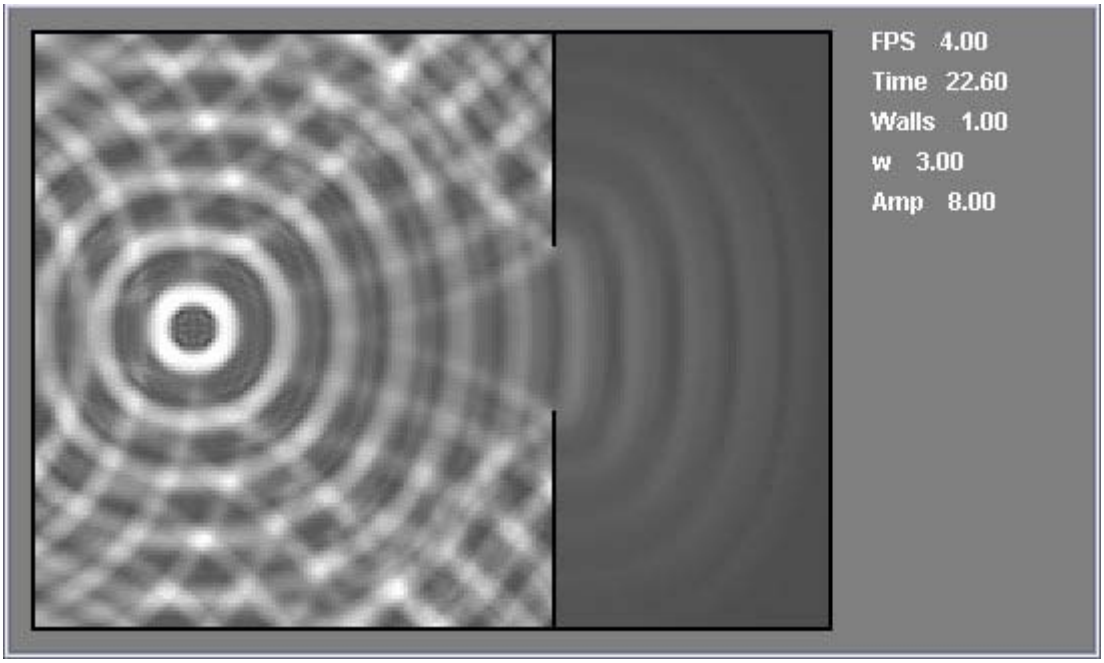
Modelling of Wave Propagation by Implementing the Transmission Line Matrix in C++



Tommy Hinks – tomhi761@student.liu.se
Johan Höglund – johho927@student.liu.se

Linköpings Universitet – ITN
Norrköping 2004-01-18

Abstract

This report describes basic Transmission Line Matrix theory, along with guidelines on how to implement TLM in C++; source code is included as an appendix. As can be seen in the source code, class-based implementation of the TLM method is a powerful approach, due to flexibility and straight-forward programming. A brief quantitative study of diffraction effects is discussed, with the conclusion that smaller slit width and larger wavelength increase diffraction effects. Also mentioned briefly is pressure distribution in a room. The visual effects of the application are good enough to appreciate acoustic wave propagation, but not adequate for scientific studies.

Index

Introduction

This section outlines the purpose of this project, identifies the methods that have been used and comments on where references have been sought.

Purpose

The main purpose of this report is to explain how sound wave propagation is modelled in a room by implementing the Transmission Line Matrix (TLM) method in C++. The application is used to study diffraction effects that occur when sound waves pass through a single slit of varying size. Finally there is a short general discussion about pressure distribution in the room.

Methods

The mathematical method used in the application is the TLM method, which will be further explained and validated later on. The application is class-based in order to make the source code easier to understand and to make updates and changes to the application simpler. For discussing diffraction effects screenshots from the application are taken. These are then edited with standard image editing programs to highlight what is interesting. The graphs appearing in this report have been produced by exporting values from the application to a text file, and then importing the text file as data into Microsoft Excel.

References

A number inside brackets ([x]) means that the preceding statement has a connection to the corresponding reference at the end of this report.

References have been sought mainly on the Internet and in literature from past courses. Articles found on the Internet are published work.

Modelling room acoustics using the transmission line method in C++

This section gives a brief introduction to the TLM method and explains how it was implemented in C++. After that data from the application is used as material to discuss diffraction effects and pressure distribution in the room.

Background theory of the TLM

The TLM method is a numerical method used to simulate wave propagation. TLM is based on Huygens' Principle, which is in modern terms is [1]:

> *Each point on a wave front acts as a source of secondary wavelets. At a later time, the envelope of the leading edges of the wavelets forms the new wave front.*

TLM was originally designed to model electromagnetic wave propagation, but can be used to model any phenomena that obeys Huygens' Principle, such as sound waves [1]. Figure 1 shows why the TLM method is a good numerical approximation of wave propagation according Huygens' Principle. At a time, $t_1 = t_0+\Delta t$, the wave front has travelled a distance $d=c\Delta t$ from the source in all directions. The TLM method's "wave front" only consists of four points at this stage and is not yet a very good approximation. At the next time instant, $t_2 = t_1+\Delta t$, the wave front has travelled a distance $2d$ from the source. The TLM method's "wave front" has also travelled further away from the source and is beginning to look more and more spherical. One can easily imagine the next few iterations of the TLM method and see that the two different wave fronts will grow more and more similar.

The TLM method makes use of an array of nodes, between which, in acoustic applications, acoustic pressure (or acoustic potential) is exchanged. In two dimensions a rectangular grid of nodes is setup.

If the medium in which the waves are propagating is homogeneous, has the same properties everywhere, one can assume that neighbouring nodes will spread pressure to each other in the same way[1]. This makes it possible to outline a reusable method for one node, which can then be used on all nodes.

The scattering principle used in our application assumes that the medium in the room is homogeneous and that the medium is non-viscos, so that there is no resistance when transmitting pressure. These assumptions have been made by others before us so a well-known scattering model exists [2] [3]. Figure 2 shows the different parts of a node and equation (1) explains how pressure is transmitted between nodes.

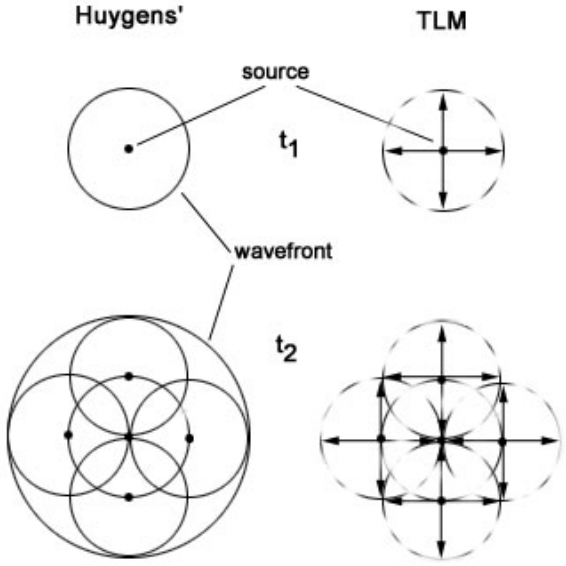


Figure 1. TLM wave propagation compared to Huygens' Principle.

[1] In our application the walls of the room are also represented as nodes and do not scatter pressure the same way as the nodes representing the medium in the room.

$$(1) \qquad \begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \end{bmatrix}_{t+1}^{Out} = \frac{1}{2} \begin{bmatrix} -1 & 1 & 1 & 1 \\ 1 & -1 & 1 & 1 \\ 1 & 1 & -1 & 1 \\ 1 & 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \end{bmatrix}_t^{In}$$

The above equation describes how a node scatters incoming pressure (the right hand vector). The pressure that arrived at a node at time *t*, is transmitted to neighbouring nodes at *t+1*. As *t* increases, as the application is run, wave propagation is modelled in the room.

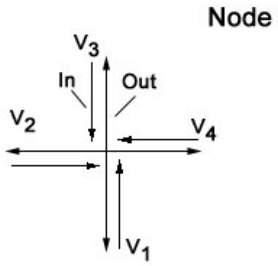


Figure 2. A node

Implementing the TLM method in C++

The implementation of the grid of nodes and the wave propagation was made in C++ together with OpenGL, to make visualization more effective. The application is divided into several classes to make it more efficient and easier to adjust as work progresses and gets more complex. The top class is Room, which in some way contains all the other classes. Most importantly an instance of the class Room contains the grid of nodes that is used in the TLM method. Each node is represented on the screen as a two by two pixel square that has a colour that depends on the amount of pressure currently present in the node. Every node has two vectors that represent the flow of pressure between the nodes. One vector to keep track of incoming pressure and one to keep track of outgoing pressure. The program uses the TLM methods to distribute all the pressure to the neighbouring nodes. First it scatters the pressure between the nodes, and then it collects the incoming pressure, summarizes it and sets the new colour of the nodes. These four steps are repeated as long as the application is running.

A sinusoidal source was introduced at a node in the room, adding pressure to the incoming vector of this node at each time instant. Without a source of some kind there is no pressure to be scattered between the nodes.

Virtual microphones in our application simply listen to a certain node and record the pressure there for as long as the application is running. This recording is output to a text file, where a pressure is given for each time instant. The pressure is measured precisely after the incoming pressure for the node is summarized.

The source code for the application is available in Appendix 1.

Dealing with boundaries

The walls of the room were modelled almost like regular nodes, with some crucial differences. A wall node has a static colour and the incoming pressure is reflected only in the same direction as it came from. Every instance of the class Room has a wall coefficient which determines how much of the incoming pressure is to be reflected back into the room. The wall coefficient varies between zero and one, a value of one makes the room an echo chamber and a value of zero means that the walls don't reflect any pressure at all.

Diffraction effects

Diffraction occurs when waves travel trough an opening that is relatively smaller then the wavelength [1]. To create these effects in our application, an extra wall was added to the room dividing the room in two. This wall had an opening to allow waves to pass through it. The angle of diffraction was measured during a couple of experiments to find a relation between the angle of diffraction, the size of the opening and the wavelength. See figure 4-7.

Pressure distribution in the room

Several recordings were made with a virtual microphone to study the pressure distribution in the room. The microphone was placed at two randomly selected nodes and also on five nodes along a line perpendicular to the normal of the opening. See figure 8-9.

Results

The application shows how waves propagate in a room. It shows pressure at different points at different times. The application also handles reflecting boundaries, in our case walls. Any kind of source could be implemented and run; the sources are not limited to being sinusoidal. Figure 3 shows what the application produces when running.

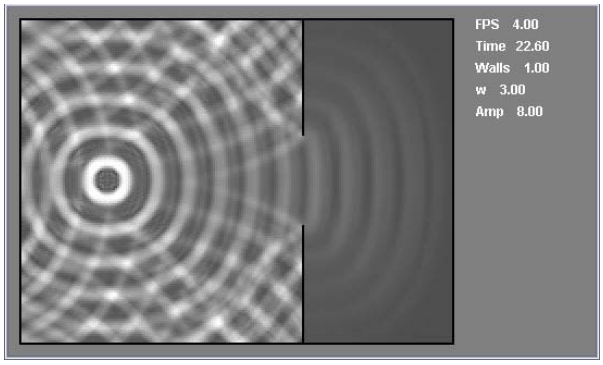


Figure 3. Final view of the application

Diffraction effects were studied by using screenshots from the application. The source was placed at position (40, 75). The diffraction angle was measured after the same amount of time had passed and is shown in the four figures below. The amplitude of the source has been increased a lot in order to see the diffraction effects better, giving a rather strange look to the region around the source. More screenshots were taken and the results recorded, but those screenshots are not shown here.

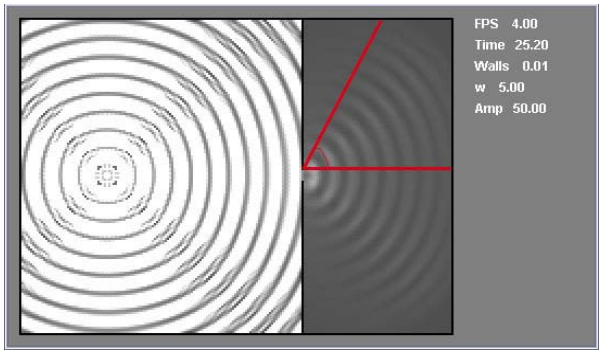


Figure 4. Diffraction effects studied by varying width of the slit. Slit width = 4 units. Angle of diffraction = 63 deg.

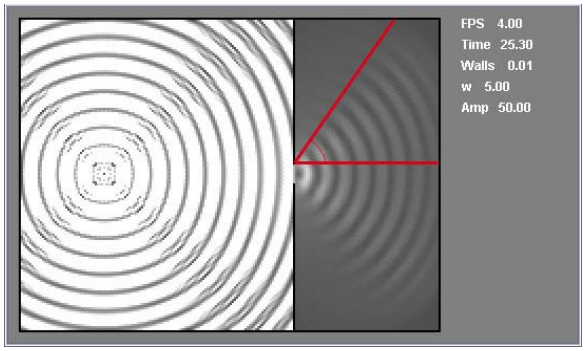


Figure 5. Diffraction effects studied by varying width of the slit. Slit width = 8 units. Angle of diffraction = 55 deg.

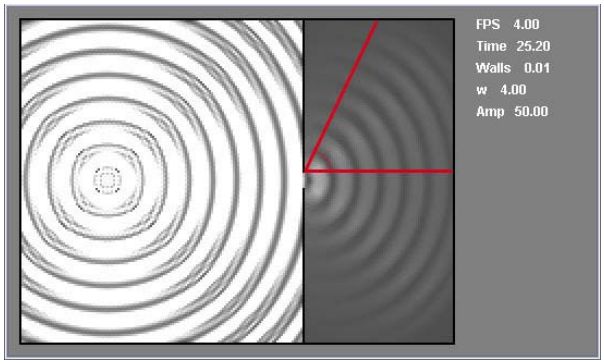


Figure 6. Diffraction effects studied by varying frequency (wavelength). Frequency = 4 Hz. Angle of diffraction = 64 deg.

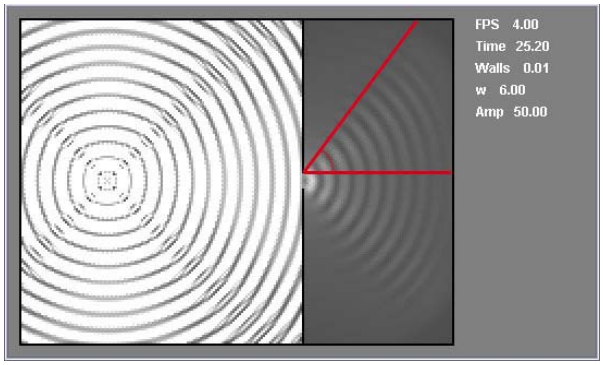


Figure 7. Diffraction effects studied by varying frequency (wavelength). Frequency = 6 Hz. Angle of diffraction = 53 deg.

The total results of the diffraction study are shown in the table below.

Frequency (Hz)	Angle of diffraction (deg)	Slit width (units)
2	70	6
3	65	6
4	64	6
5	60	6
6	56	6
7	48	6
8	44	6
5	65	2
5	64	4
5	62	6
5	58	8
5	55	10
5	53	12

Table 1. Total results of diffraction study.

Pressure distribution in the room was briefly studied by using virtual microphones placed at different positions in the room. The source was placed at position (50, 50). The graphs obtained are shown in figures 8-9.

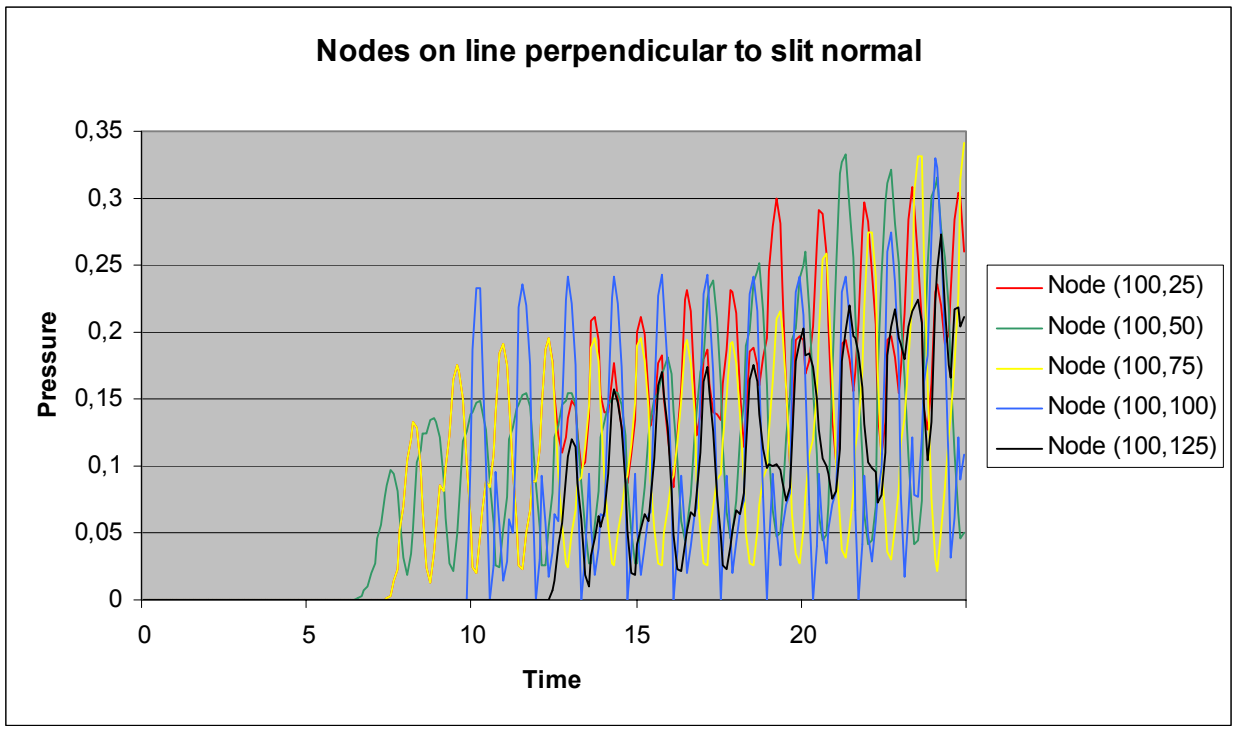


Figure 8. Pressure at nodes on a line perpendicular to the normal vector of the opening. Five nodes ploted in the same diagram.

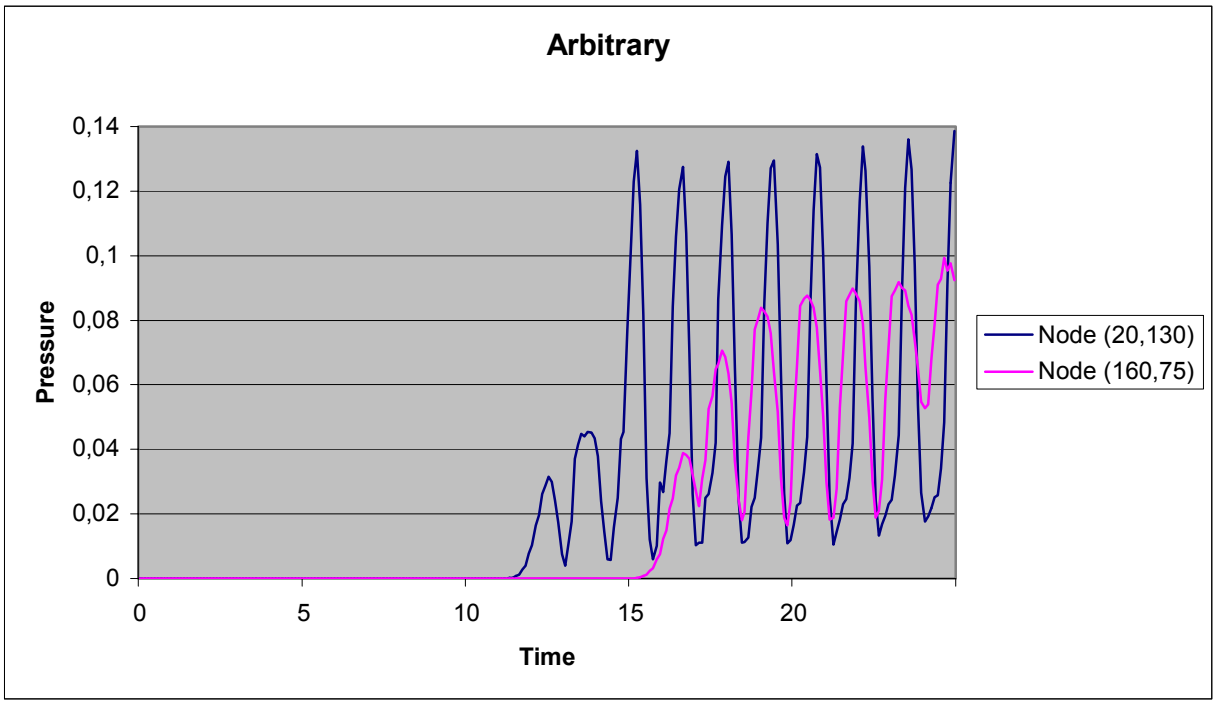


Figure 9. Pressure at two arbitrary nodes in the room.

Discussion

Our method has limitations. One of these is the fact that the application becomes computationally heavy when the number of nodes becomes large. This means that it would take a long time to model large rooms. Another aspect of the same problem is that the application cannot simulate the speed of wave propagation. This is simply because the application runs as fast as possible (which varies from computer to computer), meaning that the wave front moves with different velocities on different machines. The colour representation is not always correct due to the fact that there is a maximum shade of white that can be produced by the monitor. This means nodes with the same colour may in fact have different pressure.

The way the source code is built makes it easy to add new features, such as multiple or moving sources, or extra walls. It is also easy to change the colour representation to suit specific purposes.

The results of the diffraction study are presented in figures 4-7 and table 1. What can be noticed are two things: the diffraction angle increases as the slit becomes smaller[2], the diffraction angle increases as the frequency (wavelength) decreases[3]. The measurements that were made did not show more than trends. It was too difficult to measure the diffraction angle from the screenshot, even after the amplitude was greatly increased.

Figure 8 shows pressure distribution along a line perpendicular to the slit normal. The nodes closest to the source (green and yellow lines) are reached by the wave front first. The fact that they don't rise to full amplitude directly is natural, since the top of the wave hasn't reached the node yet. In terms of the TLM method this is modelled by the fact that these nodes are not receiving pressure from all their surrounding nodes, but merely from one or two neighbours. When the wave passes over the node, it receives pressure from all surrounding nodes and reaches the full amplitude of the wave. After a while wall reflections, as well as direct sound, reach some of the nodes in figure 8. By studying the black line, one can see that just before time = 20 the pressure level rises. This would never happen if there were no walls.

The blue curve in figure 9 shows a microphone placed in the top left corner. Almost immediately after the direct sound has reached the node, the reflections from the walls hit the node, hence the pressure spikes. The pink line has lower pressure amplitude because it is further away from the source. The pink line doesn't make the same kind of pressure leap as the blue line. This is because it is further away, and is not hit by any reflections.

[2] Constant frequency (wavelength).
[3] Constant slit width. As frequency decreases, wavelength increases.

Conclusion

The application could be used to model simple environments, where details can be ignored, and make quantitative studies of how acoustic waves behave. As was mentioned in the discussion it is hard to make qualitative studies of physical phenomena using the application. It might be possible to enhance the application, and run it on a powerful computer, to obtain high-resolution images, with more detailed information. The TLM method works well, but without optimized algorithms and super computers it can be computationally heavy and difficult to put into practical use.

References

A list of references used in this report.

[1] Benson, Harris (1995). *University Physics*. Wiley. United States.

[2] Mansour, Ahmadian (2001). *Transmission Line Matrix (TLM) modelling of medical ultrasound*. Accessed online 2004-02-20. <http://oldeee.see.ed.ac.uk/SaS/Thesis/Ma/thesis.pdf>. University of Edinburgh.

[3] Razvan, Ciocan (2003). *Applications of Transmission Line Matrix Method For NDT*. Accessed online 2004-02-02. <http://www.ndt.net/article/ecndt02/319/319.htm>. University of Akron.

Appendix 1 Source Code

```
#include <windows.h>                    // Header File For Windows

#include <iostream>
#include <fstream>

#include <stdarg.h>                     // Header File For Variable Argument Routines
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#include <gl\gl.h>                  // Header File For The OpenGL32 Library
#include <gl\glu.h>                     // Header File For The GLu32 Library
#include <gl\glaux.h>                   // Header File For The Glaux Library

using namespace std;

HDC         hDC=NULL;                   // Private GDI Device Context
HGLRC           hRC=NULL;                       // Permanent Rendering Context
HWND            hWnd=NULL;                      // Holds Our Window Handle
HINSTANCE       hInstance;              // Holds The Instance Of The Application

bool    keys[256];                      // Array Used For The Keyboard Routine
bool    active=TRUE;                    // Window Active Flag Set To TRUE By Default
bool    fullscreen=false;               // Fullscreen Flag Set To Fullscreen Mode By Default

bool bKeyUp, bKeyDown;                  // Determines if a key is pressed

#define     RX  200                     // Room dimensions
#define     RY  150



GLuint      base;                       // Base Display List For The Font


GLvoid glPrint(const char *fmt, ...)                    // Custom GL "Print" Routine
{
        char        text[256];                          // Holds Our String
        va_list             ap;                         // Pointer To List Of Arguments

        if (fmt == NULL)                                // If There's No Text
                return;                                 // Do Nothing

        va_start(ap, fmt);                              // Parses The String For Variables
           vsprintf(text, fmt, ap);                     // And Converts Symbols To Actual Numbers
        va_end(ap);                                             // Results Are Stored In Text

        glPushAttrib(GL_LIST_BIT);                      // Pushes The Display List Bits
        glListBase(base - 32);                          // Sets The Base Character to 32
        glCallLists(strlen(text), GL_UNSIGNED_BYTE, text);      // Draws The Display List Text
        glPopAttrib();                                  // Pops The Display List Bits
}


GLvoid BuildFont()                                      // Build Our Bitmap Font
{
        HFONT       font;                                       // Windows Font ID
        HFONT       oldfont;                                    // Used For Good House Keeping

        base = glGenLists(96);                                  // Storage For 96 Characters

        font = CreateFont(      -12,                            // Height Of Font
                                0,                              // Width Of Font
                                0,                              // Angle Of Escapement
                                0,                              // Orientation Angle
                                FW_BOLD,                        // Font Weight
                                FALSE,                          // Italic
                                FALSE,                          // Underline
                                FALSE,                          // Strikeout
                                ANSI_CHARSET,                   // Character Set Identifier
```

```
                                OUT_TT_PRECIS,                  // Output Precision
                                CLIP_DEFAULT_PRECIS,            // Clipping Precision
                                ANTIALIASED_QUALITY,            // Output Quality
                                FF_DONTCARE|DEFAULT_PITCH,      // Family And Pitch
                                "Arial");                       // Font Name

        oldfont = (HFONT)SelectObject(hDC, font);               // Selects The Font We Want
        wglUseFontBitmaps(hDC, 32, 96, base);                   // Builds 96 Characters Starting At Character 32
        SelectObject(hDC, oldfont);                             // Selects The Font We Want
        DeleteObject(font);                                     // Delete The Font
}


GLvoid KillFont(GLvoid)                                         // Delete The Font From Memory
{
        glDeleteLists(base,256);                                // Delete All 256 Display Lists
}



class Vector4 {                                                 // Vector class

public:

        Vector4::Vector4()
        {
                return;
        };

        Vector4::Vector4(float tx, float ty, float tz, float tw)
        {
                v1 = tx;
                v2 = ty;
                v3 = tz;
                v4 = tw;
                return;
        };

        float Vector4::getV1() { return (v1); };
        float Vector4::getV2() { return (v2); };
        float Vector4::getV3() { return (v3); };
        float Vector4::getV4() { return (v4); };

        void Vector4::setV1(float t) { v1 = t; };
        void Vector4::setV2(float t) { v2 = t; };
        void Vector4::setV3(float t) { v3 = t; };
        void Vector4::setV4(float t) { v4 = t; };


private:
        float v1,v2,v3,v4;
};

class Source {                                                  // Source class
public:
        Source::Source(int tx, int ty, float tamp, float tfreq)     // Creates a source
        {
                x = tx;
                y = ty;
                amp = tamp;
                freq = tfreq;
                return;
        };

        void Source::setAmp(float t) { amp = t; };              // Set/get the different values of a source
        void Source::setFreq(float t) { freq = t; };
        void Source::setX(int t) { x = t; };
        void Source::setY(int t) { y = t; };

        int Source::getX() { return (x); };
        int Source::getY() { return (y); };
        float Source::getFreq() { return (freq); };
        float Source::getAmp() { return (amp); };
```

```
private:
	int x, y;
	float amp, freq;
};


class Node {                                        // Node Class
public:

	Node::Node()                                    // Creates a node with standard wall coefficient
	{
		in.setV1(0.0f);
		in.setV2(0.0f);
		in.setV3(0.0f);
		in.setV4(0.0f);

		out.setV1(0.0f);
		out.setV2(0.0f);
		out.setV3(0.0f);
		out.setV4(0.0f);

		setR(0.3f);
		setG(0.3f);
		setB(0.3f);

		return;
	};

	Node::Node(float w)                             // Creates a node
	{
		wall = w;

		in.setV1(0.0f);
		in.setV2(0.0f);
		in.setV3(0.0f);
		in.setV4(0.0f);

		out.setV1(0.0f);
		out.setV2(0.0f);
		out.setV3(0.0f);
		out.setV4(0.0f);

		setR(0.7f);
		setG(0.7f);
		setB(0.7f);

		return;
	};

	float Node::getR() { return (r); };             // Read and set the color coefficients
	float Node::getG() { return (g); };
	float Node::getB() { return (b); };

	void Node::setR(float t) { r = t; };
	void Node::setG(float t) { g = t; };
	void Node::setB(float t) { b = t; };



	Vector4 Node::getIn() {return (in); };          // Read the energy vectors
	Vector4 Node::getOut() {return (out); };

	void Node::setW(float t) { wall = t; };         // Set and receiv the wall coefficient
	float Node::getW() { return (wall); };

	void Node::setIn(Vector4 t) {                   // Set the enrgy vectors
		in.setV1(t.getV1());
		in.setV2(t.getV2());
		in.setV3(t.getV3());
		in.setV4(t.getV4());
	}

	void Node::setOut(Vector4 t) {
```

```
		out.setV1(t.getV1());
		out.setV2(t.getV2());
		out.setV3(t.getV3());
		out.setV4(t.getV4());
	}

	float Node::sumIn() { return (in.getV1() + in.getV2() + in.getV3() + in.getV4()); };		// Sumarizes the total engry
presint in a node


	void Node::scatter() {				// Scatters the energy from a node to another

		if(this->getW()>0) {
			this->out.setV1(this->in.getV1() * this->getW());
			this->out.setV2(this->in.getV2() * this->getW());
			this->out.setV3(this->in.getV3() * this->getW());
			this->out.setV4(this->in.getV4() * this->getW());
		}
		else
		{
			this->out.setV1(0.5f*(-this->in.getV1() + this->in.getV2() + this->in.getV3() + this->in.getV4()));
			this->out.setV2(0.5f*(this->in.getV1() - this->in.getV2() + this->in.getV3() + this->in.getV4()));
			this->out.setV3(0.5f*(this->in.getV1() + this->in.getV2() - this->in.getV3() + this->in.getV4()));
			this->out.setV4(0.5f*(this->in.getV1() + this->in.getV2() + this->in.getV3() - this->in.getV4()));

		}

		this->in.setV1(0.0f);
		this->in.setV2(0.0f);
		this->in.setV3(0.0f);
		this->in.setV4(0.0f);
	};



private:
	Vector4 in, out;		// Vectors for the energy
	float r, g, b;		// The colour of a node
	float wall;		// Wall Coefficient. If 0 the node is not a wall, if 1 the node is a perfectly reflecting wall



};

class Room {		// Room class
public:


	Room::Room(float tc, float tw, float tb) {	// Creates a room with a grid of nodes
		c = tc;
		time = 0;
		tb=tb/2;
		for(int i=0; i<RX; i++)
		{
			for(int j=0; j<RY; j++)
			{
				if(i<1 || i>RX-2 || j<1 || j>RY-2 )		// Creates the nodes and set the edge nodes as walls
					grid[i][j] = Node(tw);
				else
					grid[i][j] = Node(0.0f);

				if(i==130 && (j<(75-tb) || j>(75+tb))) // Adds the wall between the rooms
					grid[i][j].setW(tw);
			}
		}

		return;
	};

	void Room::setC(float t) { c = t; };		// Set/get the values of a room

	float Room::getTime() { return (time); };
```

```
void Room::setTime(float t) { time = t; };

float Room::getFps() { return (fps); };

void Room::drawRoom() {                                          // Draws the room and sets the colours

        for(int x=0; x < RX; x++)
        {
                for(int y=0; y < RY; y++)
                {
                        if(grid[x][y].getW()>0)
                                glColor3f(0.0f, 0.0f, 0.0f);            // Wall colour

                        else
                                glColor3f(grid[x][y].getR(),grid[x][y].getG(),grid[x][y].getB()); // Regular node color


                        glBegin(GL_QUADS);
                        glVertex2i(2*x, 2*y);                           // Left And Up 1 Unit (Top Left)
                        glVertex2i(2*(x+1), 2*y);                       // Right And Up 1 Unit (Top Right)
                        glVertex2i(2*(x+1),2*(y+1));                    // Right And Down One Unit (Bottom Right)
                        glVertex2i(2*x,2*(y+1));                // Left And Down One Unit (Bottom Left)
                        glEnd();
                }
        }

};


void Room::collect() {                  // Collects the incomming energy

        for(int i=0; i<RX; i++)
        {
                for(int j=0; j<RY; j++)
                {
                        if(grid[i][j].getOut().getV1() > 0)
                                grid[i][j-1].setIn(Vector4(grid[i][j-1].getIn().getV1(), grid[i][j-1].getIn().getV2(),
grid[i][j].getOut().getV1(), grid[i][j-1].getIn().getV4()));

                        if(grid[i][j].getOut().getV2() > 0)
                                grid[i-1][j].setIn(Vector4(grid[i-1][j].getIn().getV1(), grid[i-1][j].getIn().getV2(), grid[i-
1][j].getIn().getV3(), grid[i][j].getOut().getV2()));

                        if(grid[i][j].getOut().getV3() > 0)
                                grid[i][j+1].setIn(Vector4(grid[i][j].getOut().getV3(), grid[i][j+1].getIn().getV2(),
grid[i][j+1].getIn().getV3(), grid[i][j+1].getIn().getV4()));

                        if(grid[i][j].getOut().getV4() > 0)
                                grid[i+1][j].setIn(Vector4(grid[i+1][j].getIn().getV1(), grid[i][j].getOut().getV4(),
grid[i+1][j].getIn().getV3(), grid[i+1][j].getIn().getV4()));


                        grid[i][j].setOut(Vector4(0.0f, 0.0f, 0.0f, 0.0f));



                }
        }


};

void Room::scatterNodes() {             // Loops through the nodes and calls scatter() in the node class

        for(int i=0; i<RX; i++)
        {
                for(int j=0; j<RY; j++)
                {
                        grid[i][j].scatter();
                }
        }

};
```

```
	void Room::setColors() {			// Determines the correct color depending on the total energy in a node

		for(int i=0; i<RX; i++)
		{
			for(int j=0; j<RY; j++)
			{
				float sum = grid[i][j].sumIn();
				grid[i][j].setR(0.3f + sum);
				grid[i][j].setG(0.3f + sum);
				grid[i][j].setB(0.3f + sum);

			}
		}

	};

	void Room::addSource(Source t, float time)	// Adds the energy from a source to the receiving node
	{
		grid[t.getX()][t.getY()].setIn(Vector4(grid[t.getX()][t.getY()].getIn().getV1() + (t.getAmp()*sinf(t.getFreq()*time))/4,
grid[t.getX()][t.getY()].getIn().getV2() + (t.getAmp()*sinf(t.getFreq()*time))/4, grid[t.getX()][t.getY()].getIn().getV3() +
(t.getAmp()*sinf(t.getFreq()*time))/4, grid[t.getX()][t.getY()].getIn().getV4() + (t.getAmp()*sinf(t.getFreq()*time))/4 ));

	};


	void Room::setWallUp()				// Increases the wall coefficient
	{
		for(int i=0; i<RX; i++)
		{
			for(int j=0; j<RY; j++)
				{
				if(grid[i][j].getW()>0 && grid[i][j].getW()<1)
					grid[i][j].setW(grid[i][j].getW()+0.1f);
				}
		}
	};

	void Room::setWallDown()			// Decreases the wall coefficient
	{
		for(int i=0; i<RX; i++)
		{
			for(int j=0; j<RY; j++)
				{
				if(grid[i][j].getW()>0.1f)
					grid[i][j].setW(grid[i][j].getW()-0.1f);
				}
		}
	};

	void Room::microphone(int x, int y, Source t)	// Write the results of the microphone to a textfile
	{

		ofstream debug("microphone.txt", ios::app);

		//debug << getTime() << ":" << grid[x][y].sumIn() << "\n";

		debug << getTime() << ":" << t.getAmp()*sinf(t.getFreq()*time)/4 << "\n";

	   debug.close();

	};

	void Room::calculateFrameRate()			// Calculates the frame rate
	{

	static float framesPerSecond    = 0.0f;		// This will store our fps
   static float lastTime			= 0.0f;	// This will hold the time from the last frame

   float currentTime = GetTickCount() * 0.001f;

   ++framesPerSecond;

   if( currentTime - lastTime > 1.0f )
		{
```

```
            lastTime = currentTime;
                fps = framesPerSecond;
        framesPerSecond = 0;

                }
        };

        void Room::drawInfo(Source s) {                 // Draws the info on the screen

                glColor3f(1.0f, 1.0f, 1.0f);
                glTranslatef(0.0f,0.0f,-1.0f);

                glRasterPos2f(420.0f, 290.0f);
                glPrint("FPS %7.2f", fps);              // Print GL Text To The Screen

                glRasterPos2f(420.0f, 270.0f);
                glPrint("Time %7.2f", time);

                glRasterPos2f(420.0f, 250.0f);
                glPrint("Walls %7.2f", grid[0][0].getW());  // Print GL Text To The Screen

                glRasterPos2f(420.0f, 230.0f);
                glPrint("w %7.2f", s.getFreq());        // Print GL Text To The Screen

                glRasterPos2f(420.0f, 210.0f);
                glPrint("Amp %7.2f", s.getAmp());       // Print GL Text To The Screen
        };


private:
        float c, fps;           // Speed of sound
        float time;                     // The total running time
        Node grid[RX][RY];      // Grid of nodes
};



Room r(340.0f, 1.0f, 20.0f);    // Creates a room
Source s(40, 75, 8, 5);         // Adds a Source


LRESULT   CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);       // Declaration For WndProc


GLvoid ReSizeGLScene(GLsizei width, GLsizei height)             // Resize And Initialize The GL Window
{
        if (height==0)                                  // Prevent A Divide By Zero By
        {
                height=1;                               // Making Height Equal One
        }

        glViewport(0,0,width,height);                   // Reset The Current Viewport

        glMatrixMode(GL_PROJECTION);                            // Select The Projection Matrix
        glLoadIdentity();                               // Reset The Projection Matrix

        gluOrtho2D(0.0,(GLdouble) 540,0.0,(GLdouble) 320);

        glMatrixMode(GL_MODELVIEW);                             // Select The Modelview Matrix
        glLoadIdentity();                               // Reset The Modelview Matrix
}


int InitGL(GLvoid)                                      // All Setup For OpenGL Goes Here
{
        glShadeModel(GL_SMOOTH);                                // Enable Smooth Shading
        glClearColor(0.5f, 0.5f, 0.5f, 0.5f);           // Black Background

        BuildFont();

        return TRUE;
}
```

```
int DrawGLScene(GLvoid)                                         // Here's Where We Do All The Drawing
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);     // Clear Screen And Depth Buffer

        glLoadIdentity();                                       // Resets the view matrix
        glTranslatef(10.0f, 10.0f, 0.0f);                       // Moves the view

        r.setTime(r.getTime() + 0.1f);                  // Increases the time
        r.addSource(s, r.getTime());                            // Adds the source

        r.scatterNodes();                                       // Scatters the pressure
        r.collect();                                    // Collects the pressure
        r.setColors();                                          // Set the color

        //r.microphone(49, 50, s);                              // Add a microphone at a node

        r.drawRoom();                                           // Draw the room

        r.calculateFrameRate();                                 // Calculate the frame rate
        r.drawInfo(s);                                          // Draw the info on the screen

        return TRUE;                                                                    // Keep Going
}



        //STANDARD WINDOW CODE, not implemented in this project.

GLvoid KillGLWindow(GLvoid)                                             // Properly Kill The Window
{
        if (fullscreen)                                         // Are We In Fullscreen Mode?
        {
                ChangeDisplaySettings(NULL,0);                  // If So Switch Back To The Desktop
                ShowCursor(TRUE);                                       // Show Mouse Pointer
        }

        if (hRC)                                                // Do We Have A Rendering Context?
        {
                if (!wglMakeCurrent(NULL,NULL))                 // Are We Able To Release The DC And RC Contexts?
                {
                        MessageBox(NULL,"Release Of DC And RC Failed.","SHUTDOWN ERROR",MB_OK |
MB_ICONINFORMATION);
                }

                if (!wglDeleteContext(hRC))                     // Are We Able To Delete The RC?
                {
                        MessageBox(NULL,"Release Rendering Context Failed.","SHUTDOWN ERROR",MB_OK |
MB_ICONINFORMATION);
                }
                hRC=NULL;                                               // Set RC To NULL
        }

        if (hDC && !ReleaseDC(hWnd,hDC))                        // Are We Able To Release The DC
        {
                MessageBox(NULL,"Release Device Context Failed.","SHUTDOWN ERROR",MB_OK | MB_ICONINFORMATION);
                hDC=NULL;                                                               // Set DC To NULL
        }

        if (hWnd && !DestroyWindow(hWnd))                       // Are We Able To Destroy The Window?
        {
                MessageBox(NULL,"Could Not Release hWnd.","SHUTDOWN ERROR",MB_OK | MB_ICONINFORMATION);
                hWnd=NULL;                                              // Set hWnd To NULL
        }

        if (!UnregisterClass("OpenGL",hInstance))               // Are We Able To Unregister Class
        {
                MessageBox(NULL,"Could Not Unregister Class.","SHUTDOWN ERROR",MB_OK | MB_ICONINFORMATION);
                hInstance=NULL;                                 // Set hInstance To NULL
        }
        KillFont();
```

```
}

/*      This Code Creates Our OpenGL Window.  Parameters Are:                   *
 *      title                   - Title To Appear At The Top Of The Window      *
 *      width                   - Width Of The GL Window Or Fullscreen Mode     *
 *      height                  - Height Of The GL Window Or Fullscreen Mode    *
 *      bits                    - Number Of Bits To Use For Color (8/16/24/32)  *
 *      fullscreenflag  - Use Fullscreen Mode (TRUE) Or Windowed Mode (FALSE)   */

BOOL CreateGLWindow(char* title, int width, int height, int bits, bool fullscreenflag)
{
        GLuint          PixelFormat;            // Holds The Results After Searching For A Match
        WNDCLASS        wc;                     // Windows Class Structure
        DWORD           dwExStyle;              // Window Extended Style
        DWORD           dwStyle;                // Window Style
        RECT            WindowRect;                     // Grabs Rectangle Upper Left / Lower Right Values
        WindowRect.left=(long)0;                // Set Left Value To 0
        WindowRect.right=(long)width;           // Set Right Value To Requested Width
        WindowRect.top=(long)0;                 // Set Top Value To 0
        WindowRect.bottom=(long)height;         // Set Bottom Value To Requested Height

        fullscreen=fullscreenflag;              // Set The Global Fullscreen Flag

        hInstance               = GetModuleHandle(NULL);                        // Grab An Instance For Our Window
        wc.style                = CS_HREDRAW | CS_VREDRAW | CS_OWNDC;   // Redraw On Size, And Own DC For
Window.
        wc.lpfnWndProc          = (WNDPROC) WndProc;                            // WndProc Handles Messages
        wc.cbClsExtra           = 0;                                            // No Extra Window Data
        wc.cbWndExtra           = 0;                                            // No Extra Window Data
        wc.hInstance            = hInstance;                                    // Set The Instance
        wc.hIcon                = LoadIcon(NULL, IDI_WINLOGO);                  // Load The Default Icon
        wc.hCursor              = LoadCursor(NULL, IDC_ARROW);                  // Load The Arrow Pointer
        wc.hbrBackground        = NULL;                                         // No Background Required For GL
        wc.lpszMenuName                 = NULL;                                 // We Don't Want A Menu
        wc.lpszClassName        = "OpenGL";                                             // Set The Class Name

        if (!RegisterClass(&wc))                                        // Attempt To Register The Window Class
        {
                MessageBox(NULL,"Failed To Register The Window Class.","ERROR",MB_OK|MB_ICONEXCLAMATION);
                return FALSE;                                           // Return FALSE
        }

        if (fullscreen)                                                 // Attempt Fullscreen Mode?
        {
                DEVMODE dmScreenSettings;                                       // Device Mode
                memset(&dmScreenSettings,0,sizeof(dmScreenSettings));           // Makes Sure Memory's Cleared
                dmScreenSettings.dmSize=sizeof(dmScreenSettings);               // Size Of The Devmode Structure
                dmScreenSettings.dmPelsWidth    = width;                        // Selected Screen Width
                dmScreenSettings.dmPelsHeight   = height;                       // Selected Screen Height
                dmScreenSettings.dmBitsPerPel   = bits;                         // Selected Bits Per Pixel
                dmScreenSettings.dmFields=DM_BITSPERPEL|DM_PELSWIDTH|DM_PELSHEIGHT;

                // Try To Set Selected Mode And Get Results.  NOTE: CDS_FULLSCREEN Gets Rid Of Start Bar.
                if (ChangeDisplaySettings(&dmScreenSettings,CDS_FULLSCREEN)!=DISP_CHANGE_SUCCESSFUL)
                {
                        // If The Mode Fails, Offer Two Options.  Quit Or Use Windowed Mode.
                        if (MessageBox(NULL,"The Requested Fullscreen Mode Is Not Supported By\nYour Video Card. Use
Windowed Mode Instead?","NeHe GL",MB_YESNO|MB_ICONEXCLAMATION)==IDYES)
                        {
                                fullscreen=FALSE;       // Windowed Mode Selected.  Fullscreen = FALSE
                        }
                        else
                        {
                                // Pop Up A Message Box Letting User Know The Program Is Closing.
                                MessageBox(NULL,"Program Will Now Close.","ERROR",MB_OK|MB_ICONSTOP);
                                return FALSE;                                   // Return FALSE
                        }
                }
        }

        if (fullscreen)                                                 // Are We Still In Fullscreen Mode?
        {
                dwExStyle=WS_EX_APPWINDOW;                                      // Window Extended Style
                dwStyle=WS_POPUP;                                               // Windows Style
```

```
		ShowCursor(FALSE);					// Hide Mouse Pointer
	}
	else
	{
		dwExStyle=WS_EX_APPWINDOW | WS_EX_WINDOWEDGE;		// Window Extended Style
		dwStyle=WS_OVERLAPPEDWINDOW;				// Windows Style
	}

	AdjustWindowRectEx(&WindowRect, dwStyle, FALSE, dwExStyle);	// Adjust Window To True Requested Size

	// Create The Window
	if (!(hWnd=CreateWindowEx( dwExStyle,				// Extended Style For The Window
					"OpenGL",			// Class Name
					title,				// Window Title
					dwStyle |			// Defined Window Style
					WS_CLIPSIBLINGS |		// Required Window Style
					WS_CLIPCHILDREN,		// Required Window Style
					0, 0,				// Window Position
					WindowRect.right-WindowRect.left,	// Calculate Window Width
					WindowRect.bottom-WindowRect.top,	// Calculate Window Height
					NULL,				// No Parent Window
					NULL,				// No Menu
					hInstance,			// Instance
					NULL)))				// Dont Pass Anything To
WM_CREATE
	{
		KillGLWindow();					// Reset The Display
		MessageBox(NULL,"Window Creation Error.","ERROR",MB_OK|MB_ICONEXCLAMATION);
		return FALSE;					// Return FALSE
	}

	static PIXELFORMATDESCRIPTOR pfd=		// pfd Tells Windows How We Want Things To Be
	{
		sizeof(PIXELFORMATDESCRIPTOR),		// Size Of This Pixel Format Descriptor
		1,					// Version Number
		PFD_DRAW_TO_WINDOW |			// Format Must Support Window
		PFD_SUPPORT_OPENGL |			// Format Must Support OpenGL
		PFD_DOUBLEBUFFER,			// Must Support Double Buffering
		PFD_TYPE_RGBA,				// Request An RGBA Format
		bits,					// Select Our Color Depth
		0, 0, 0, 0, 0, 0,			// Color Bits Ignored
		0,					// No Alpha Buffer
		0,					// Shift Bit Ignored
		0,					// No Accumulation Buffer
		0, 0, 0, 0,				// Accumulation Bits Ignored
		16,					// 16Bit Z-Buffer (Depth Buffer)
		0,					// No Stencil Buffer
		0,					// No Auxiliary Buffer
		PFD_MAIN_PLANE,				// Main Drawing Layer
		0,					// Reserved
		0, 0, 0					// Layer Masks Ignored
	};

	if (!(hDC=GetDC(hWnd)))				// Did We Get A Device Context?
	{
		KillGLWindow();				// Reset The Display
		MessageBox(NULL,"Can't Create A GL Device Context.","ERROR",MB_OK|MB_ICONEXCLAMATION);
		return FALSE;				// Return FALSE
	}

	if (!(PixelFormat=ChoosePixelFormat(hDC,&pfd)))	// Did Windows Find A Matching Pixel Format?
	{
		KillGLWindow();				// Reset The Display
		MessageBox(NULL,"Can't Find A Suitable PixelFormat.","ERROR",MB_OK|MB_ICONEXCLAMATION);
		return FALSE;				// Return FALSE
	}

	if(!SetPixelFormat(hDC,PixelFormat,&pfd))	// Are We Able To Set The Pixel Format?
	{
		KillGLWindow();				// Reset The Display
		MessageBox(NULL,"Can't Set The PixelFormat.","ERROR",MB_OK|MB_ICONEXCLAMATION);
		return FALSE;				// Return FALSE
	}
```

```
	if (!(hRC=wglCreateContext(hDC)))				// Are We Able To Get A Rendering Context?
	{
		KillGLWindow();						// Reset The Display
		MessageBox(NULL,"Can't Create A GL Rendering Context.","ERROR",MB_OK|MB_ICONEXCLAMATION);
		return FALSE;						// Return FALSE
	}

	if(!wglMakeCurrent(hDC,hRC))					// Try To Activate The Rendering Context
	{
		KillGLWindow();						// Reset The Display
		MessageBox(NULL,"Can't Activate The GL Rendering Context.","ERROR",MB_OK|MB_ICONEXCLAMATION);
		return FALSE;						// Return FALSE
	}

	ShowWindow(hWnd,SW_SHOW);					// Show The Window
	SetForegroundWindow(hWnd);					// Slightly Higher Priority
	SetFocus(hWnd);						// Sets Keyboard Focus To The Window
	ReSizeGLScene(width, height);					// Set Up Our Perspective GL Screen

	if (!InitGL())						// Initialize Our Newly Created GL Window
	{
		KillGLWindow();						// Reset The Display
		MessageBox(NULL,"Initialization Failed.","ERROR",MB_OK|MB_ICONEXCLAMATION);
		return FALSE;						// Return FALSE
	}


	return TRUE;						// Success
}

LRESULT CALLBACK WndProc(	HWND	hWnd,				// Handle For This Window
				UINT	uMsg,				// Message For This Window
				WPARAM	wParam,			// Additional Message Information
				LPARAM	lParam)			// Additional Message Information
{
	switch (uMsg)						// Check For Windows Messages
	{
		case WM_ACTIVATE:					// Watch For Window Activate Message
		{
			if (!HIWORD(wParam))			// Check Minimization State
			{
				active=TRUE;			// Program Is Active
			}
			else
			{
				active=FALSE;			// Program Is No Longer Active
			}

			return 0;				// Return To The Message Loop
		}

		case WM_SYSCOMMAND:					// Intercept System Commands
		{
			switch (wParam)				// Check System Calls
			{
				case SC_SCREENSAVE:			// Screensaver Trying To Start?
				case SC_MONITORPOWER:			// Monitor Trying To Enter Powersave?
				return 0;				// Prevent From Happening
			}
			break;					// Exit
		}

		case WM_CLOSE:					// Did We Receive A Close Message?
		{
			PostQuitMessage(0);			// Send A Quit Message
			return 0;				// Jump Back
		}

		case WM_KEYDOWN:					// Is A Key Being Held Down?
		{
			keys[wParam] = TRUE;			// If So, Mark It As TRUE
			return 0;				// Jump Back
		}
```

```
            case WM_KEYUP:                                  // Has A Key Been Released?
            {
                keys[wParam] = FALSE;                       // If So, Mark It As FALSE
                return 0;                                   // Jump Back
            }

            case WM_SIZE:                                   // Resize The OpenGL Window
            {
                ReSizeGLScene(LOWORD(lParam),HIWORD(lParam));  // LoWord=Width, HiWord=Height
                return 0;                                   // Jump Back
            }
        }

        // Pass All Unhandled Messages To DefWindowProc
        return DefWindowProc(hWnd,uMsg,wParam,lParam);
}

int WINAPI WinMain(    HINSTANCE       hInstance,           // Instance
                       HINSTANCE       hPrevInstance,       // Previous Instance
                       LPSTR           lpCmdLine,           // Command Line Parameters
                       int             nCmdShow)            // Window Show State
{
        MSG     msg;                                        // Windows Message Structure
        BOOL    done=FALSE;                                 // Bool Variable To Exit Loop

        // Ask The User Which Screen Mode They Prefer
        //if (MessageBox(NULL,"Would You Like To Run In Fullscreen Mode?", "Start
FullScreen?",MB_YESNO|MB_ICONQUESTION)==IDNO)
        //{
        //      fullscreen=FALSE;                           // Windowed Mode
        //}

        // Create Our OpenGL Window
        if (!CreateGLWindow("Room Acoustics",540,320,16,fullscreen))
        {
            return 0;                                       // Quit If Window Was Not Created
        }

        while(!done)                                        // Loop That Runs While done=FALSE
        {
            if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))       // Is There A Message Waiting?
            {
                if (msg.message==WM_QUIT)                   // Have We Received A Quit Message?
                {
                    done=TRUE;                              // If So done=TRUE
                }
                else                                        // If Not, Deal With Window Messages
                {
                    TranslateMessage(&msg);                 // Translate The Message
                    DispatchMessage(&msg);                  // Dispatch The Message
                }
            }
            else                                            // If There Are No Messages
            {
                // Draw The Scene.  Watch For ESC Key And Quit Messages From DrawGLScene()
                if (active)                                 // Program Active?
                {
                    if (keys[VK_ESCAPE])                    // Was ESC Pressed?
                    {
                        done=TRUE;                          // ESC Signalled A Quit
                    }
                    else                                    // Not Time To Quit, Update Screen
                    {
                        DrawGLScene();                      // Draw The Scene
                        SwapBuffers(hDC);                   // Swap Buffers (Double Buffering)
                    }

                }

                if (keys[VK_F1])                        // Is F1 Being Pressed?
                {
                    keys[VK_F1]=FALSE;                  // If So Make Key FALSE
                    KillGLWindow();                     // Kill Our Current Window
                    fullscreen=!fullscreen;             // Toggle Fullscreen / Windowed Mode
```

```
                        // Recreate Our OpenGL Window ( Modified )
                        if (!CreateGLWindow("Room Acoustics",420,320,16,fullscreen))
                        {
                                return 0;                       // Quit If Window Was Not Created
                        }
                }

                if (keys[VK_UP] && !bKeyUp)
                {
                        r.setWallUp();
                }
                if (!keys[VK_UP]) bKeyUp=false;


                if (keys[VK_DOWN] && !bKeyDown)
                {
                        r.setWallDown();
                }
                if (!keys[VK_DOWN])    bKeyDown=false;



        }
    }

    // Shutdown
    KillGLWindow();                                     // Kill The Window
    return (msg.wParam);                                // Exit The Program
}
```