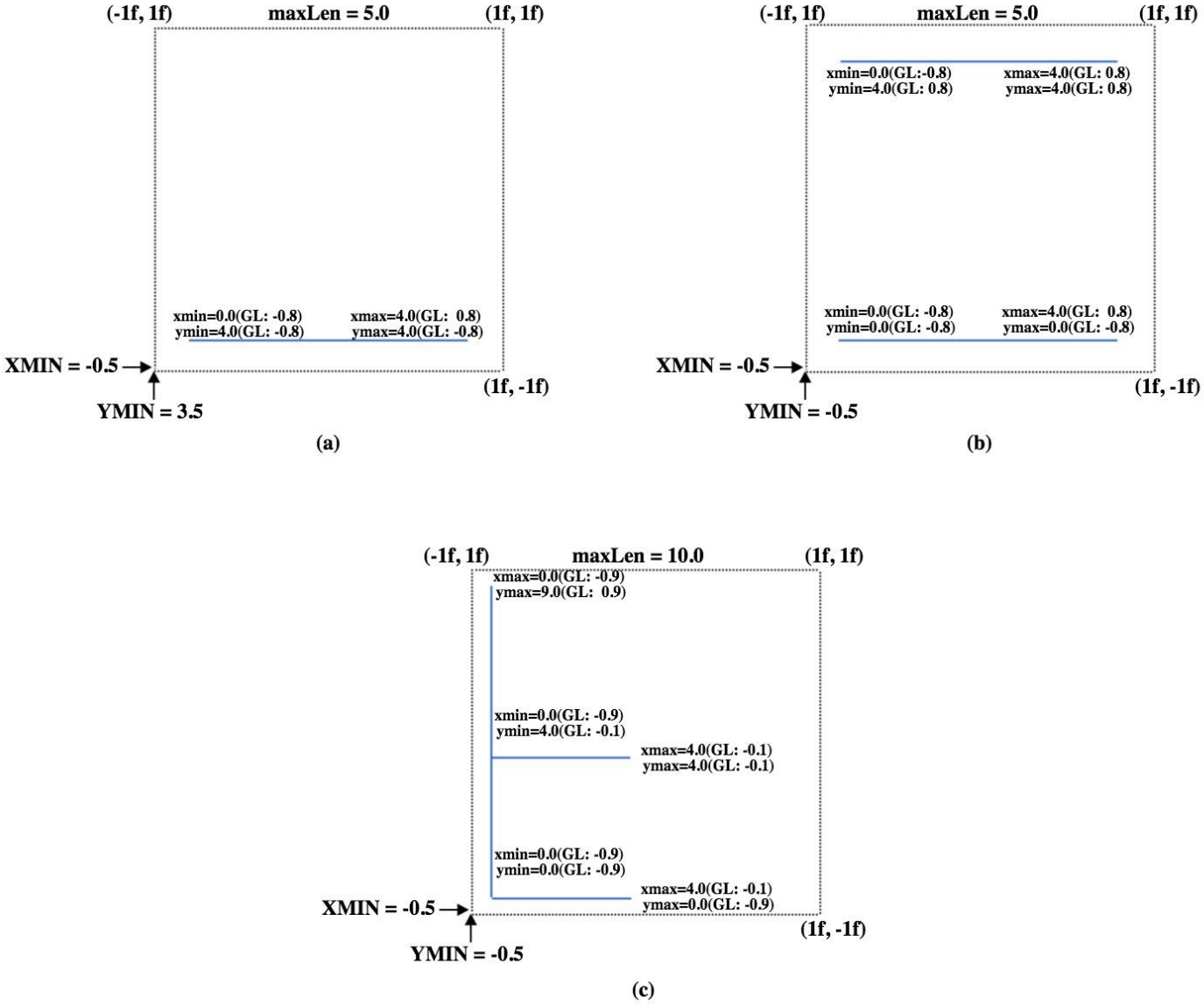


Figure 21: Map boundaries from real world to OpenGL

In (a), the first boundary is added with input data of $[0.0\text{mm}, 4.0\text{mm}, 4.0\text{mm}, 4.0\text{mm}]$ which is $[xmin, xmax, ymin, ymax]$ in the real-world coordinate. Since TLM boundaries must be at half-way between two shunt nodes, *XMIN* should be $\Delta l/2$ left to $xmin$ and *YMIN* should be $\Delta l/2$ beneath $ymin$. This is why in Listing 11 *XMIN* and *YMIN* are compared with $xmin - \Delta l/2$ and $ymin - \Delta l/2$. With the same reason, Δl is added to *maxLen* to prevent the OpenGL coordinates from getting out of $[-1, 1]$.

In this case, the first boundary updates *XMIN* to -0.5, *YMIN* to 3.5, and *maxLen* to 5. The mapping equations in below are used to compute the OpenGL coordinated for each vertex.

$$xmin_{gl} = -1f + (xmin - XMIN) * \frac{2}{maxLen}$$

$$ymin_{gl} = -1f + (ymin - YMIN) * \frac{2}{maxLen}$$

$$wmin = 1f + (ymin - YMIN) * \frac{1}{maxLen}$$

$$xmax_{gl} = -1f + (xmax - XMIN) * \frac{2}{maxLen}$$

$$ymax_{gl} = -1f + (ymax - YMIN) * \frac{2}{maxLen}$$

$$wmax = 1f + (ymax - YMIN) * \frac{1}{maxLen}$$

As the boundary is $[0.0mm, 4.0mm, 4.0mm, 4.0mm]$ and OpenGL assembles the line primitive for the boundary, two vertices with OpenGL coordinates for this boundary are $[-0.8, -0.8, 0.0, 1.1]$ and $[0.8, -0.8, 0.0, 1.1]$, respectively.

In (b), the second input boundary is $[0.0mm, 4.0mm, 0.0mm, 0.0mm]$. Following the same process in Table 15, the latest *YMIN* is -0.5. Since *YMIN* is different, the vertices for the first boundary are updated to $[-0.8, 0.8, 0.0, 1.9]$ and $[0.8, 0.8, 0.0, 1.9]$. OpenGL also has two new vertices, $[-0.8, -0.8, 0.0, 1.1]$ and $[0.8, -0.8, 0.0, 1.1]$ for the second boundary, to assemble the line primitive.

In (c), the third input boundary is $[0.0mm, 0.0mm, 0.0mm, 9.0mm]$. In this case, *maxLen* is changed into 10. With the new *maxLen*, the vertices for the first and second boundaries are updated to $[-0.9, -0.1, 0.0, 1.45]$, $[-0.1, -0.1, 0.0, 1.45]$, $[-0.9, -0.9, 0.0, 1.05]$ and $[-0.1, -0.9, 0.0, 1.05]$, respectively; new vertices $[-0.9, -0.9, 0.0, 1.05]$, $[-0.9, 0.9, 0.0, 1.95]$ for the third boundary are added into the vertex array as well.

The example in Figure 21 explains the mapping process from real-world coordinates to OpenGL coordinates with *XMIN*, *YMIN* and *maxLen*. Whenever Δl needed to be changed,

the value of *XMIN*, *YMIN*, *maxLen* and vertex data must be updated accordingly to keep consistent with the new Δl.

As OpenGL can only read data stored in the native memory and the vertex data is stored in Java memory, the data in Java memory must be copied to the native memory after the mapping process. A native *ByteBuffer* object is created to receive data from the *put* method which retrieves vertex data from Java memory.

4.5 Assemble Primitives

In order to assemble the primitives with the defined vertex and fragment shaders in the *ComputationGL* class, OpenGL enables the computation box program first. When the program links the vertex shader, the uniform and attribute in that shader are assigned with unique location ids. These ids can be retrieved after enabling the program and tell OpenGL the data in the *ByteBuffer* object should be sent to these locations. Listing 12 shows the implementation for finding the location and sending the data in the *ComputationGL* class.

```
private static final String A_POSITION = "a_Position";
private static final String U_MATRIX = "u_Matrix";
private int POSITION_COMPONENT_COUNT = 4;

public void useComProgram() {
    glUseProgram(com_program);
    aPositionLocation = glGetAttribLocation(com_program, A_POSITION);
    uMatrixLocation = glGetUniformLocation(com_program, U_MATRIX);
}

public void drawComMesh() {
    if (comboxVertexBuffer != null) {
        glUniformMatrix4fv(uMatrixLocation, 1, false, TLMGLRenderer.pmMatrix, 0);
        comboxVertexBuffer.position(0);
        glVertexAttribPointer(aPositionLocation, POSITION_COMPONENT_COUNT, GL_FLOAT,
                              false, 0, comboxVertexBuffer);
        glEnableVertexAttribArray(aPositionLocation);
        glDrawArrays(GL_TRIANGLES, 0, combox_up);
    }
}
```

Listing 12: Find the vertex data and project matrices via attribute and uniform locations.

The program id for the computation boxes is *com_program* which is argument in the *glUseProgram* method to enable the computation box program. The uniform and attribute location ids are found by the *glGetUniformLocation* method and *glGetAttribLocation* method, respectively. In the *ComputationGL* class, *uMatrixLocation* and *aPositionLocation* are the two variables storing the location ids for the computation box program. With the location ids, OpenGL begins to send the data in the *ByteBuffer* object

or the matrix arrays to the vertex shader which is linked with the computation box program. In the *ComputationGL* class, OpenGL uses the *glUniformMatrix4fv* method to send the matrix data to the uniform location. Therefore, the *location* is *uMartixLocation* and *value* is *pmMatrix*. On the other hand, OpenGL sends the vertex data via the *glVertexAttribPointer* method where *location* is *aPositionLocation* and *ptr* is the *ByteBuffer* object.

After sending the vertices and matrices, OpenGL begins to assemble primitives by the *glDrawArrays* method. OpenGL reads the *num_element* vertices in the *ByteBuffer* object from index *begin_id* in order to assemble the primitives. The primitive type is indicated by the argument *type* which can be points, lines or triangles.

The *ComputationGL* class encapsulates these steps in the *useComBoxProgram* and *drawComBox* methods as shown in Table 16. Since the app has other programs for the TLM components, the *onDrawFrame* method in the *renderer* always executes the *useComBoxProgram* ahead of the *drawComBox* method to inform OpenGL to assemble the computation boxes for the specific program.

Chapter 5. Concurrent Threads Mechanism

5.1 TLM Process Thread

In this app, absolute coordinates of TLM components are mapped to mesh indices before executing the *iterate* method in the *TLMProcess*. The absolute coordinates use *XMIN*, *YMIN* and Δl to compute $[xmin_{mesh}, xmax_{mesh}, ymin_{mesh}, ymax_{mesh}]$ which are the relative coordinates to *XMIN* and *YMIN* as well. If the structure of TLM components works well and only Δl needs to be modified to get the better the simulation result, this mapping process can promise to get the latest mesh indices for TLM components instead of re-editing them. How to get the mesh indices is explained in below via an example shown in Figure 22. On the other hand, OpenGL needs to know the maximal mesh indices in the x and y directions in order to assemble the primitives for the TLM mesh. This app stores the maximal indices in two variables–*nx* and *ny*. When computing the mesh indices for a new TLM component, *nx* and *ny* are updated if the new mesh indices have the greater values in the x or y directions. After traversing the TLM components to compute the mesh indices, maximal *nx* and *ny* are passed to the *AnimationGL* object which invokes OpenGL to assemble the primitives for the mesh. Figure 22 (a) employs the default Δl which is 1mm and has two electric boundaries with their vertices at $[0.0\text{mm}, 4.0\text{mm}, 0.0\text{mm}, 0.0\text{mm}]$ and $[0.0\text{mm}, 0.0\text{mm}, 0.0\text{mm}, 4.0\text{mm}]$ respectively. As introduced in Chapter 3, when these two boundaries are added, *XMIN* and *YMIN* are respectively revised to be –0.5 and –0.5. Since the mesh index begins at 1 and the boundary is at half-way between two shunt nodes, the mapping equations in Table 5 are used to compute the mesh indices and find the maximal *nx*, *ny*.

Horizontal boundaries:	Vertical boundaries:
$xmin_{mesh} = \left\lceil \frac{xmin - XMIN}{\Delta l} \right\rceil + 1$	$xmin_{mesh} = \left\lceil \frac{xmin - XMIN}{\Delta l} \right\rceil$
$xmax_{mesh} = \left\lceil \frac{xmax - XMIN}{\Delta l} \right\rceil$	$xmax_{mesh} = \left\lceil \frac{xmax - XMIN}{\Delta l} \right\rceil$
$ymin_{mesh} = \left\lceil \frac{ymin - YMIN}{\Delta l} \right\rceil$	$ymin_{mesh} = \left\lceil \frac{ymin - YMIN}{\Delta l} \right\rceil + 1$
$ymax_{mesh} = \left\lceil \frac{ymax - YMIN}{\Delta l} \right\rceil$	$ymax_{mesh} = \left\lceil \frac{ymax - YMIN}{\Delta l} \right\rceil$
$nx = Math.max(nx, xmax_{mesh} - xmin_{mesh} + 3)$ $ny = Math.max(ny, ymax_{mesh} - ymin_{mesh} + 3)$	

Table 5: Mapping equations to get mesh indices.

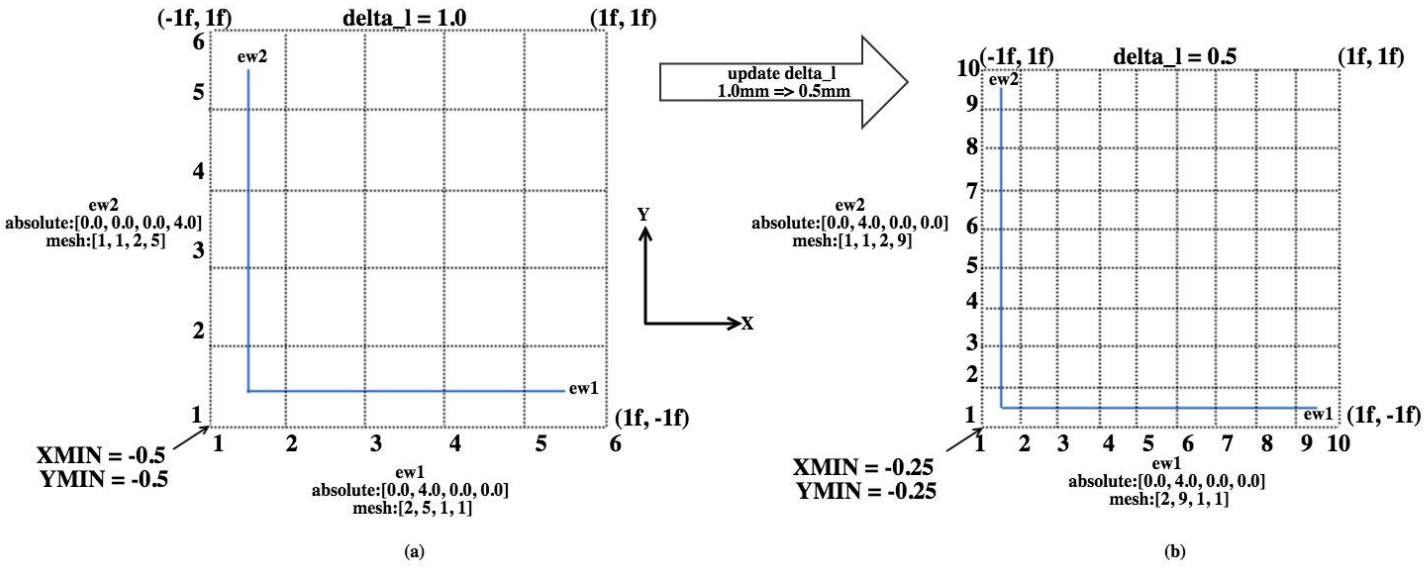


Figure 22: Map the absolution positions of boundaries to mesh indices.

In Figure 22 (a), when the first horizontal electric boundary–*ew1* goes through the mapping equations shown in Table 5, the mesh indices for *ew1* is [2,5,1,1]; *nx* is 6 and *ny* is 3. The second vertical electric boundary—*ew2* has mesh indices as [1,1,2,5] which updates *ny* to 6. Clicking on the "MESH" button passes $nx = 6$ and $ny = 6$ to the *AnimationGL* object which follows the similar process introduced in Chapter 4 to assemble the primitives for the mesh. The dash lines in the figure are the mesh added among the TLM components. If

the simulation result is not satisfying, the "UNMESH" button cleans the mesh vertex data in the *AnimationGL* object, then Δl become editable. As shown in Figure 22 (b), Δl is modified to be 0.5mm, and the values of *XMIN* and *YMIN* are updated to –0.25 and –0.25. Clicking the "MESH" button allows to go through the mapping equations in Table 5 again. The latest mesh indices for *ew1* and *ew2* are respectively $[2, 9, 1, 1]$ and $[1, 1, 2, 9]$. New $nx = 10$ and $ny = 10$ are also passed the *AnimationGL* object to assemble the latest mesh. Therefore, different mesh indices are applied to the same structure of the TLM components when Δl is changed, re-editing the TLM components is not necessary.

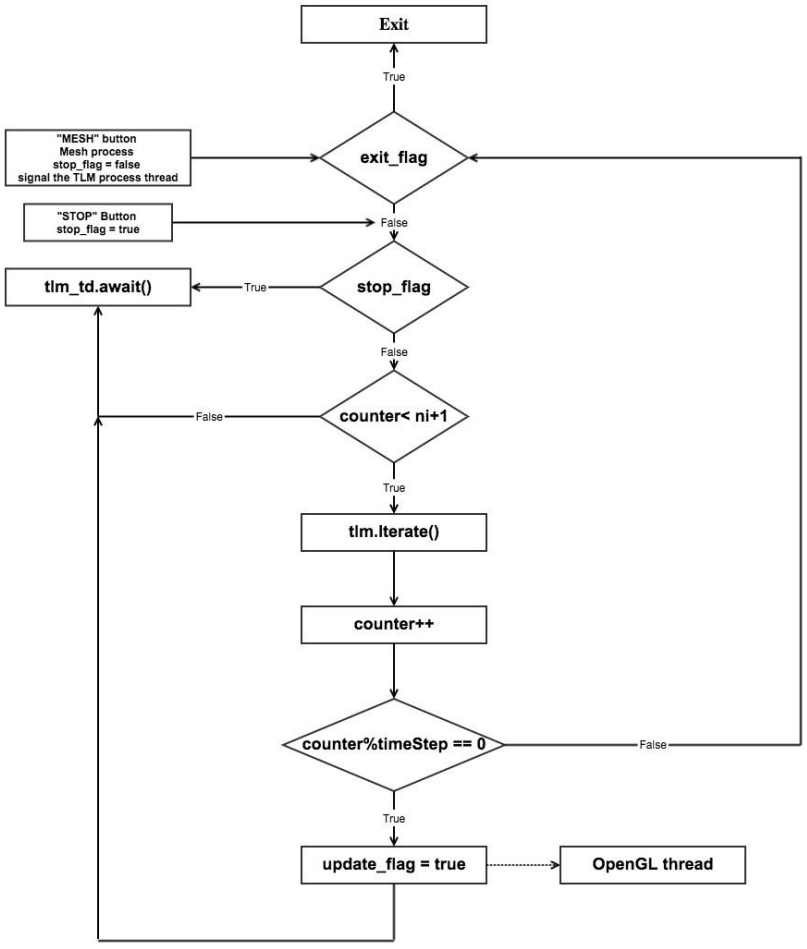


Figure 23: Flow chart for the TLMProcess thread.

Since the "MESH" button gets the TLM network ready, the app can execute the TLM method to simulate electromagnetic wave propagation. The TLM method is implemented in the *iterate* method of the *TLMProcess* class and a TLMProcess thread is created to

control the iteration steps as shown in Figure 23. In the *onCreate()* method of the *MainActivity* class, this thread is created and started immediately. Before running the *iterate* method, the TLMProcess thread checks two conditions first – *exit_flag* and *stop_flag* which are initialized to be "false" and "true", respectively. Since the thread should never exit if the app is running, *exit_flag* is always "false". Because there is no TLM network yet when the app is launched, the default value for *stop_flag* is "true" which means the TLMProcess thread is suspended immediately after it is brought up in the *MainActivity* class. Clicking the "MESH" button not only passes the *nx* and *ny* to the *AnimationGL* object, but also updates the *stop_flag* as "false" and signals the suspended TLMProcess thread. After the thread is active, it also needs to check whether the iteration progress reaches the preset iteration number limit. This is implemented by a counter added to the thread. When the value of this counter is smaller than or equal to the iteration limit, the thread executes the *iterate* method and the counter increases by 1. Then the thread goes back to the loop beginning and repeats the loop. On the opposite, the TLMProcess thread is suspended which means the thread has finished producing data. In the *iterate* method, the active thread executes one TLM iteration to produce the scattered voltage impulses. These impulses are used to calculate the output voltage at the output point with the equation of $V_y = \frac{2}{4+y_0}\left(\sum_{m=1}^{4} V_m^i + V_5^i y_0\right)$. The output voltage in each TLM iteration is added into arrays where the array index indicates how many iterations has been operated. After executing one TLM iteration, the "$counter\%timeStep == 0$" constrain is used by the TLM Process thread to determine whether to notify the OpenGL thread to update the vertices at every $timeStep$ iterations. Since the TLMProcess thread only records the scattered impulses in the current TLM iteration, when the "$counter\%timeStep == 0$" constrain is satisfied, the TLMProcess thread is suspended until the OpenGL thread finishes updating the vertices.

The "STOP" sets the *stop_flag* to "true" to suspend the TLMProcess thread. Since the TLMProcess thread is the data producing thread, this thread is kept active until it finishes producing data or the OpenGL thread needs to update the vertices. If the TLM network needs to be modified, the "STOP" button must be clicked first before doing any change in the TLM structure. Other buttons, such as "PAUSE", "RESET" or "++", cannot control

the TLMProcess thread. They only manipulate the simulation steps via controlling how much data should be read from the output voltage arrays. The simulation steps are implemented by other threads which are explained in the next sections.

5.2 OpenGL Thread

The thread for OpenGL is maintained by Android. Once a *renderer* object is instantiated, the OpenGL thread is created and keeps active until the *GLSurfaceView* instance gets destroyed. Therefore, the vertex data is the only access to control the primitives assembled by OpenGL.

The OpenGL thread keeps executing the *onDrawFrame* method when it's active. Figure 24 shows the *onDrawFrame* method assembling primitives for a TLM component such as a boundary, computation box, excitation probe or output probe. The *onDrawFrame* method execute three other methods – *bindVertexData*, *useProgram* and *drawComponent*. The *bindVertexData* method checks the *update_flag* constrain first. If the flag is "true", the *bindVertexData* method re-computes the OpenGL coordinates of the TLM components and re-sends the latest vertex data to the *ByteBuffer* object. Then the *useProgram* and *drawComponent* methods are executed to assemble primitives for the TLM components with modified vertex data. If the flag is "false", re-computing the OpenGL coordinates and updating the *ByteBuffer* object in the *bindVertexData* method are skipped and OpenGL directly runs the *useProgram* and *drawComponent* methods to assemble the primitives with the vertex data already in the byte buffer. In fact, any change in the TLM components, such as adding, deleting, or moving components, should set the flag as "true" to invoke the updating process. Figure 24 also indicates that the *update_flag* is set as "false" after updating the vertex data. This step makes sure OpenGL only updates the vertex data for one time if the TLM components have any change.

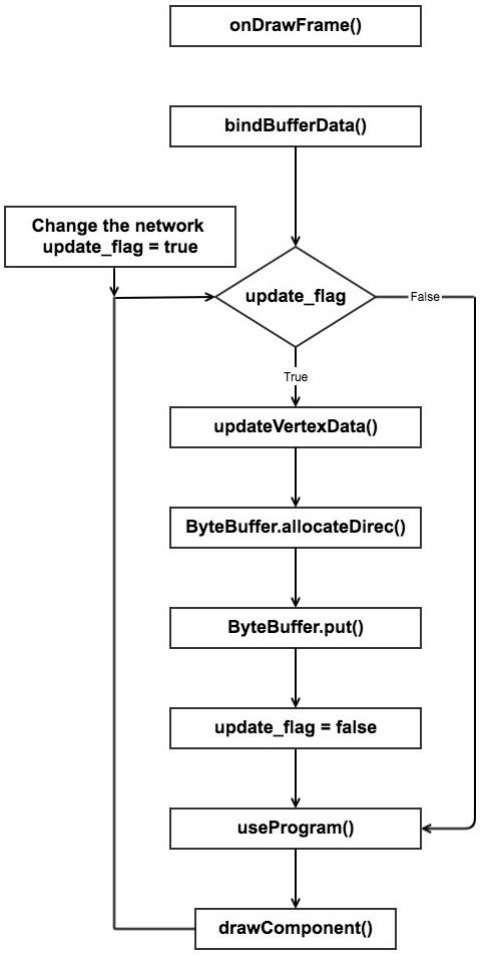
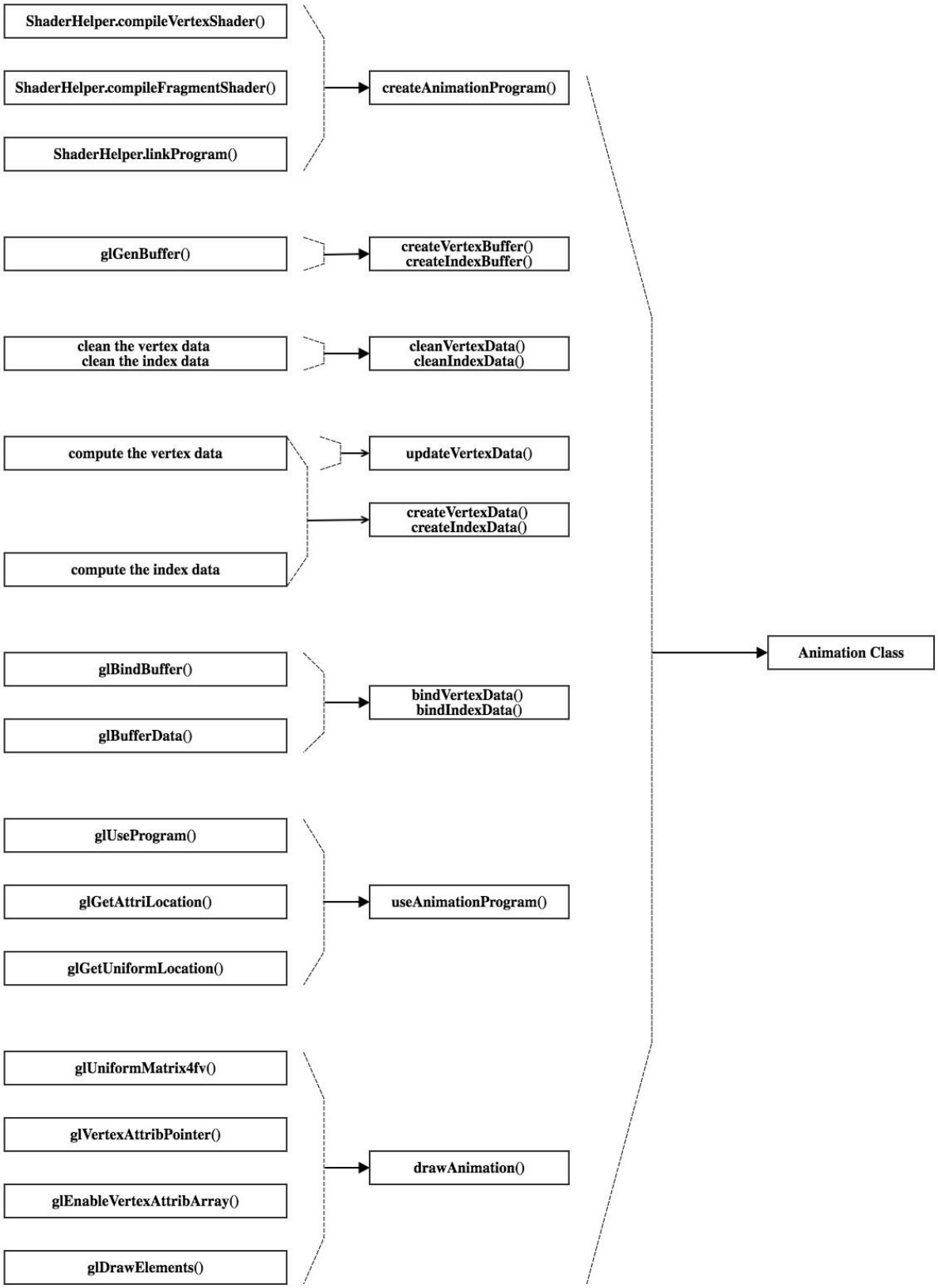
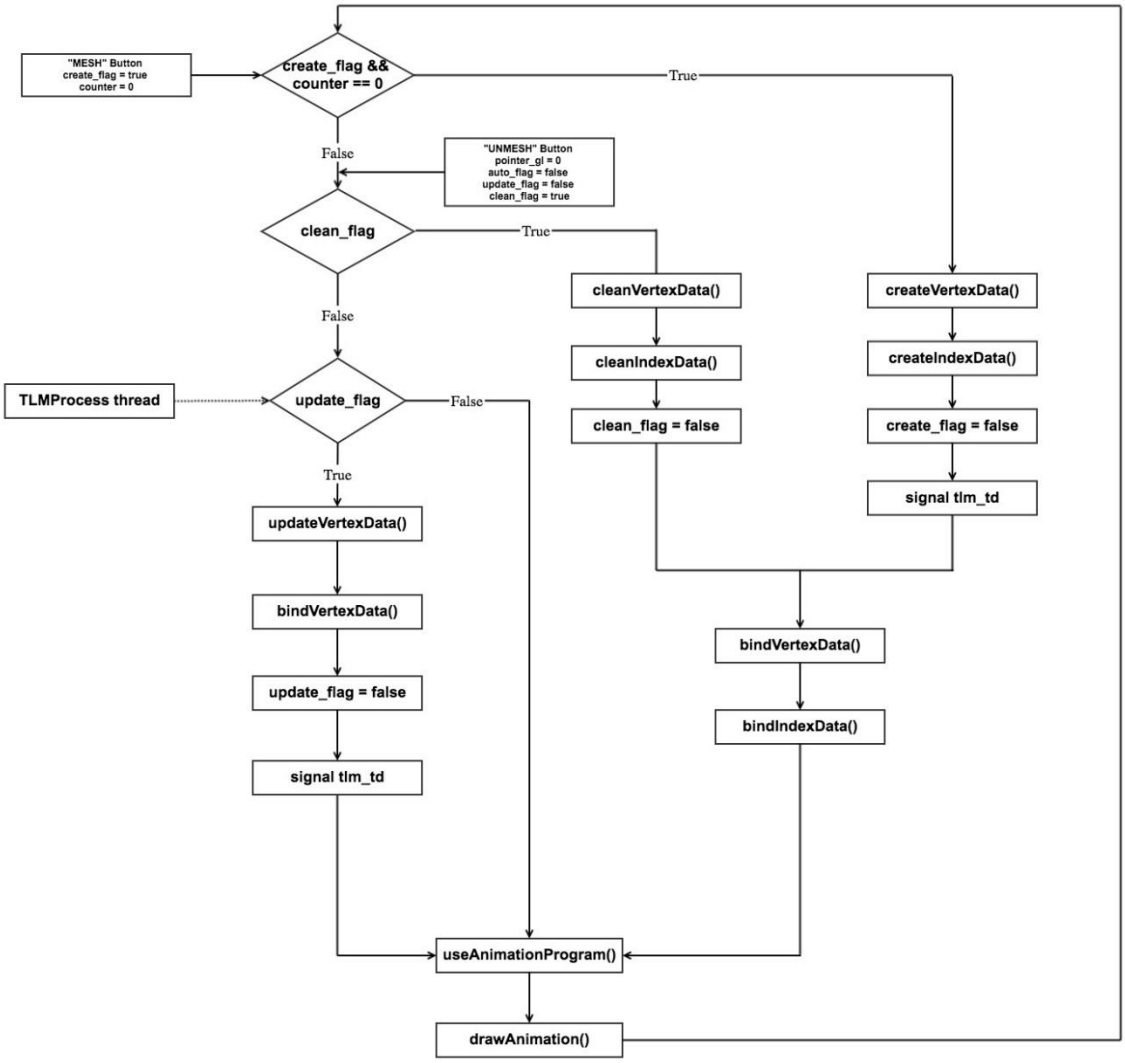


Figure 24: The flow chart to assemble the TLM components in the *onDrawFrame* method.

Once all the TLM components are edited, a 2D shunt-connected mesh may be created by clicking the "MESH" button. As soon as a mesh is created, a signal is sent to the TLMProcess thread. This mesh size is determined by the *nx* and *ny* introduced in Section 5.1. Assembling the primitives for a TLM mesh is implemented by the *AnimationGL* class. In this class, OpenGL employs both of Vertex Buffer Object (VBO) and Index Buffer Object (IBO) because one vertex in the mesh may be reused up to four times. Sending the vertex data to shaders via an IBO instead of repeating the vertex in the *ByteBuffer* object saves the native memory and improves the OpenGL performance. Figure 25 shows the *AnimationGL* class and the flow chart for the OpenGL thread assembling the mesh in the *onDrawFrame* method.



(a) The *AnimationGL* class.



(b) Flow chart of assembling the mesh in the *onDrawFrame()* method.
Figure 25: The *AnimationGL* class and the flow chart for assembling the mesh.

In the *onDrawFrame* method shown in Figure 25(b), the *AnimationGL* class also programs the data in the VBO and IBO to control the primitives assembled by OpenGL since the *useAnimationProgram* and *drawAniamtion* methods are always executed. When the "MESH" button is clicked, the *create_flag* is set to "true" to create vertex and index data which is determined by *nx* and *ny*. Similar to the *ByteBuffer* object, the VBO and IBO receive data from the *glBindBuffer* and *glBufferData* methods which copy the vertex and index data from Java memory to the native memory. The "UNMESH" button is used to remove the mesh before modifying Δl. The button sets the *clean_flag* as "true" to clean the vertex and index data in Java memory. The *glBindBuffer* and *glBufferData* methods are executed as well to clean the VBO and IBO. After the data in the VBO and IBO is created

or cleaned, the OpenGL thread executes the *useAnimationProgram* and *drawAniamtion* methods to assemble the primitives for the mesh. Both the *create_flag* and *clean_flag* are set as "false" before sending data to the VBO and IBO in order to ensure the creation and cleaning processes are only executed once. The complete codes of creating the vertex and index data, sending data to VBO and IBO, and assembling primitives are in the Appendix.

As each vertex of the mesh represents a shunt node in the TLM network, the value of z in the OpenGL coordinate for a vertex can be updated via calculating and normalizing the shunt node output voltage in the current TLM iteration. Since the TLMProcess thread is suspended before the TLM network is complete, the initial values of z for the mesh vertices are 0. In the creating process triggered by the "MESH" button, creating the vertex and index data indicates the TLM network is accomplished. Therefore, the TLMProcess thread is signaled to execute the TLM iteration after the *createVertexData* and *createIndexData* methods. After every $timeStep$ TLM iterations, the TLMProcess thread notifies the OpenGL thread to update z via setting the *update_flag* as "true". Similar as the updating process for constructing the TLM components, the updating process for the mesh is executed ahead of the *useAnimationPogram* and *drawAniamtion* methods if the *update_flag* is "true". This updating process only requires to revise the vertex data and re-send to the VBO since the mesh indices are same. Setting the *update_flag* as "false" after the *updateVertexData()* method also ensure OpenGL to execute the updating process once. Before running the *useAnimationProgram* method, the OpenGL thread signals the TLMProcess to continue the TLM iterations.

5.3 DFT Threads

The DFT process in this app has two threads which employ *canvas* objects to draw the view. One thread is to handle the time-domain data sequence and another one is for computing the required DFT result. Since these two threads employ a similar concurrent thread mechanisms, this section only explains the time-domain thread. But Figure 26 shows the complete flow chart for both the time-domain and frequency-domain threads.

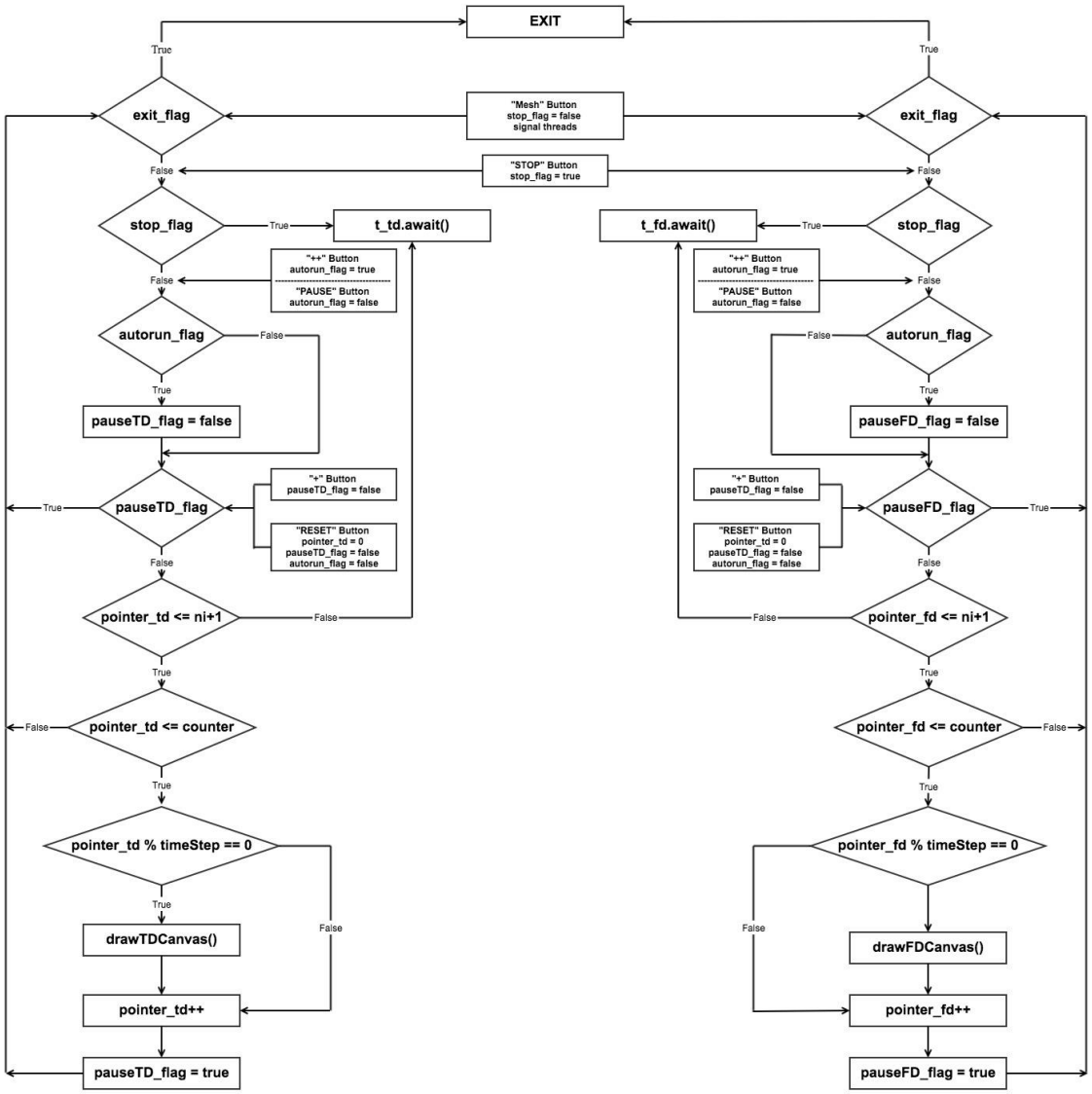


Figure 26: The complete time-domain and frequency-domain threads.

When the "FT" *fragment* instance is brought up in the *FragmentPagerAdapter* class, the time-domain thread is initialized and started immediately as well. Since the *exit_flag* and *stop_flag* are global variables which are also applied to the threads for the DFT process, the time-domain thread is suspended immediately after it is brought up because the default values for *exit_flag* and *stop_flag* are "false" and "true", respectively. As introduced in Section 5.2, the TLMProcess thread is signaled after adding the mesh. Actually, the time-domain thread is signaled as well. Though the time-domain thread is active, this thread doesn't draw on the *canvas* object since the default value for the *pauseTD_flag* is "true".

The thread fails at the *pauseTD_flag* condition and goes back to the beginning of the loop. The "+" button is used to set the *pauseTD_flag* as "false" which allows the time-domain thread to check the next constrains. The ensuing constrains are the pointer in the time-domain thread should neither exceed the iteration number nor the counter in the TLMProcess thread. This pointer indicates the index of the array which stores the output voltage in each TLM iteration, and the time-domain thread draws the data sequence from *array[0]* to *array[pointer]* on the *canvas* object. If the pointer exceeds the iteration number, it means all the output voltages stored in the array have been drawn on the *canvas* object. If the pointer exceeds the counter, it means the TLMProcess thread hasn't produced enough data. The time-domain thread needs to wait. To improve the efficiency of the time-domain thread, the *timeStep* applied in the TLMProcess thread is also used in the time-domain thread to execute the drawing process only when the pointer meets the "constrain – $pointer\_td\%timeStep == 0$". No matter whether the drawing process is executed, the pointer adds 1 and the *pauseTD_flag* is reset as "true" again which means the "+" button only goes through the time-domain loop once. In this case, the "+" button needs to be clicked *timeStep* times, then the time-domain thread can execute the drawing process. To avoid this problem, "$pointer\_td = (pointer\_td/timeStep + 1) \times timeStep$" is also executed when clicking the "+" button to ensure the drawing process can be executed immediately. The time-domain thread is supposed to go through the loop continuously when there are data available in the array, the "++" button can be clicked. This button sets the *autoRun_flag* as "true" which always lets the *pauseTD_flag* be "false". In this circumstance, the pointer in the time-domain can continuously increment by 1 and the thread can execute the drawing process when "$pointer\_td\%timeStep == 0$".

If the pointer reaches to the iteration limit, the time-domain thread is suspended. This is different from the OpenGL thread which is still active in such a circumstance. The OpenGL thread just skips the updating the vertex and always assembles primitives. Suspending the time-domain thread can also be implemented by the "STOP" button which sets *stop_flag* as "true".

In this app, it also has a "PAUSE" button which sets *autoRun_flag* as "false". In this case, the OpenGL thread, the time-domain and frequency-domain threads can finish the updating

and drawing process in the current loop and set *update_flag=false*, *pauseTD_flag=true*, *pauseFD_flag=true*. In the ensuing loops, these threads cannot operate the updating and drawing processes because they fail to pass the constrains, but they are still active. In order to continue at the point where they pause at the last time, the "+" or "++" buttons can be clicked. The "RESET" button works similar as the "PAUSE" button. The only difference is that the "RESET" buttons resets the pointers in the OpenGL thread and DFT threads to 0 which means the threads read the array from index 0, not the index where they pause.

5.4 Constrains Controlled by Buttons

The above three sections introduced the threads for showing the result of the simulation and DFT processes, this section uses a complete table to depict the buttons and their associated actions.

default value	create_flag (OpenGL thread for the mesh) = false clean_flag (OpenGL thread for the mesh) = false update_flag (OpenGL thread for the mesh) = false pauseTD_flag (time-domain thread) = true pauseFD_flag (frequency-domain thread) = true exit_flag (global) = false stop_flag (global) = true autoRun_flag (global) = false
"MESH"	create_flag = true stop_flag = false signal the TLMProcess thread signal the DFT threads
"UNMESH"	stop_flag = true autoRun_flag = false update_flag = false clean_flag = true
"+"	pauseTD_flag = false pauseFD_flag = false $pointer\_td = (pointer\_td/timeStep + 1) \times timeStep$ $pointer\_fd = (pointer\_fd/timeStep + 1) \times timeStep$
"++"	autoRun_flag = true
"STOP"	stop_flag = true autoRun_flag = false

	update_flag = false
“PAUSE”	autoRun_flag = false
“RESET”	autoRun_flag = false pointer_td=0, pointer_fd=0

Table 6: Constrains controlled by buttons.

Chapter 6. Validation and Conclusion

6.1 Validation

To validate the correctness and accuracy of the TLM app, the app is used to compute the cut-off frequencies for a number of rectangular waveguides and the results are compared with theoretical results and from MEFiSTo, another TLM based modeling tool.

Table 7 shows the properties for TE and TM equations for a rectangular waveguide with cross-sectional dimensions equal to $a \times b$ as shown Figure 27(a). In the equations, m and n are positive integers for various TE and TM modes. However, the lowest order for the TE mode should be $m = 1, n = 0$ if $a > b$ or $m = 0, n = 1$ if $a < b$. And the TM mode begins from $m = 1, n = 1$ as either m or n equal to zero would yield a null solution. Figure 27 (b) depicts a rectangular waveguide cross-section modeled by the TLM app.

TE Mode	TM Mode
$\widetilde{E_x} = \frac{j\omega\mu}{k_c^2}\left(\frac{n\pi}{b}\right) H_0 \cos\left(\frac{m\pi x}{a}\right) sin\left(\frac{n\pi y}{b}\right) e^{-j\beta z}$	$\widetilde{E_x} = \frac{-j\beta}{k_c^2}\left(\frac{m\pi}{a}\right) E_0 \cos\left(\frac{m\pi x}{a}\right) sin\left(\frac{n\pi y}{b}\right) e^{-j\beta z}$
$\widetilde{E_y} = \frac{-j\omega\mu}{k_c^2}\left(\frac{m\pi}{a}\right) H_0 \sin\left(\frac{m\pi x}{a}\right) cos\left(\frac{n\pi y}{b}\right) e^{-j\beta z}$	$\widetilde{E_y} = \frac{-j\beta}{k_c^2}\left(\frac{n\pi}{b}\right) E_0 \sin\left(\frac{m\pi x}{a}\right) cos\left(\frac{n\pi y}{b}\right) e^{-j\beta z}$
$\widetilde{E_z} = 0$	$\widetilde{E_z} = E_0 \sin\left(\frac{m\pi x}{a}\right) sin\left(\frac{n\pi y}{b}\right) e^{-j\beta z}$
$\widetilde{H_x} = -\widetilde{E_y}/Z_{TE}$	$\widetilde{H_x} = -\widetilde{E_y}/Z_{TM}$
$\widetilde{H_y} = \widetilde{E_x}/Z_{TE}$	$\widetilde{H_y} = \widetilde{E_x}/Z_{TM}$
$\widetilde{H_z} = H_0 \cos\left(\frac{m\pi x}{a}\right) cos\left(\frac{n\pi y}{b}\right) e^{-j\beta z}$	$\widetilde{H_z} = 0$
$f_{mn} = \frac{u_{p0}}{2}\sqrt{\left(\frac{m}{a}\right)^2 + \left(\frac{n}{b}\right)^2}$ (cut-off frequency)	

Table 7: Wave properties for TE and TM modes of a rectangular waveguide.

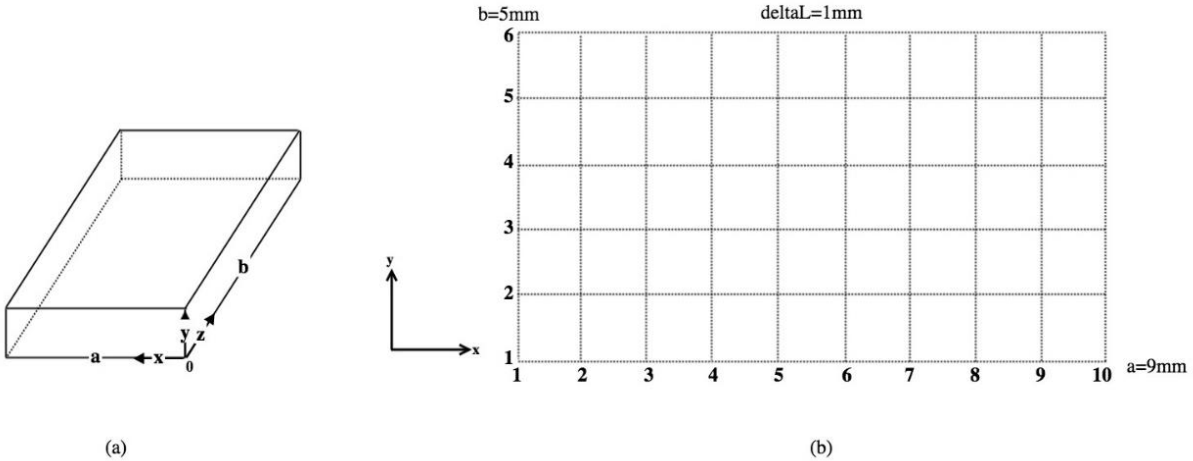
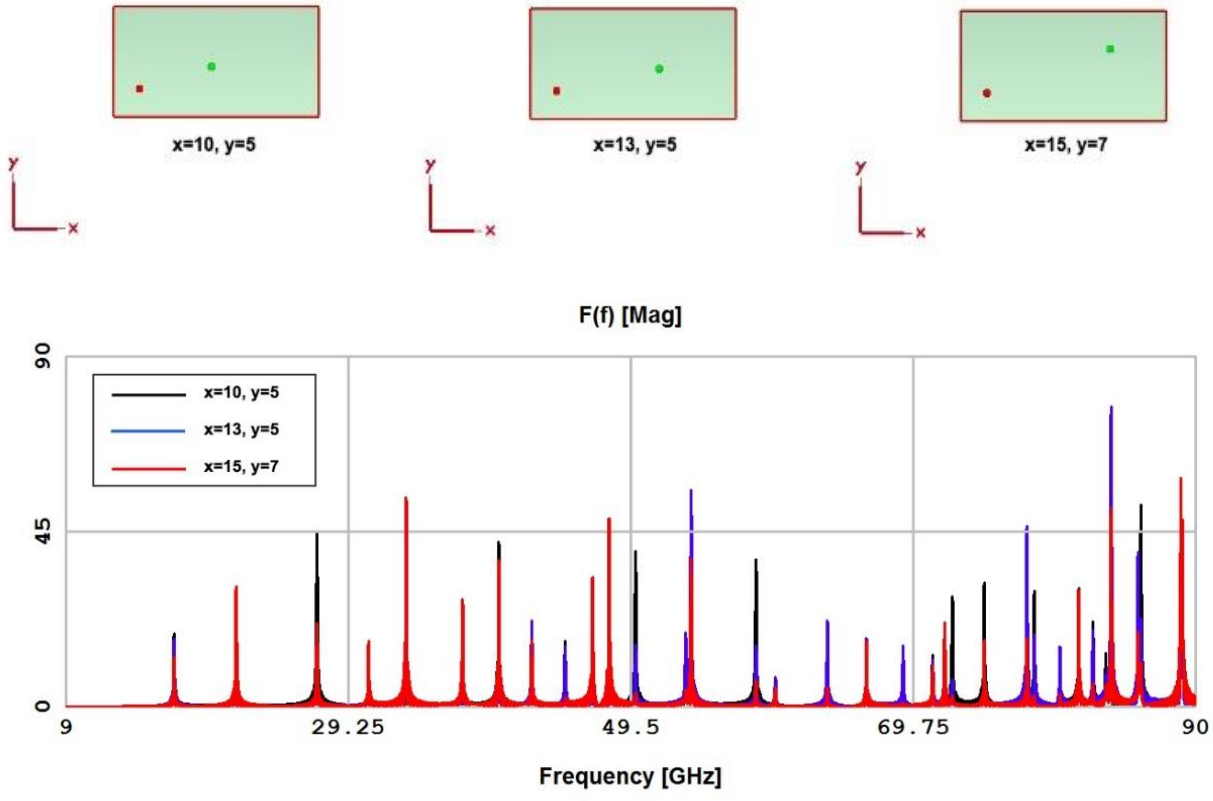


Figure 27: A cross-sectional plane and a rectangular waveguide applied in the app.
(a)A cross-sectional plane. The x-y plane is the rectangular waveguide applied to the app.
(b)An example of a rectangular waveguide where a=9mm, b=5mm and $\Delta l = 1mm$.

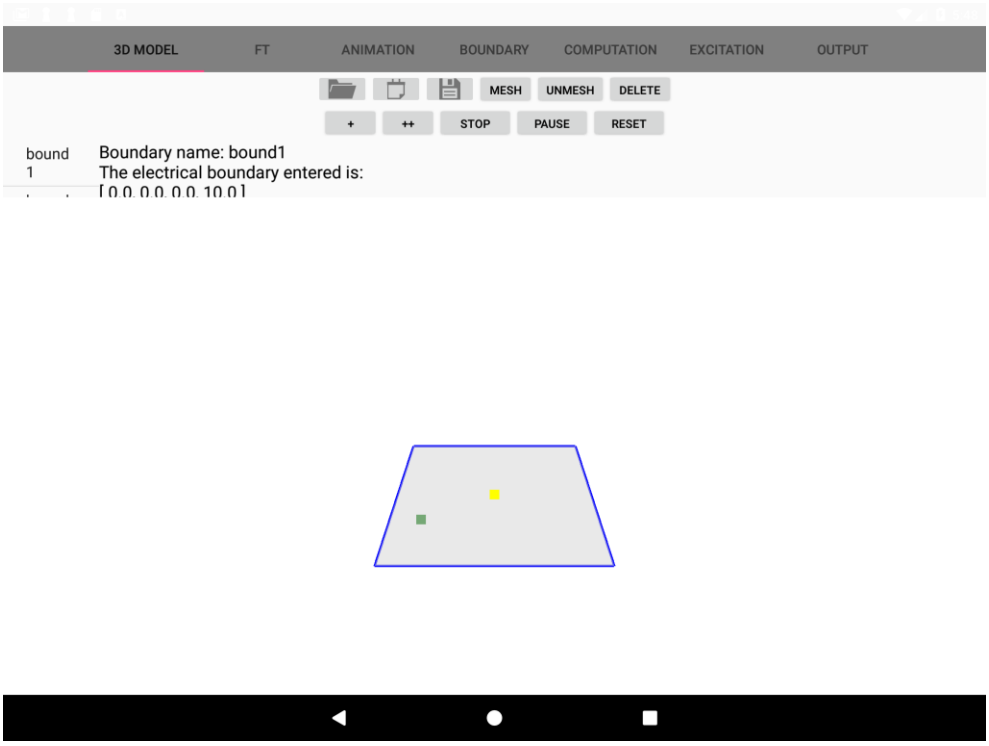
Since the $\widetilde{E_z}$ in the TM mode and $\widetilde{H_z}$ in the TE mode are periodical, the magnitudes of $\widetilde{E_z}$ and $\widetilde{H_z}$ of different modes vary with positions in the xy plane. For example, if $\widetilde{E_z}$ with $m = 1, n = 1$ has $0.8 \times peak\ value$ at the point of $a = 1$ mm, $b = 1$ mm, and $\widetilde{E_z}$ with $m = 2, n = 1$ has the value of $0.5 \times peak\ value$ at this point, the magnitude response at order of $m = 2, n = 1$ is very close the magnitude response at order of $m = 1, n = 1$ after DFT. In this circumstance, both of the cut-off frequencies for these two orders are invalid as the valid cut-off frequency should have the significantly greater magnitude than nearby frequency points. If the point of $a = 1$ mm, $b = 1$ mm moves a little bit in the rectangular waveguide, $\widetilde{E_z}$ with $m = 1, n = 1$ might be the peak value and $\widetilde{E_z}$ with $m = 2, n = 1$ may be zero. In this case, the cut-off frequency for $m = 1, n = 1$ is valid since the magnitude at this frequency point has the significantly greater magnitude than the nearby frequencies. It is also the reason why different output probe positions are used in MEFiSTo and in the TLM app.

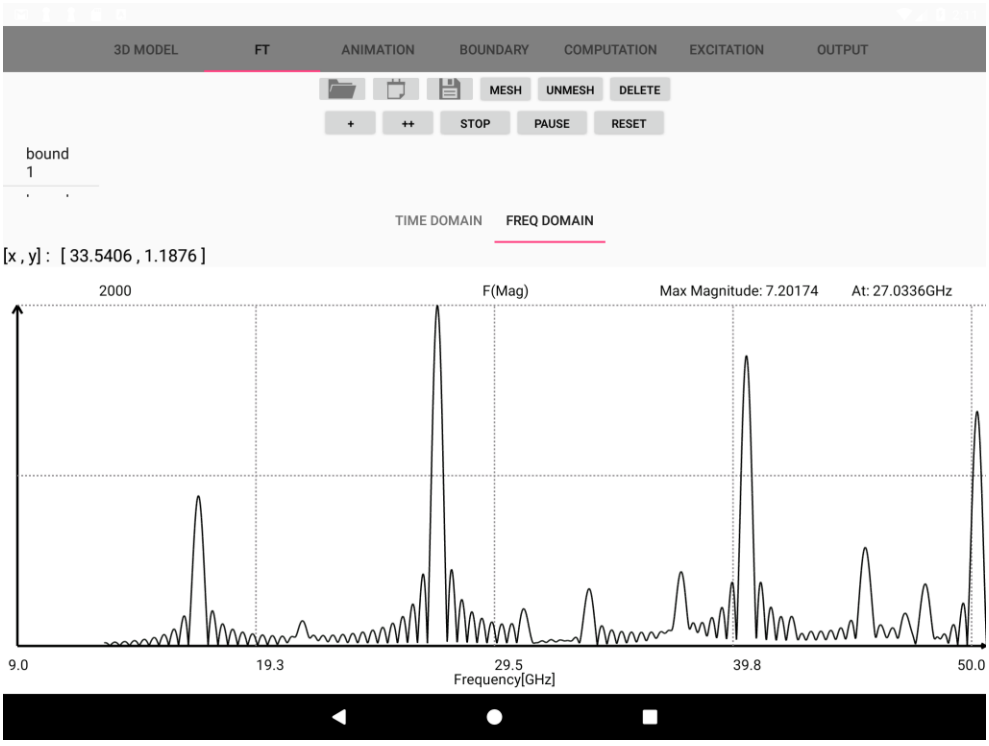
6.1.1 Test Case 1 – TM Mode in $20\text{mm} \times 10\text{mm}$ Rectangular Waveguide

The first test case is the TM mode in $a = 20\text{mm}, b = 10\text{mm}$ rectangular waveguide. To get the valid cut-off frequencies, three output probes are used in both MEFiSTo and in the TLM app. Figure 28(a) shows the TLM structure with different output probe positions and the corresponding magnitude responses of DFT in MEFiSTo with $\Delta l = 1\text{mm}$. As expected, the magnitudes of cut-off frequencies vary with output point positions. On the other hand, when the value of Δl is decreased to increase the spatial resolution of the TLM mesh, the computed result become more accurate. Therefore, the app executes the TLM method with two resolutions. Figure 28(b) is the magnitude response of DFT with $\Delta l = 0.5\text{mm}$ resolution in the app and (c) 0.25mm resolution, respectively. The cut-off frequencies obtained with theoretical equations are shown in Table 7; the results from MEFiSTo and the TLM app are shown in Figure 28. The difference between the theoretical and modeling results are shown in Figure 29.

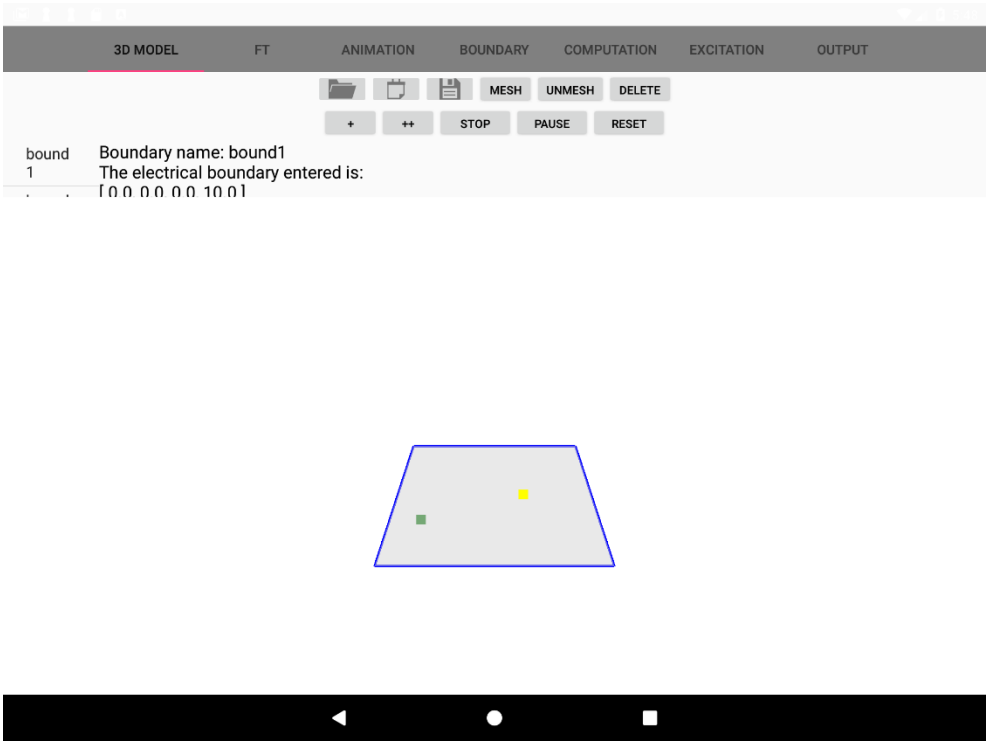


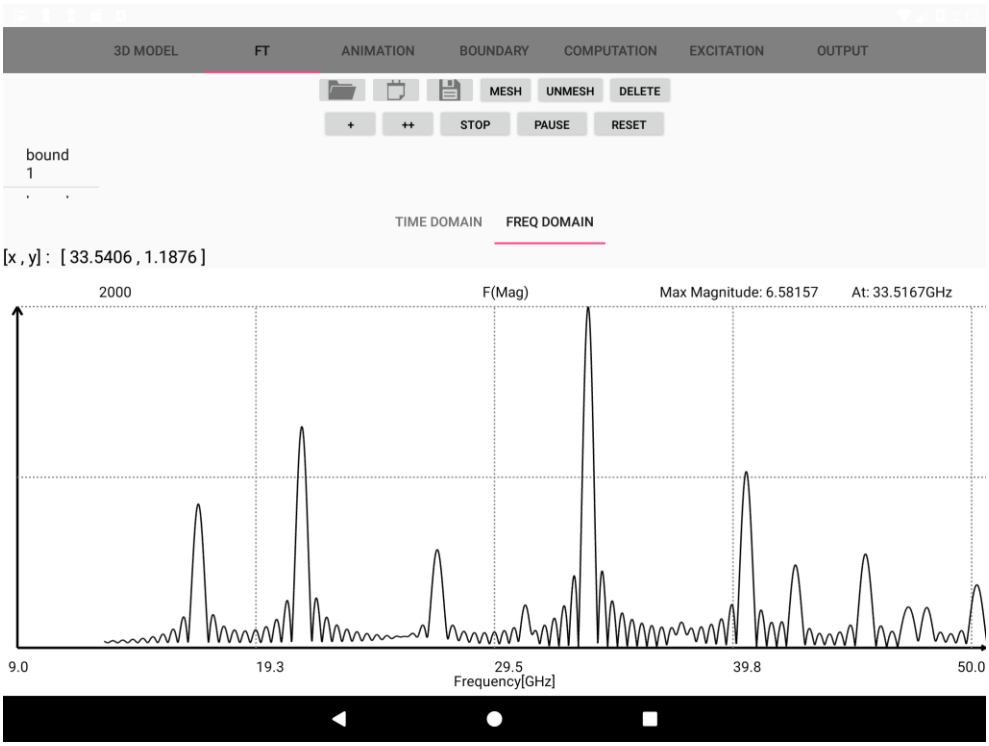
(a) Magnitude response of DFT at three output points in MEFiSTo with $\Delta l = 1mm$.



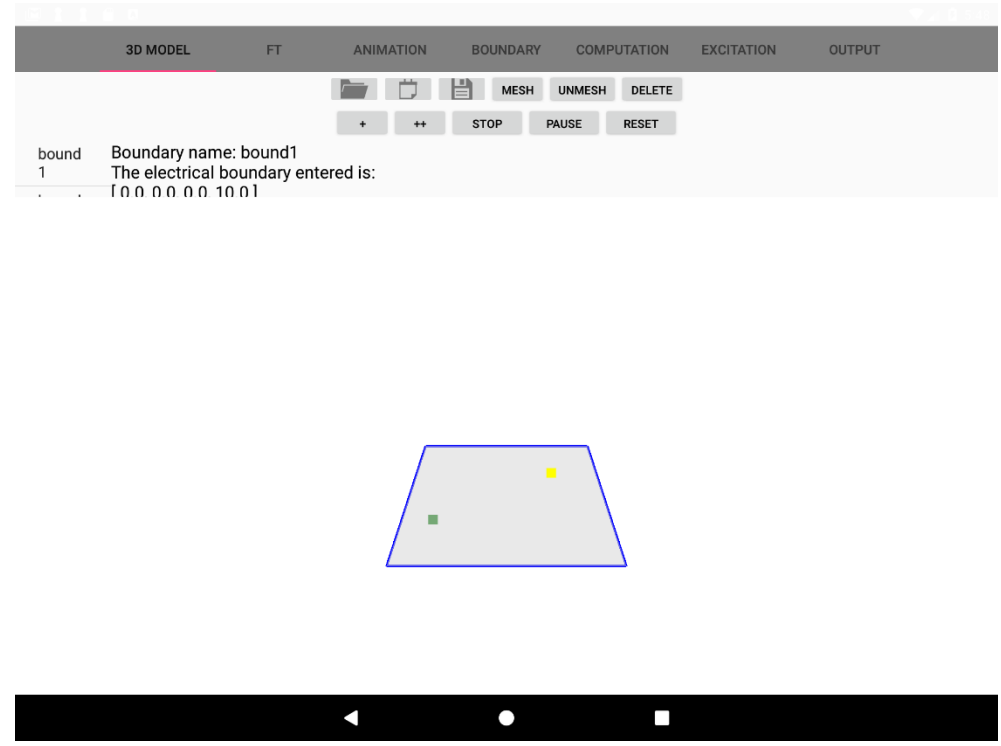


At Probe x=10, y=5



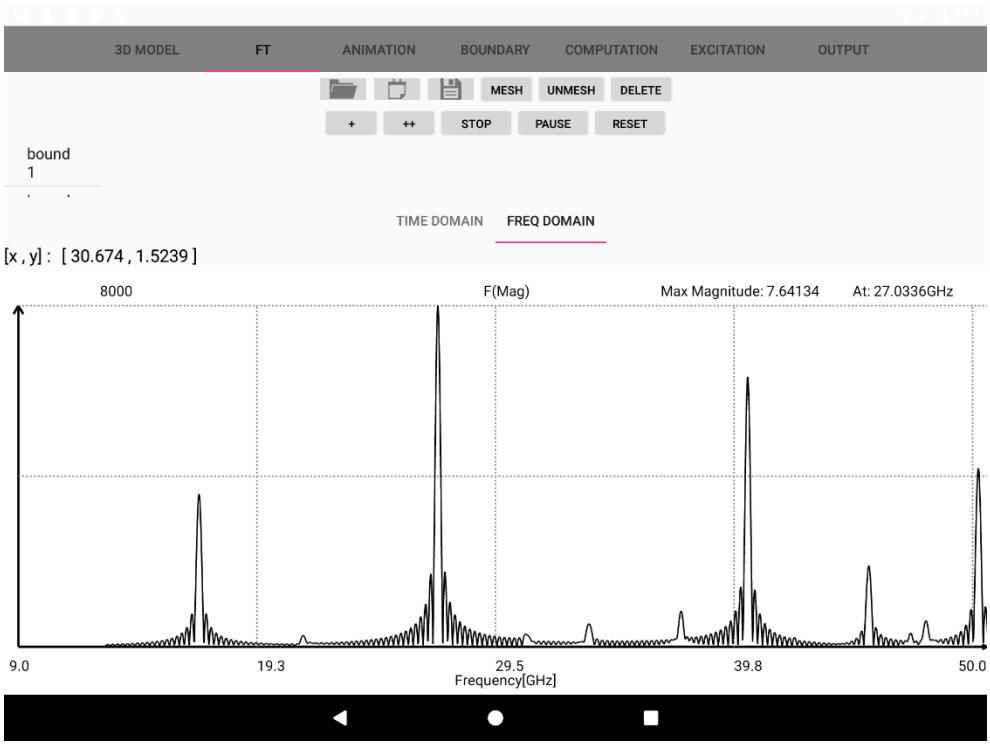


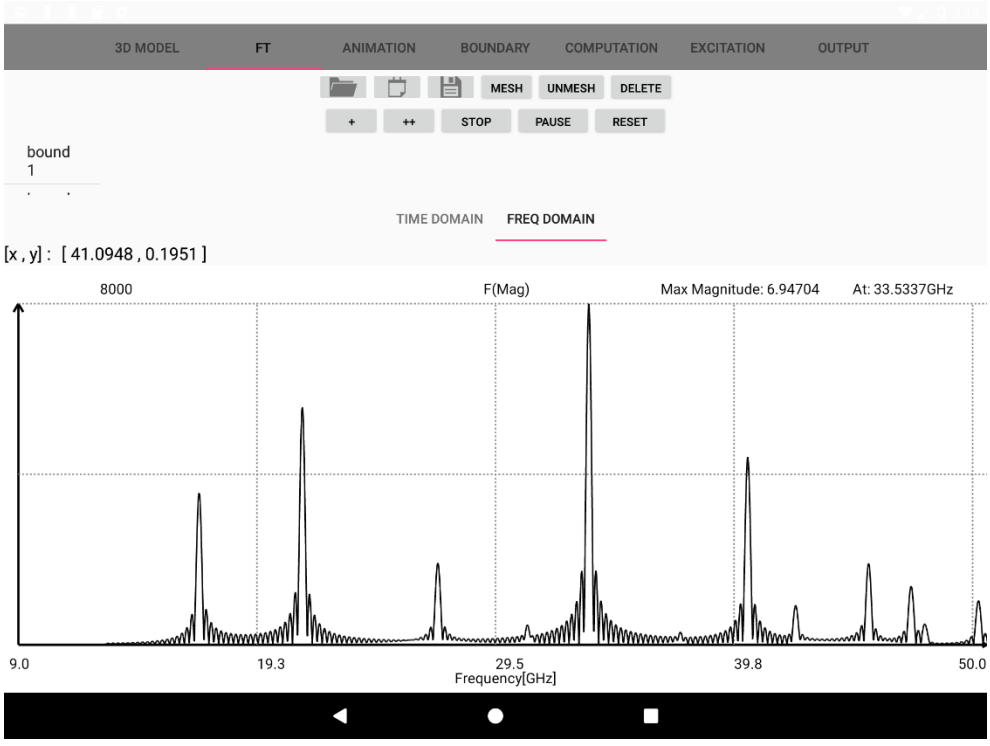
At Probe x=13, y=5

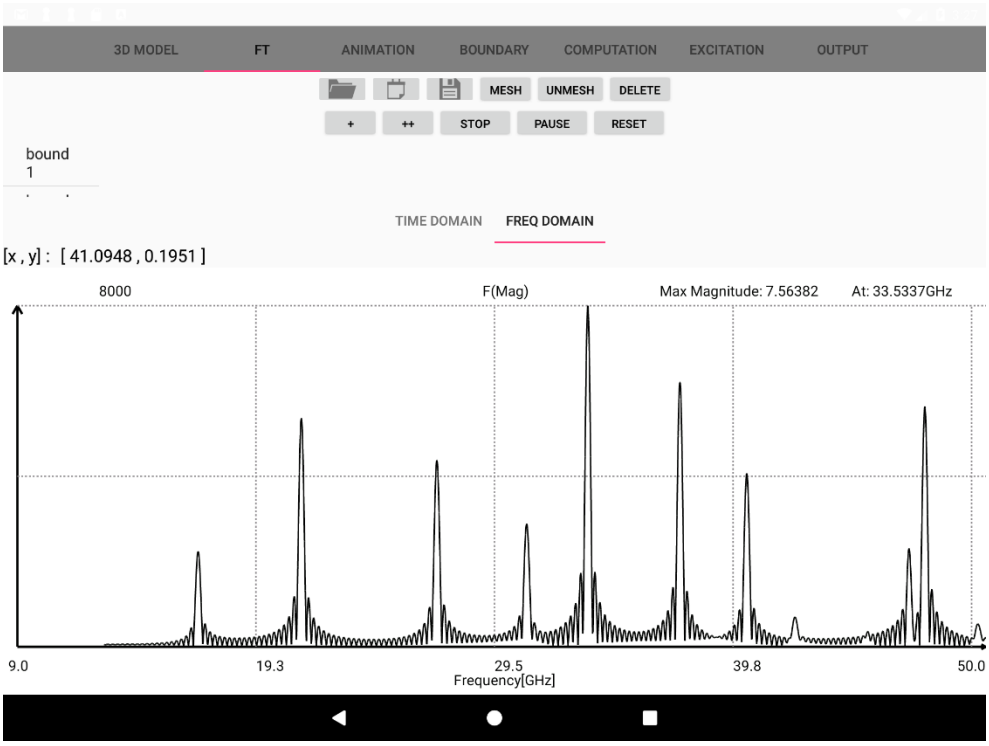


At Probe x=15, y=7

(b) Magnitude response of DFT at three output points in the app with $\Delta l = 0.5mm$.

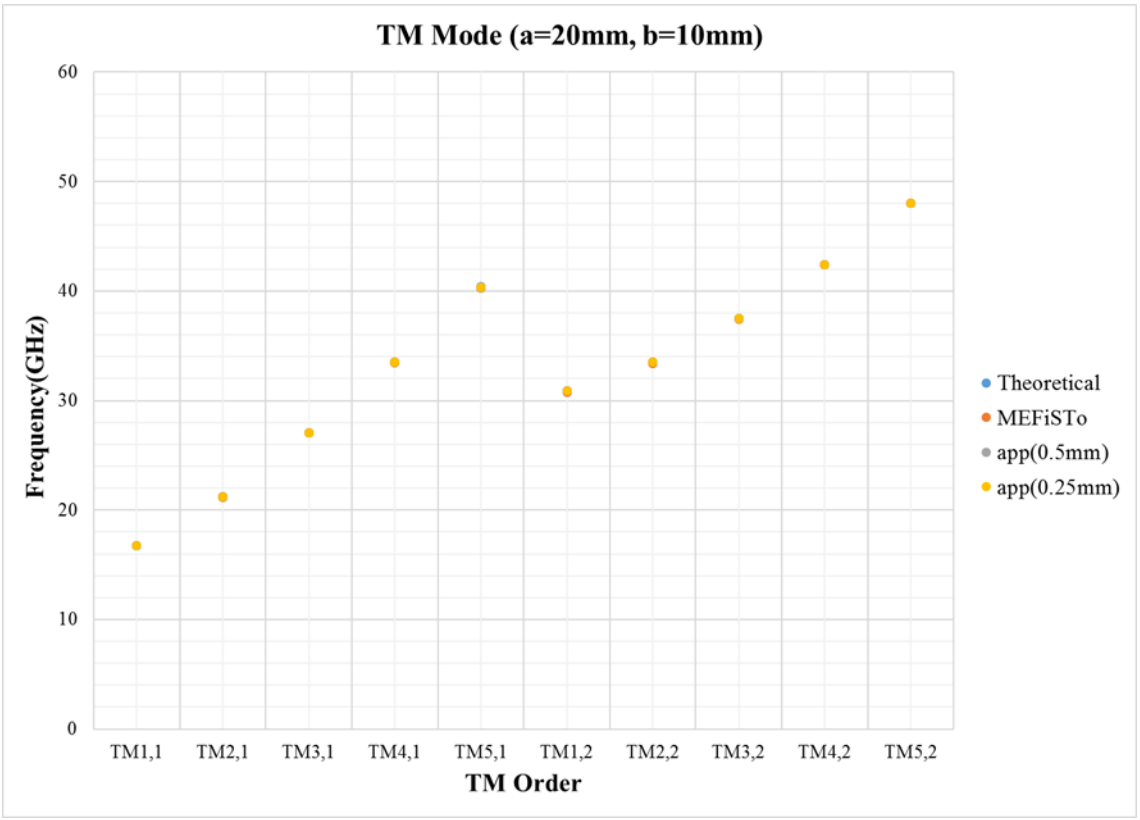

3D MODEL
FT
ANIMATION
BOUNDARY
COMPUTATION
EXCITATION
OUTPUT
MESH
UNMESH
DELETE
+
++
STOP
PAUSE
RESET
bound
1
TIME DOMAIN
FREQ DOMAIN
[x , y] : [30.674 , 1.5239]
8000
F(Mag)
Max Magnitude: 7.64134
At: 27.0336GHz
9.0
19.3
29.5
39.8
50.0
Frequency[GHz]


3D MODEL
FT
ANIMATION
BOUNDARY
COMPUTATION
EXCITATION
OUTPUT
MESH
UNMESH
DELETE
+
++
STOP
PAUSE
RESET
bound
1
TIME DOMAIN
FREQ DOMAIN
[x , y] : [41.0948 , 0.1951]
8000
F(Mag)
Max Magnitude: 6.94704
At: 33.5337GHz
9.0
19.3
29.5
39.8
50.0
Frequency[GHz]

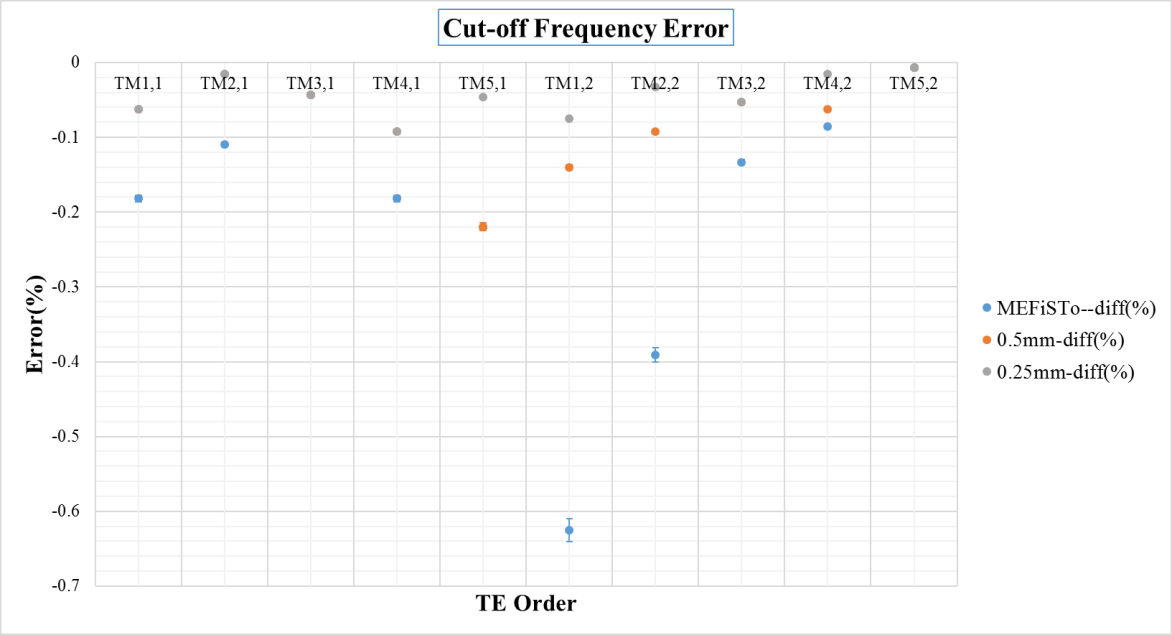


(c) Magnitude response of DFT at three output points in the app with $\Delta l = 0.25mm$.

Figure 28: The magnitude responses of DFT with $a = 20\text{mm}, b = 10\text{mm}$ in the TM mode.



(a) Cut-off frequencies obtained with the theoretical equations, MEFiSTo, and the app.

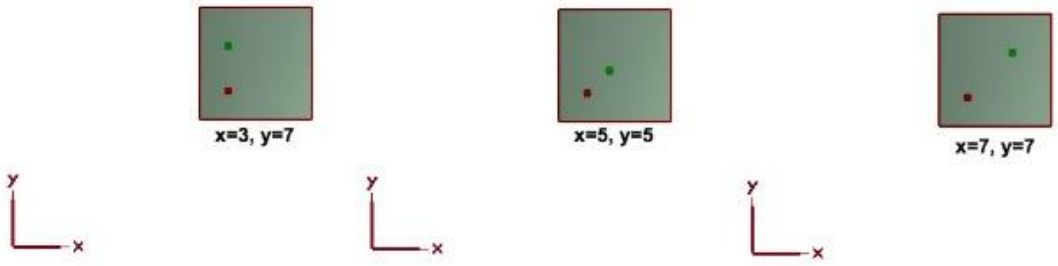


(b) The difference between the theoretical frequencies and those from MEFiSTo or the app.

Figure 29: Cut-off frequencies and differences with $a = 20\text{mm}, b = 10\text{mm}$ in the TM mode.

6.1.2 Test Case 2 – TM Mode in 10mm × 10mm Rectangular Waveguide

The second test case is $a = 10\text{mm}, b = 10\text{mm}$ for the TM mode. Three output probes are used in MEFiSTo and the TLM app. The TLM structure with various probes and the magnitude responses of DFT in MEFiSTo with $\Delta l = 1\text{mm}$ are in Figure 30. The TLM app also model the structure with two resolutions: $\Delta l = 0.5\text{mm}$ and $\Delta l = 0.25\text{mm}$. The frequencies obtained with the app, MEFiSTo and theoretical equations are shown in Figure 31. The figure also shows the frequency differences with the theoretical results.



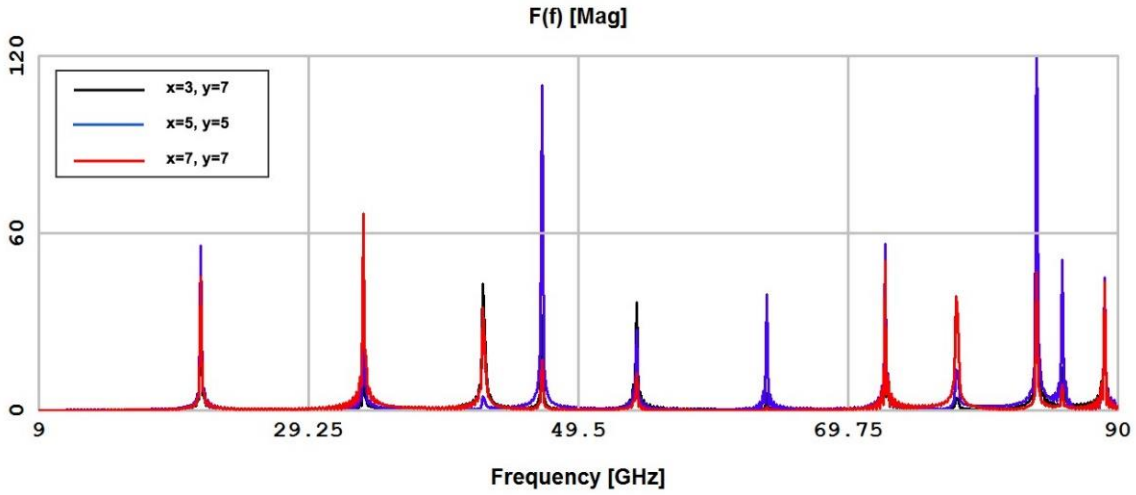
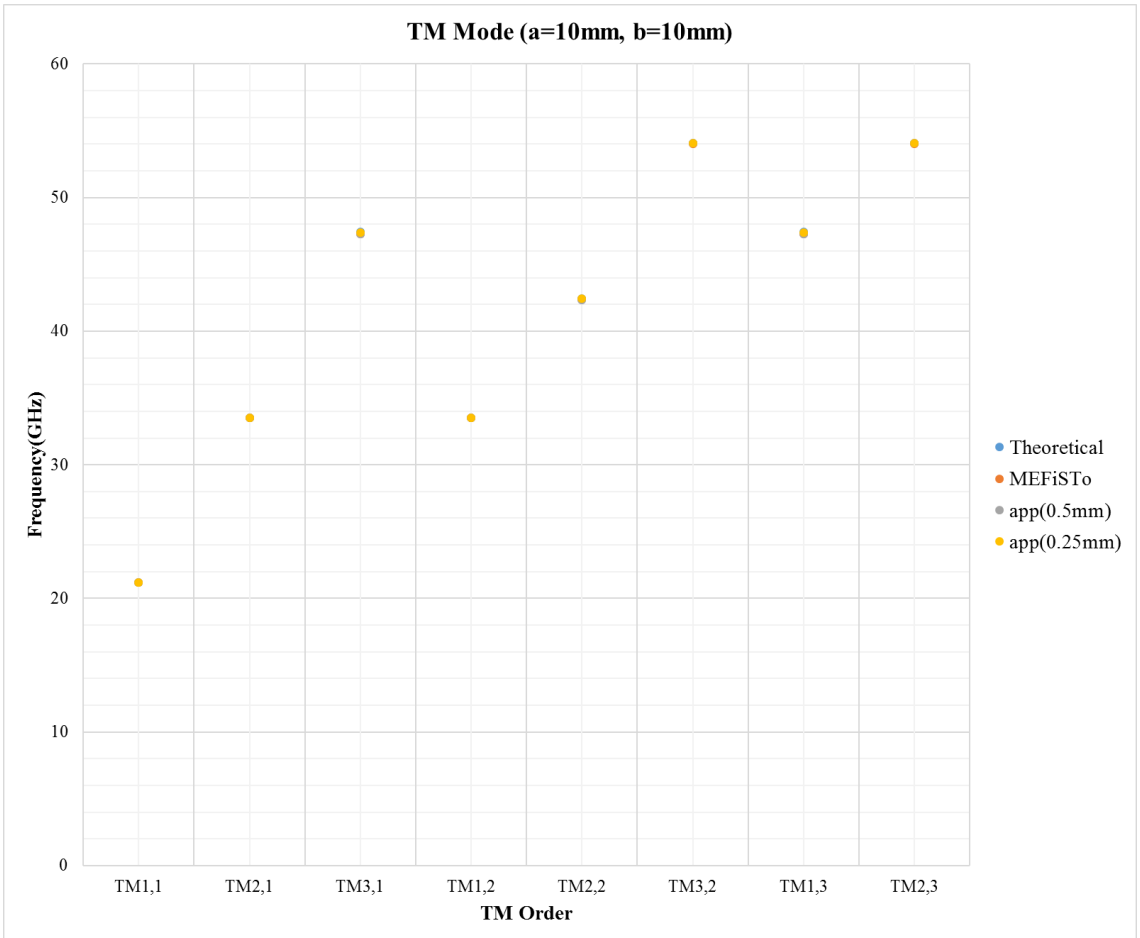
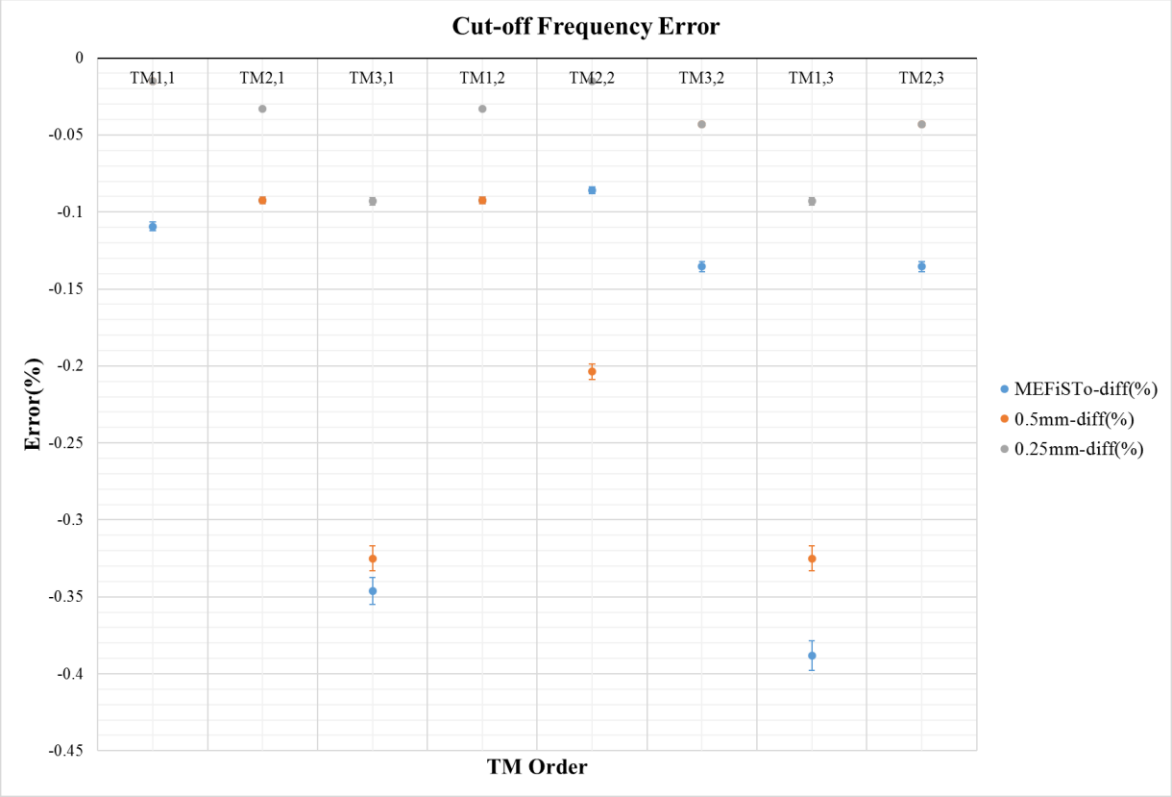


Figure 30: MEFiSTo-Magnitude response of DFT with $a = 10\text{mm}, b = 10\text{mm}$ in the TM mode.



(a) Cut-off frequencies obtained with the theoretical equations, MEFiSTo, and the app.

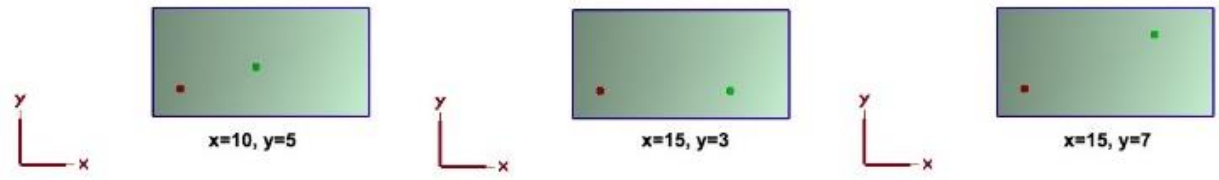


(b) The difference between the theoretical frequencies and ones from MEFiSTo or the app.

Figure 31: Cut-off frequencies and difference with $a = 10\text{mm}, b = 10\text{mm}$ in the TM mode.

6.1.3 Test Case 3 – TE Mode in $20\text{mm} \times 10\text{mm}$ Rectangular Waveguide

The third test case is $a = 20\text{mm}, b = 10\text{mm}$ for the TE mode. Figure 32 shows the TLM structure with three output probe positions and the magnitude responses of DFT obtained using MEFiSTo with $\Delta l = 1\text{mm}$. Figure 33 shows the cut-off frequencies from MEFiSTo, the app and theoretical equations. The figure also indicates the frequency differences.



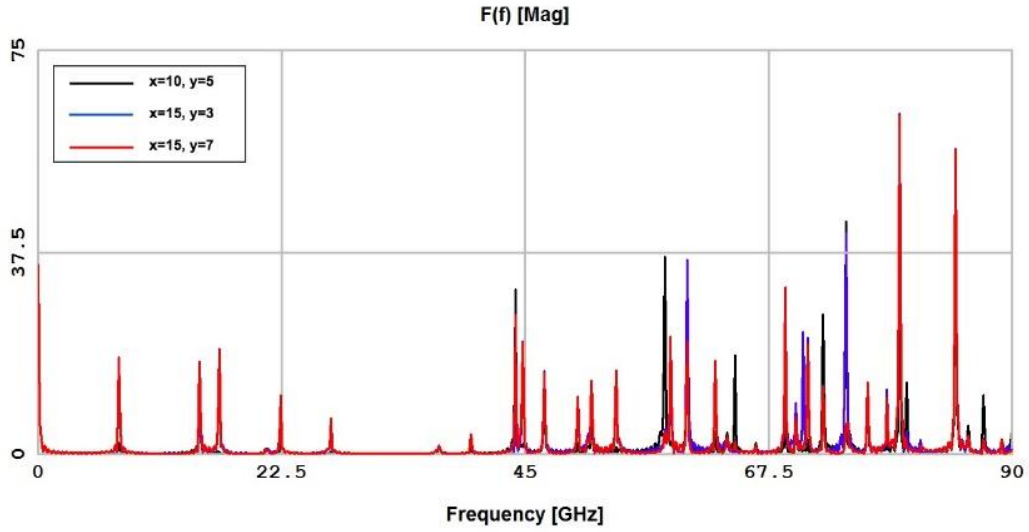
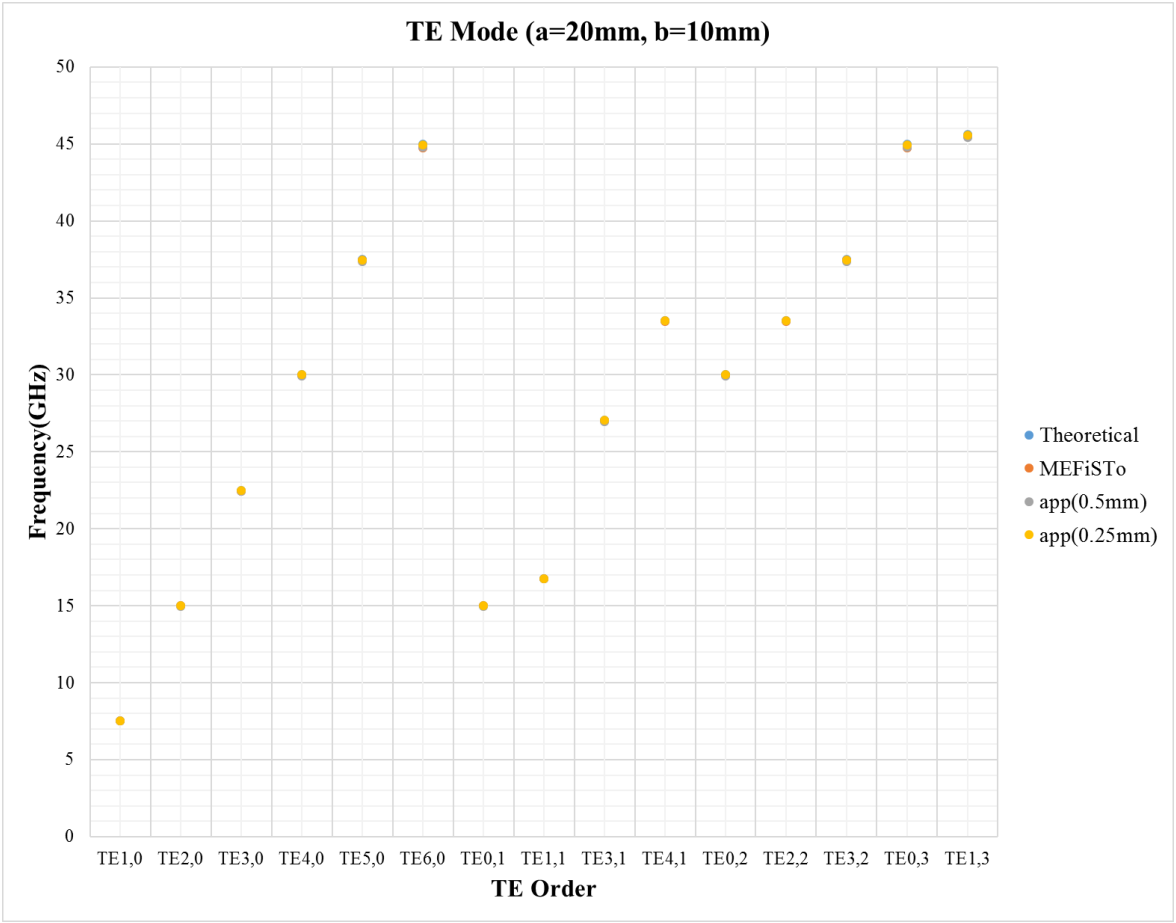
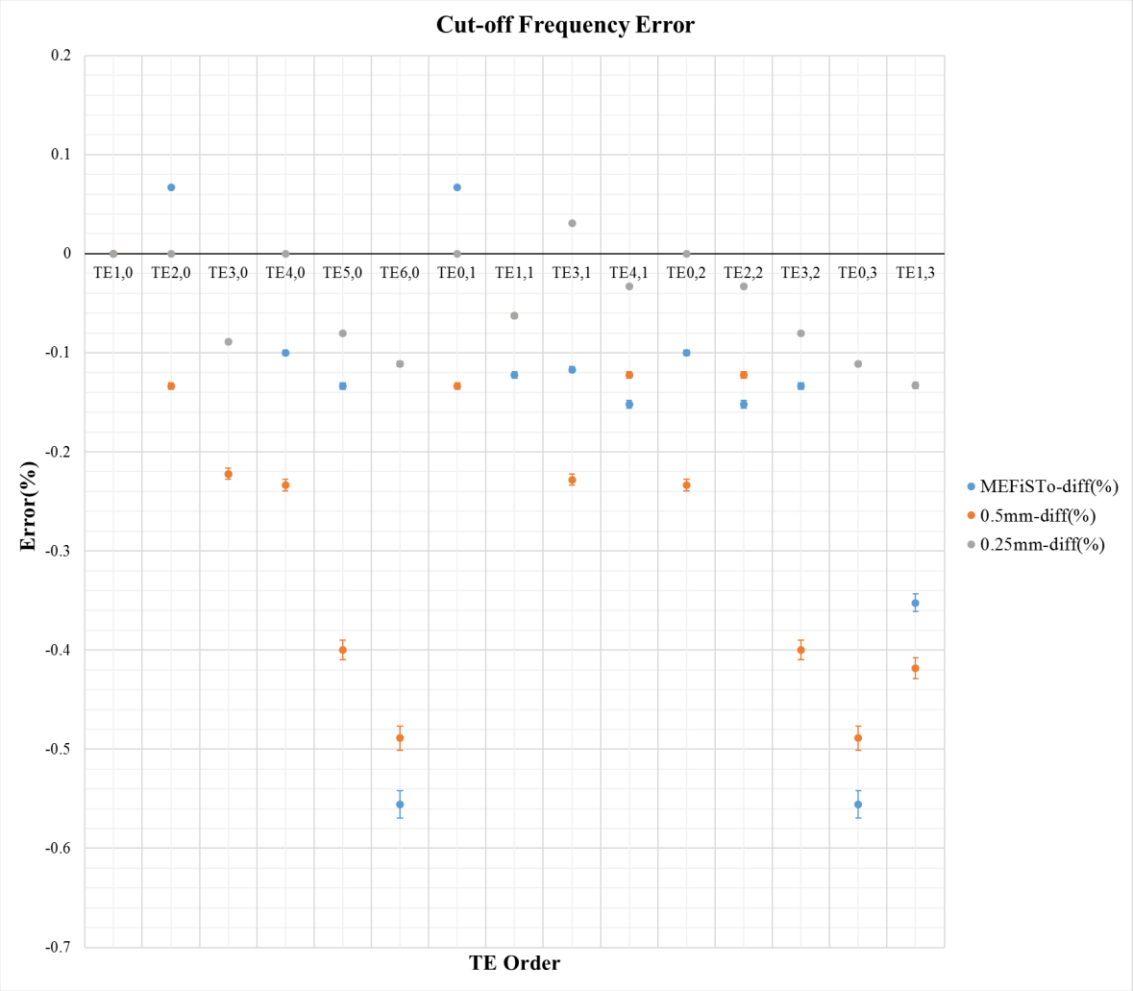


Figure 32: MEFiSTo-Magnitude response of DFT with $a = 20mm, b = 10mm$ in the TE mode.



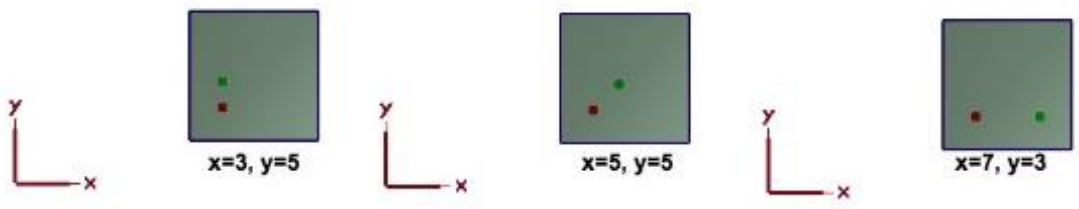
(a) Cut-off frequencies obtained with the theoretical equations, MEFiSTo, and the app.



(b) The difference between the theoretical frequencies and ones from MEFiSTo or the app.
Figure 33: Cut-off frequencies and difference with $a = 20mm, b = 10mm$ in the TE mode.

6.1.4 Test Case 4 – TE Mode in 10mm × 10mm Rectangular Waveguide

The fourth test case is $a = 10\text{mm}, b = 10\text{mm}$ for the TE mode. Figure 34 shows the TLM structure with three output probe positions and the magnitude responses obtained using MEFiSTo with $\Delta l = 1\text{mm}$. Figure 35 shows the cut-off frequencies from MEFiSTo, the app and theoretical equations; the figure also depicts the frequency differences.



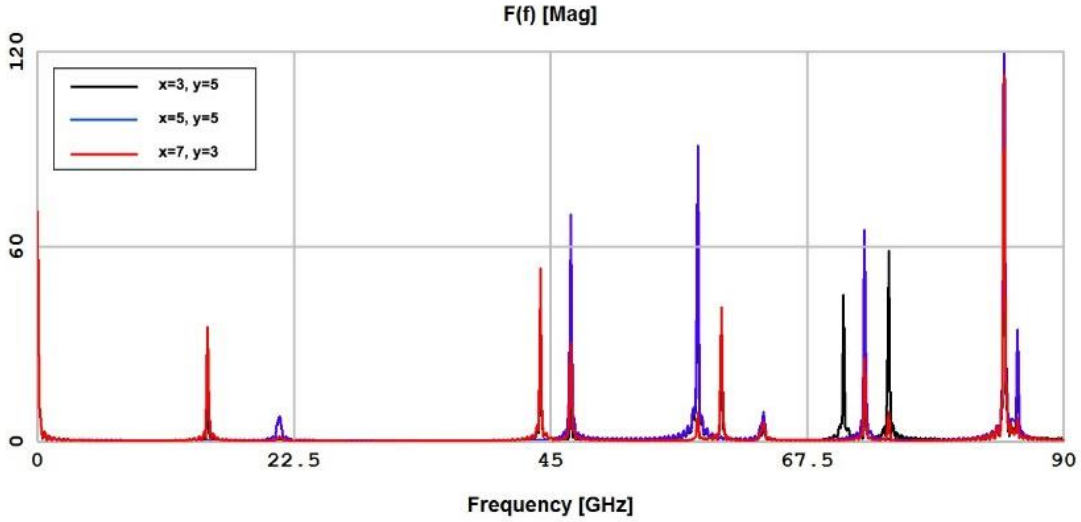
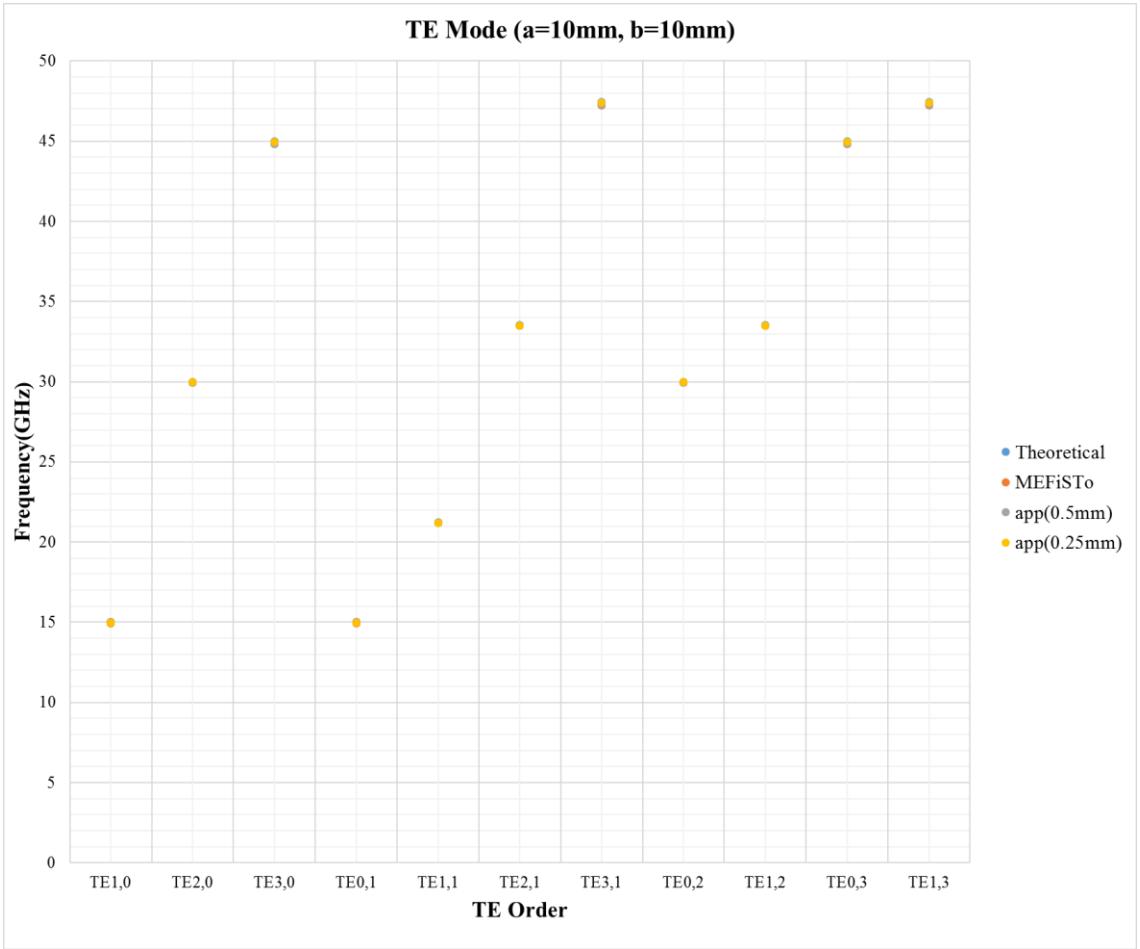
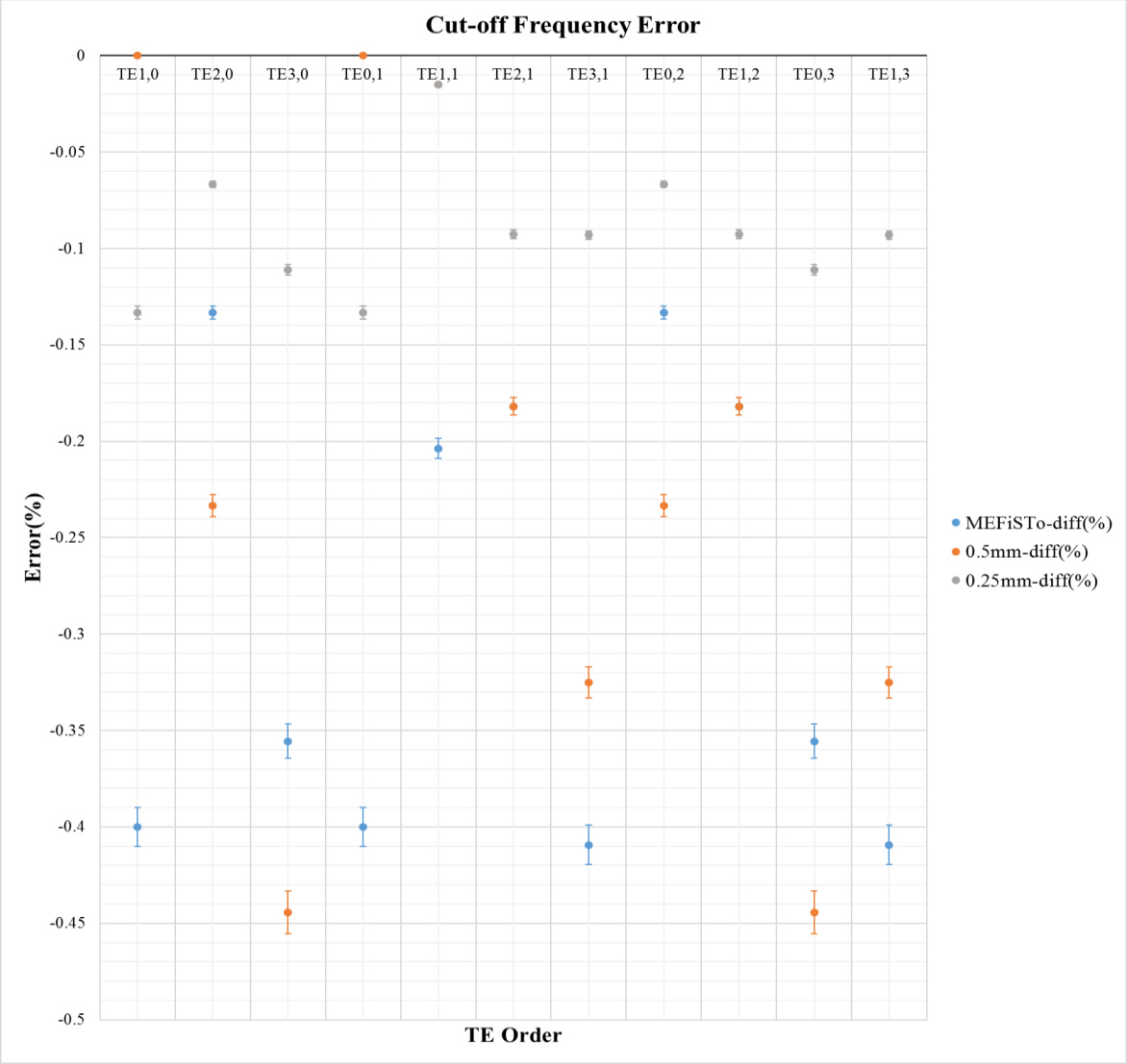


Figure 34: MEFiSTo-Magnitude response of DFT with $a = 10\text{mm}, b = 10\text{mm}$ in the TE mode.



(a) Cut-off frequencies obtained with the theoretical equations, MEFiSTo, and the app.



(b) The difference between the theoretical frequencies and those from MEFiSTo and the app.

Figure 35: Cut-off frequencies and difference with $a = 10\text{mm}, b = 10\text{mm}$ in the TE mode.

Chapter 7. Conclusion and Outlook

The results in figure 28–35 indicate that the cut-off frequencies obtained with the TLM app are very close to the results obtained with MEFiSTo and theoretical equations. The differences between the frequencies obtained from the TLM app and from analytical approach is within $[-0.7\%, +0.1\%]$ for $\Delta l = 0.5\text{mm}$ or smaller. This is a good confirmation that the TLM app is working correctly and thus can be used to model electromagnetic wave propagation in dielectric media.

Despite of that, there is room for improvements to make the developed TLM app executes better on Android. Two problems that we have encountered so far over the course of the TLM app development and the suggested solutions are given in below.

1. The current implementation has only one simulation thread to model TLM impulse propagation and one DFT thread to compute the cut-off frequencies. In this circumstance, the algorithm cannot take advantage of the multicore feature of most modern CPUs. If this app can employ as many threads as the number of cores in the CPU then the computing efficient should be increase proportionally to the number of cores. Since quad-core CPUs are readily available, it is possible to quadruple the speed of the TLM app if a multi-threaded implementation is employed.
2. The OpenGL thread crashes when the mesh resolution is high. Since the accuracy of the frequency result does not depend on the graphic resolution, the OpenGL drawing module may continue to use a lower resolution graphic mesh to avoid hitting the instability due to the OpenGL graphic library.

In addition to the above issues, future implementation may include automatic mesh discretization so that the app may handle arbitrary shape boundaries.

Bibliography

[1] Kevin Brothaler. *OpenGL ES 2 for Android: A Quick-Start Guide.* Pragmatic Bookshelf, 2013.

[2] Umberto Ravaioli Fawwaz T. Ulaby, Eric Michielssen. *Fundamentals Of Applied Electromagnetics.* Pearson, 2007.

[3] C. Huygens. *Traité de la Lumiére.* Leiden, 1690.

[4] Tatsuo Itoh. *Numerical Techniques For Microwave And Mullimeter-wave Passive Structures.* A Wiley-Interscience Publication, 1989.

[5] P. B. Johns and R. L. Beurle. *Numerical solution of 2-dimensional scattering problems using a transmission-line matrix. Proceedings of the Institution of Electrical Engineers,* 118(9):1203-1208, September 1971.

Appendix

1. The *handleTouchDown*, *handleDoubleTap*, and *handleMove* methods in the *renderer* class.

```
    public void handleTouchDown(float xevent, float yevent) {

        float xw = xevent;
        float yw = viewportMatrix[3] - yevent;
        getRay(xw, yw);


        if (!selected_boundary_name.isEmpty() || selected_boundary_name != null) {
            selected_boundary_name.clear();
        }

        for (String key : boundary_vertices.keySet()) {
            getTriangle(key);
            TriangleTest();
            if (test_result1 == 1 || test_result2 == 1) {
                selected_boundary_name.add(key);
            }
        }

        if (!selected_boundary_name.isEmpty()) {
            Log.d("NewEditor", "The chosen boundary is: " +
                Arrays.toString(boundary_vertices.get(selected_boundary_name.get(0))));
            selectedBoundary(selected_boundary_name.get(0));

            updatecallback.updatetextview(selected_boundary_name.get(0));
            updatecallback.setdeletelistener(selected_boundary_name.get(0));
            chosen_flag = true;
        } else {
            Log.d("NewEditor", "chosen boundary has NO element");
            if (!selected_boundary.isEmpty() || selected_boundary != null) {
                selected_boundary.clear();
            }
            chosen_flag = false;
        }

        updateVertexArray_flag = true;

    }
```

Listing 13: The *handleTouchDown* method.

```
public void handleDoubelTap() {

    if (chosen_flag) {
        getDoubleTapAlertDialogBuilder(context).create().show();
    }

}

public AlertDialog.Builder getDoubleTapAlertDialogBuilder(final Context context) {

    AlertDialog.Builder builder = new AlertDialog.Builder(context);

    builder.setIcon(android.R.drawable.ic_dialog_alert);
    builder.setCancelable(true);

    final ArrayAdapter<String> arrayAdapter = new ArrayAdapter<String>(context,
                                              android.R.layout.select_dialog_item);
    arrayAdapter.add("Delete");
    arrayAdapter.add("Copy");
```

```
    arrayAdapter.add("Paste");

    builder.setNegativeButton("cancel", new DialogInterface.OnClickListener() {
        @Override
        public void onClick(DialogInterface dialog, int which) {
            dialog.cancel();
        }
    });

    builder.setAdapter(arrayAdapter, new DialogInterface.OnClickListener() {
        @Override
        public void onClick(DialogInterface dialog, int which) {

            switch (which) {
                case 0:
                    getAlertDialogBuilder(context).create().show();
                    break;
                case 1:
                    if (chosen_flag) {
                        copyBoundary();
                        Toast.makeText(context, "Copy " +
                             selected_boundary_name.get(0), Toast.LENGTH_SHORT).show();
                    } else {
                        Toast.makeText(context, "Choose one boundary first.",
                                       Toast.LENGTH_SHORT).show();
                    }
                    break;
                case 2:
                    if (!copy_boundary.isEmpty()) {
                        pasteBoundary();
                        Toast.makeText(context, "Paste " +
                                    selected_boundary_name.get(0) + " with new name " +
                                    newBoundaryName + ".", Toast.LENGTH_SHORT).show();
                    } else {
                        Toast.makeText(context, "Copy one boundary first.",
                                       Toast.LENGTH_SHORT).show();
                    }
                    break;
            }

        }
    });

    return builder;

}

public void copyBoundary() {
    if (!copy_boundary.isEmpty() || copy_boundary != null) {
        copy_boundary.clear();
    }

    String key = selected_boundary_name.get(0);

    if (MainActivity.e_boundary.containsKey(key)) {
        boundaryTypeTag = 1;
    }
    if (MainActivity.m_boundary.containsKey(key)) {
        boundaryTypeTag = 2;
    }
    if (MainActivity.adi_boundary.containsKey(key)) {
        boundaryTypeTag = 3;
        copyCoeff = MainActivity.adi_coeff.get(key);
    }

    double[] boundary = boundary_vertices.get(key);
    double[] boundary_new = new double[8];

    switch (boundaryTypeTag) {
```

```
            case 1:
                newBoundaryName = "ew" + Integer.toString(MainActivity.e_boundary.size() +
                                                          1);
                break;
            case 2:
                newBoundaryName = "mag" + Integer.toString(MainActivity.m_boundary.size() +
                                                          1);
                break;
            case 3:
                newBoundaryName = "air-di" +
                                  Integer.toString(MainActivity.adi_boundary.size() + 1);
                break;
            default:
                newBoundaryName = "bound" + Integer.toString(boundary_vertices.size() + 1);
                break;
        }
        //vertical lines
        if (boundary[0] == boundary[1]) {
            if (boundary[0] + MainActivity.delta_l > maxLen - MainActivity.delta_l/2.0) {
                boundary new[0] = boundary[0] - MainActivity.delta l;
                boundary_new[1] = boundary[0] - MainActivity.delta_l;
            } else {
                boundary_new[0] = boundary[0] + MainActivity.delta_l;
                boundary_new[1] = boundary[0] + MainActivity.delta_l;
            }
            boundary new[2] = boundary[2];
            boundary_new[3] = boundary[3];
            boundary_new[4] = boundary[4];
            boundary_new[5] = boundary[5];
            boundary_new[6] = boundary[6];
            boundary_new[7] = boundary[7];
        } else {
            if (boundary[2] + MainActivity.delta_l > maxLen - MainActivity.delta_l/2.0){
                boundary_new[2] = boundary[2] - MainActivity.delta_l;
                boundary_new[3] = boundary[2] - MainActivity.delta_l;
            } else {
                boundary new[2] = boundary[2] + MainActivity.delta l;
                boundary_new[3] = boundary[2] + MainActivity.delta_l;
            }

            boundary_new[0] = boundary[0];
            boundary_new[1] = boundary[1];
            boundary new[4] = boundary[4];
            boundary_new[5] = boundary[5];
            boundary_new[6] = boundary[6];
            boundary_new[7] = boundary[7];
        }

        copy_boundary.put(newBoundaryName, boundary_new);

    }

    public void pasteBoundary() {

        MainActivity.saveStructure = false;

        double[] tmp = copy_boundary.get(newBoundaryName);
        boundary_vertices.put(newBoundaryName, tmp);

        switch (boundaryTypeTag) {
            case 1:
                MainActivity.e_boundary.put(newBoundaryName, tmp);
                break;
            case 2:
                MainActivity.m_boundary.put(newBoundaryName, tmp);
                break;
            case 3:
                MainActivity.adi_boundary.put(newBoundaryName, tmp);
                MainActivity.adi_coeff.put(newBoundaryName, copyCoeff);
                break;
```

```
    }
    MainActivity.wall_name.add(newBoundaryName);

    updatecallback.updatelistview();

    updateVertexArray_flag = true;
}
```

Listing 14: The *handleDoubleTap* method.

```
public void handleMove(float x, float y) {

    if (chosen_flag) {
        dx_pixel = maxLen / (viewportMatrix[2] - viewportMatrix[0]);
        dy_pixel = maxLen / (viewportMatrix[3] - viewportMatrix[1]);

        final float dx = x - mLastTouchX;
        final float dy = mLastTouchY - y;

        final String key_name = selected_boundary_name.get(0);
        final double[] boundary = boundary_vertices.get(key_name);
        final double[] boundary_new = new double[8];

        if (Math.abs(dx) > Math.abs(dy)) { // move horizontal
            double move_distance = ((int) (dx > 0 ?
                                    Math.ceil(dx_pixel * dx / MainActivity.delta_l) :
                                    Math.floor(dx_pixel * dx / MainActivity.delta_l)))
                                    * MainActivity.delta_l;
            Log.d("Move", "moving distance is: " + move_distance);
            //vertical lines
            if (boundary[0] == boundary[1]) {
                // out of boundary case
                if (boundary[0] + move_distance > maxLen + XMIN -
                                                  MainActivity.delta_l/2.0) {
                    boundary new[0] = maxLen + XMIN - MainActivity.delta l/2.0;
                    boundary new[1] = maxLen + XMIN - MainActivity.delta l/2.0;
                } else if (boundary[0] + move_distance < XMIN + MainActivity.delta_l /
                                                                        2.0) {
                    boundary_new[0] = XMIN + MainActivity.delta_l / 2.0;
                    boundary_new[1] = XMIN + MainActivity.delta_l / 2.0;
                } else {
                    boundary_new[0] = boundary[0] + move_distance;
                    boundary_new[1] = boundary[1] + move_distance;
                }
            //horizontal lines
            } else {
                // out of boudnary
                if (boundary[1] + move_distance > maxLen + XMIN -
                                                  MainActivity.delta_l/2.0) {
                    boundary_new[0] = maxLen + XMIN - MainActivity.delta_l/2.0 -
                                                    (boundary[1] - boundary[0]);
                    boundary_new[1] = maxLen + XMIN - MainActivity.delta_l/2.0;
                } else if (boundary[0] + move_distance < XMIN + MainActivity.delta_l /
                                                                        2.0) {
                    boundary_new[0] = XMIN + MainActivity.delta_l / 2.0;
                    boundary_new[1] = XMIN + MainActivity.delta_l / 2.0 + (boundary[1]
                                                    - boundary[0]);
                } else {
                    boundary new[0] = boundary[0] + move distance;
                    boundary_new[1] = boundary[1] + move_distance;
                }
            }

            boundary new[2] = boundary[2];
            boundary_new[3] = boundary[3];
            boundary_new[4] = boundary[4];
            boundary_new[5] = boundary[5];
            boundary_new[6] = boundary[6];
            boundary new[7] = boundary[7];
```

```
            Log.d("Move", "original position is: " + Arrays.toString(boundary));
            Log.d("Move", "moved position is: " + Arrays.toString(boundary_new));

        } else { // move vertical
            double move_distance = ((int) (dy > 0 ?
                                    Math.ceil(dy_pixel * dy / MainActivity.delta_l) :
                                    Math.floor(dy_pixel * dy / MainActivity.delta_l)))
                                    * MainActivity.delta_l;
            Log.d("Move", "moving distance is: " + move_distance);
            //horizontal lines
            if (boundary[2] == boundary[3]) {
                if (boundary[2] + move_distance > maxLen + YMIN -
                                                        MainActivity.delta_l/2.0) {
                    boundary_new[2] = maxLen + YMIN - MainActivity.delta_l/2.0;
                    boundary_new[3] = maxLen + YMIN - MainActivity.delta_l/2.0;
                } else if (boundary[2] + move_distance < YMIN +
                                                        MainActivity.delta_l/2.0) {
                    boundary_new[2] = YMIN + MainActivity.delta_l/2.0;
                    boundary_new[3] = YMIN + MainActivity.delta_l/2.0;
                } else {
                    boundary_new[2] = boundary[2] + move_distance;
                    boundary_new[3] = boundary[3] + move_distance;
                }
            // vertical lines
            } else {
                if (boundary[3] + move_distance > maxLen + YMIN -
                                                        MainActivity.delta_l/2.0){
                    boundary_new[2] = maxLen + YMIN - MainActivity.delta_l/2.0 -
                                                    (boundary[3] - boundary[2]);
                    boundary_new[3] = maxLen + YMIN - MainActivity.delta_l/2.0;
                } else if (boundary[2] + move_distance < YMIN +
                                                        MainActivity.delta_l/2.0){
                    boundary_new[2] = YMIN + MainActivity.delta_l/2.0;
                    boundary_new[3] = YMIN + MainActivity.delta_l/2.0 + (boundary[3] -
                                                        boundary[2]);
                } else {
                    boundary_new[2] = boundary[2] + move_distance;
                    boundary_new[3] = boundary[3] + move_distance;
                }
            }

            boundary_new[0] = boundary[0];
            boundary_new[1] = boundary[1];
            boundary_new[4] = boundary[4];
            boundary_new[5] = boundary[5];
            boundary_new[6] = boundary[6];
            boundary_new[7] = boundary[7];

            Log.d("Move", "original position is: " + Arrays.toString(boundary));
            Log.d("Move", "moved position is: " + Arrays.toString(boundary_new));
        }

        boundary_vertices.put(key_name, boundary_new);

        updateMovedBoundary(key_name, boundary_new);
        selectedBoundary(key_name);

        updatecallback.updatetextview(key_name);
        mLastTouchX = x;
        mLastTouchY = y;
        updateVertexArray_flag = true;
        printHashmap();
        MainActivity.saveStructure = false;
    }
}
```

Listing 15:The *handleMove* method.

2. The methods in *AnimationGL* class

```
public void createVertexData() {

    if (animation_meshIndex != null) {
        Log.d("June", "create vertex data");
        int offset = 0;
        int row = animation_meshIndex[3] - animation_meshIndex[2] + 1;
        int col = animation_meshIndex[1] - animation_meshIndex[0] + 1;

        animationVertexData = new float[row * col * POSITION_COMPONENT_COUNT];

        for (int j = animation_meshIndex[2]; j <= animation_meshIndex[3]; j++) {
            for (int i = animation_meshIndex[0]; i <= animation_meshIndex[1]; i++) {
                x = (float) (-1f + (i - 1) * MainActivity.delta_l *
                                                        TLMGLRenderer.GL_L);
                y = (float) (-1f + (j - 1) * MainActivity.delta_l *
                                                        TLMGLRenderer.GL_L);
                z = 0f;
                w = (float) (1f + (j - 1) * MainActivity.delta_l * TLMGLRenderer.GL_w);

                animationVertexData[offset++] = x;
                animationVertexData[offset++] = y;
                animationVertexData[offset++] = z;
                animationVertexData[offset++] = w;
            }
        }
    }

    animationVertexBuffer = ByteBuffer.allocateDirect(animationVertexData.length *
                        BYTE_PER_FLOAT).order(ByteOrder.nativeOrder()).asFloatBuffer();
}
```

Listing 16: The *createVertexData* method.

```
public void createIndexData() {
    Log.d("June", "create index data");
    int offset = 0;
    int row = animation_meshIndex[3] - animation_meshIndex[2] + 1;
    int col = animation_meshIndex[1] - animation_meshIndex[0] + 1;

    animationIndexData = new short[((col - 1) * row + (row - 1) * col) * 2];
    indexNum = ((col - 1) * row + (row - 1) * col) * 2;

    for (int i = 0; i < row; i++) {
        for (int j = 0; j < (col - 1); j++) {
            int left = i * col + j;
            int right = i * col + (j + 1);

            animationIndexData[offset++] = (short) left;
            animationIndexData[offset++] = (short) right;
        }
    }
    for (int j = 0; j < col; j++) {
        for (int i = 0; i < (row - 1); i++) {
            int up = i * col + j;
            int down = (i + 1) * col + j;
            animationIndexData[offset++] = (short) up;
            animationIndexData[offset++] = (short) down;
        }
    }

    animationIndexBuffer = ByteBuffer.allocateDirect(animationIndexData.length *
                        BYTE_PER_SHORT).order(ByteOrder.nativeOrder()).asShortBuffer();
}
```

Listing 17: The *createIndexData* method.

```
public void bindVertexData() {

    if (animationVertexData != null && animationVertexData.length > 0) {
        animationVertexBuffer.clear();
        animationVertexBuffer.put(animationVertexData);
        animationVertexBuffer.position(0);

        glBindBuffer(GL_ARRAY_BUFFER, vertexbufferId);
        glBufferData(GL_ARRAY_BUFFER, animationVertexBuffer.capacity() *
                     BYTE_PER_FLOAT, animationVertexBuffer, GL_DYNAMIC_DRAW);
        glBindBuffer(GL_ARRAY_BUFFER, 0);
    }
}
```

Listing 18: The *bindVertexData* method.

```
public void bindIndexData(){
    if (animationIndexData != null && animationIndexData.length > 0) {
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexbufferId);
        animationIndexBuffer.clear();
        animationIndexBuffer.put(animationIndexData);
        animationIndexBuffer.position(0);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, animationIndexBuffer.capacity() *
                     BYTE_PER_SHORT, animationIndexBuffer, GL_STATIC_DRAW);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
    }
}
```

Listing 19: The *bindIndexData* method.

```
public void drawAnimationRegion() {

    if (animationVertexBuffer != null) {
        glUniformMatrix4fv(uMatrixLocation, 1, false, TLMGLRenderer.pmMatrix, 0);

        glLineWidth(2f);
        glBindBuffer(GL_ARRAY_BUFFER, vertexbufferId);
        glVertexAttribPointer(aPositionLocation, POSITION_COMPONENT_COUNT, GL_FLOAT,
                              false, 0, 0);
        glEnableVertexAttribArray(aPositionLocation);

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexbufferId);
        glDrawElements(GL_LINES, indexNum, GL_UNSIGNED_SHORT, 0);
        glBindBuffer(GL_ARRAY_BUFFER, 0);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
    }
}
```

Listing 20: The *drawAnimationRegion* method.