The Mechanics of Charged Particles

Original © 15/07/2005
Updated © 19/05/2008
Posted on ViXra © 21/04/2013

The following notes are an attempt to simply and understand how charged particles work in terms of waves of energy:

An electron is the simplest of the charged particles and can be thought of in the following way:

The electron can be thought of as a standing wave comprised of either:

(a) An inward wave spiraling slightly into the centre and then being radiated directly out from the centre in a straight line as the outward wave.
(b) An inward wave flowing directly in towards the and then spiraling slightly as it flows outwards as the outward wave.

The other solution would represent a positron.

Assuming (a) is the correct model for an electron, then the following description applies:

The slight spiral is due to the slight slowing of the propagation speed of the wave as the energy density in space increases (the electron's gravitational potential), and the path direction of the wave being slightly off from pointing directly towards/away from the electron's centre. This spiralling then accounts for the ½ intrinsic angular momentum (spin) of the electron.

The standing wave thus formed when the inward and outward waves are summed has nodes that will be moving inwards or outwards and rotating around the electron's centre.

The inward waves are reflected off the standing wave nodes that are rotating and change direction to form the outward waves. Due to the path length difference between the inward and outward waves (as one is spiralling and the other is flowing in a straight line) there will be a "flow of phase" inwards with respect to the electron's centre.

Looking at a cross section through an electron, there would appear to be a slightly higher frequency wave moving at a slightly slower speed moving inwards, summed with a slightly lower frequency wave moving higher speed moving outwards. At the nodes, where the inward wave becomes the outward wave & vice versa, there would be a Doppler shift of the frequency & a change of direction.

There is an energy balance between inward and outward waves (i.e. inflow = outflow, resulting in no net energy loss/gain). The frequencies, amplitudes & speeds of the waves would be set such that the energies of the inward & outward waves are identical & each can become the other when Doppler shifted at the nodes.

There exists, at any point in space, a phase drift inwards or outwards with respect to the electron's centre. This phase drift causes the attraction/repulsion associated with the electric field when it interacts with other charged particles. As one moves further from the electron's centre, the amplitude of the waves decreases, so too does the electric field and its associated force due to these waves (i.e. lower wave amplitude = lower momentum in waves = lower force on other charged particles).

When the electric field of the electron interacts with other charged particles (for example another identical electron), the inward and outward waves of each electron overlap. When this happens, at the interface between the two electrons (exactly equal distances between each electron's centre) the two outward waves will form a standing wave and the two inward waves will form a standing wave – each of these standing waves will have no phase drift as an equal amount of each frequency is coming from the other electron. Thus the nodes of the standing wave at this point will not be moving.

The outward wave of the electron normally forms the inward wave when it reflects off a moving node (causing the wave number change between the outward and inward waves), but at this midway point the node is not moving, and so the outward wave will be reflected to form an inward wave of higher frequency than usual (Doppler shifted). Similarly, the nodes from the electron's centre to the midway point become progressively slower, until at the midway point the node stops completely. Thus the interface between two electrons provides a frequency conversion, or momentum change to the inward/outward waves. These frequency/momentum changes will propagate through each node to the electron's centre causing the whole electron to move – thus the electron has been accelerated to a certain velocity.

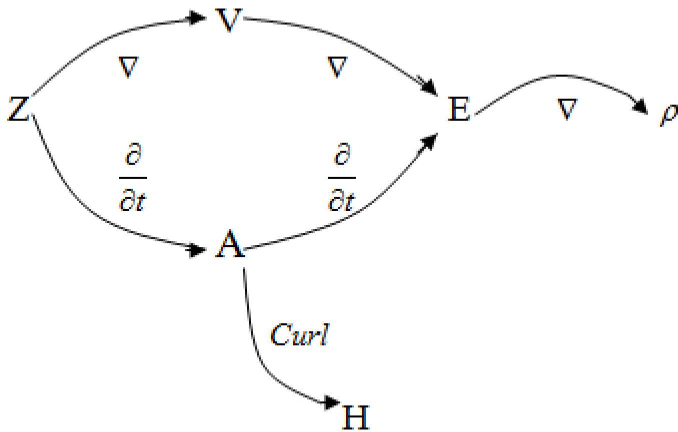
Update 2013:

I testing my ideas back in 1998 my building a model of Maxwell's Field equations for Electric/Magnetic fields in a 3D grid of vectors - the idea being that a stable particle (3D waveform) should spontaneously form if the E/M euations are modelled correctly in 3D & with the right sort of starting conditions inputted into the model. The results of this modelling can be seen in Appendix A & B. The success of this modelling led to my second Analytical model (by modifying the original model code to use the set of equations (see Towards a Unified Field Theory) for an Electron & Appendices C & D).

Towards a Unified Field Theory

© 20/7/02 Declan Traill

All of Electromagnetics can be explained in terms of a field of vectors called the Hertzian vector field. The relationships between each of the measurable quantities used in Electromagnetic Theory can be expressed quite simply by the following diagram:



Where:

Z = Hertzian Vector
V = Voltage (electric potential)
E = Electric Field Vector
A = Vector potential
H = Magnetic Field Vector
ρ = Charge density

The actual equations connecting these quantities are:

$A = -\frac{1}{c}\frac{\partial Z}{\partial t}$ (1) $V = \nabla . Z$ (2) $E = -\nabla V - \frac{1}{c}\frac{\partial A}{\partial t}$ (3)

$H = \nabla \times A$ (4) $\rho = -\frac{1}{4\pi}\nabla . E$ (5)

This set of relationships must form part of a unified field theory if one is to be found. It is a fragment of the truth that needs to be placed in its correct context.

So what is the Hertzian vector? How can it be understood or explained in terms of something more fundamental? What are the possibilities? :

(1) As it is a vector, it is a time or space gradient (or both) of some more fundamental field.

(2) It should not be a vector at all, but could equally well be a scalar quantity. The other quantities in the above diagram could be derived equally well from a scalar field than a vector field.

Charge is a physical property that we are more familiar with, and is derived from the Hertzian vector. It comes in two varieties: positive and negative. Perhaps the question we should ask is whether charge is an intrinsic physical property that cannot be explained any further, or can it be explained as a property of some other field?

If the Hertzian vector was replaced with a scalar quantity, then charge could be explained as high/low pressure zones (to use an analogy) in this field much like high and low pressure systems in the atmosphere. This is probably the simplest approach to explaining charge and getting to the bottom of what the Hertzian field is about.

The alternate approach is to derive the Hertzian vector from either a time gradient (like the Vector Potential) or a space gradient (like the voltage) or a combination of both (like the Electric field). This approach is more complicated with no obvious answer (with the information currently available to me). I shall not attempt to offer any suggestions here, other than to raise it as a possibility.

One other possibility exists that I have not explored: that the Hertzian vector is nothing but a mathematical construct and has no Physical meaning at all. This is a possibility and should not be discounted, but if one is attempting to simplify Physics in order to discover a Unified Field Theory, then one must assume that there is a bedrock of reality from which all other Physical properties are derived. This type of assumption has worked many times in the past in Science and turned out to be correct, so we should feel confident in applying this approach to this situation.

Update 2012/13

It appears, upon further analysis, that the Hertzian Vector is a measure of the flow of Phase and for a particle such as an Electron its scalar amplitude must vary with r (i.e. from the centre of the particle outwards) in an amount proportional to ln(r) - natural Log based on the distance r.

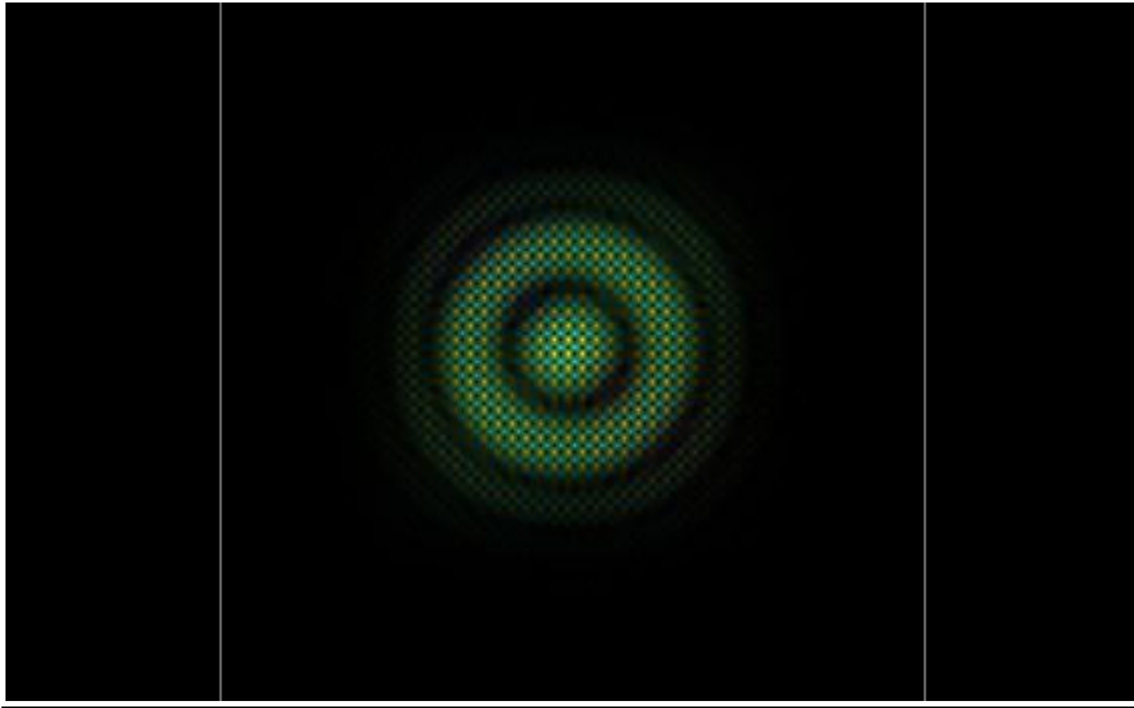
So for an electron/positron where the inward/outward waves (that comprise the particle's spinning spiral wave function) have slightly different frequencies and amplitudes, the Hertzian vector is the vector difference of these two phase/amplitude flows.

Thus, using the above equations (and my computer modeling [Appendix C, D] of them) all of the Electromagnetic fields can be derived from the fundamental underlying wave function.

References

Fundamentals of Electric Waves (1942)
H.H. Skilling. Wiley. NY p127-140

Appendix A: My first Electron Model: a finite element model of Electric & Magnetic Fields using Maxwells Field equations.



Appendix B: My Delphi Source Code for Images generated in Appendix C

```
//  VectPotential - 3D wave modelling platform.
//
//  (c) Copyright 1998 - 2011 : Declan Traill
//

unit VectorPotential;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  ExtCtrls, StdCtrls, Spin, ComCtrls, Math, Grids, Jpeg;

type            {Type declaration for a vector resolved into x,y & z
components}
  Vector = record
    x: extended;
    y: extended;
    z: extended;
  end;

  VectorGrp = record
    V0: Vector;
    V1: Vector;
    V2: Vector;
    V3: Vector;
    V4: Vector;
    V5: Vector;
    V6: Vector;
  end;

  ScalarGrp = record
    S0: extended;
    S1: extended;
    S2: extended;
    S3: extended;
    S4: extended;
    S5: extended;
    S6: extended;
  end;

  point = record      {Type declaration for a grid point}
    Electric: Vector;
    Magnetic: Vector;
  end;
```

```
  PointGrp = record
    P0: point;
    P1: point;
    P2: point;
    P3: point;
    P4: point;
    P5: point;
    P6: point;
  end;

  PointPtr = ^point;     {Define a type: pointer to a grid point}
  BitmapPtr = ^TBitmap;  {Define a type: pointer to a bitmap}
  TColorPtr = ^TColor;   {Define a type: pointer to a 24 bit colour
type (TColor)}

  EdgeMasks = array[1..9,1..3,1..3] of byte;  {Define the EdgeMasks
array type}

  TForm1 = class(TForm)
    StartGroup: TGroupBox;
    Start1: TRadioButton;
    Start2: TRadioButton;
    start3: TRadioButton;
    Start4: TRadioButton;
    Start5: TRadioButton;
    Start6: TRadioButton;
    Start7: TRadioButton;
    Start8: TRadioButton;
    Start9: TRadioButton;
    ZPlaneGroup: TGroupBox;
    ZPlane: TTrackBar;
    Z_Plane_Number: TEdit;
    MainGroup: TGroupBox;
    TimeDisplay: TEdit;
    ReScale: TSpinEdit;
    AmpDisplay: TEdit;
    TimeFreeze: TButton;
    Image1: TImage;
    DisplayLevel: TScrollBar;
    FieldGroup: TGroupBox;
    Field1: TRadioButton;
    Field2: TRadioButton;
    Field3: TRadioButton;
    Field4: TRadioButton;
    Field5: TRadioButton;
    StatsGroup: TGroupBox;
    Energy1: TEdit;
    Energy2: TEdit;
    Energy3: TEdit;
    Units1: TEdit;
    Units2: TEdit;
    Units3: TEdit;
    Energy_Msg1: TEdit;
    Energy_Msg2: TEdit;
    Energy_Msg3: TEdit;
```

```
ColourGroup: TGroupBox;
ColourX_Group: TGroupBox;
ColourY_Group: TGroupBox;
ColourZ_Group: TGroupBox;
X_Red: TRadioButton;
X_Green: TRadioButton;
X_Blue: TRadioButton;
Y_Red: TRadioButton;
Y_Green: TRadioButton;
Y_Blue: TRadioButton;
Z_Red: TRadioButton;
Z_Green: TRadioButton;
Z_Blue: TRadioButton;
X_Colour: TImage;
Y_Colour: TImage;
Z_Colour: TImage;
DisplayOptionsGroup: TGroupBox;
RendDisplay: TCheckBox;
Vect_Arrows: TCheckBox;
Vector_Group: TGroupBox;
VectorX: TCheckBox;
VectorY: TCheckBox;
VectorZ: TCheckBox;
Spacing_Text: TStaticText;
Spacing_metres: TRadioButton;
Spacing_pixels: TRadioButton;
Bevel1: TBevel;
FirstZ: TStaticText;
LastZ: TStaticText;
Z_Tiling: TButton;
TileGrid: TDrawGrid;
GridGroup: TGroupBox;
GridX: TEdit;
GridY: TEdit;
GridZ: TEdit;
GridXlabel: TLabel;
GridYlabel: TLabel;
GridZlabel: TLabel;
RendGroup: TGroupBox;
AspectControl: TCheckBox;
AcceptGridSize: TButton;
RendColour: TGroupBox;
GreyscaleButton: TRadioButton;
ColourButton: TRadioButton;
X_none: TRadioButton;
Y_none: TRadioButton;
Z_none: TRadioButton;
AutoWarnTimer: TTimer;
Timesteps: TSpinEdit;
Label1: TLabel;
RateOfTime: TScrollBar;
Label2: TLabel;
GroupBox1: TGroupBox;
GroupBox2: TGroupBox;
CheckBox1: TCheckBox;
```

```
CheckBox2: TCheckBox;
Spacing_gridpoints: TRadioButton;
Button1: TButton;
VectorSpacing: TEdit;
RenderOption1: TRadioButton;
RenderOption2: TRadioButton;
RenderOption3: TRadioButton;
ArrowScaleScroll: TScrollBar;
Label3: TLabel;
ActualGridWidth: TEdit;
Label4: TLabel;
Label5: TLabel;
Start10: TRadioButton;
Button2: TButton;
ViewFromTop: TCheckBox;
AutoScaleGroup: TGroupBox;
AutoWarn: TImage;
Auto1: TRadioButton;
Auto2: TRadioButton;
Auto3: TRadioButton;
Scale_3D: TCheckBox;
procedure FormCreate(Sender: TObject);
procedure Start1Click(Sender: TObject);
procedure Start2Click(Sender: TObject);
procedure Field1Click(Sender: TObject);
procedure Field2Click(Sender: TObject);
procedure start3Click(Sender: TObject);
procedure Start4Click(Sender: TObject);
procedure Start5Click(Sender: TObject);
procedure Start6Click(Sender: TObject);
procedure Field3Click(Sender: TObject);
procedure Field4Click(Sender: TObject);
procedure Field5Click(Sender: TObject);
procedure TimeFreezeClick(Sender: TObject);
procedure Start7Click(Sender: TObject);
procedure ZPlaneChange(Sender: TObject);
procedure DisplayLevelChange(Sender: TObject);
procedure ReScaleChange(Sender: TObject);
procedure Start8Click(Sender: TObject);
procedure Start9Click(Sender: TObject);
procedure Start10Click(Sender: TObject);
procedure X_RedClick(Sender: TObject);
procedure X_GreenClick(Sender: TObject);
procedure X_BlueClick(Sender: TObject);
procedure Y_RedClick(Sender: TObject);
procedure Y_GreenClick(Sender: TObject);
procedure Y_BlueClick(Sender: TObject);
procedure Z_RedClick(Sender: TObject);
procedure Z_GreenClick(Sender: TObject);
procedure Z_BlueClick(Sender: TObject);
procedure GreyscaleButtonClick(Sender: TObject);
procedure ColourButtonClick(Sender: TObject);
procedure Auto1Click(Sender: TObject);
procedure Auto2Click(Sender: TObject);
procedure Auto3Click(Sender: TObject);
```

```
    procedure Vect_ArrowsClick(Sender: TObject);
    procedure VectorSpacingChange(Sender: TObject);
    procedure Z_TilingClick(Sender: TObject);
    procedure Image1Click(Sender: TObject);
    procedure Image1DblClick(Sender: TObject);
    procedure VectorXClick(Sender: TObject);
    procedure VectorYClick(Sender: TObject);
    procedure VectorZClick(Sender: TObject);
    procedure Spacing_pixelsClick(Sender: TObject);
    procedure Spacing_metresClick(Sender: TObject);
    procedure Spacing_gridpointsClick(Sender: TObject);
    procedure RenderOption1Click(Sender: TObject);
    procedure RenderOption2Click(Sender: TObject);
    procedure RenderOption3Click(Sender: TObject);
    procedure RendDisplayClick(Sender: TObject);
    procedure AspectControlClick(Sender: TObject);
    procedure GridXChange(Sender: TObject);
    procedure GridYChange(Sender: TObject);
    procedure GridZChange(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure AcceptGridSizeClick(Sender: TObject);
    procedure X_noneClick(Sender: TObject);
    procedure Y_noneClick(Sender: TObject);
    procedure Z_noneClick(Sender: TObject);
    procedure AutoWarnTimerTimer(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure RateOfTimeChange(Sender: TObject);
    procedure CheckBox1Click(Sender: TObject);
    procedure CheckBox2Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure ArrowScaleScrollChange(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure ViewFromTopClick(Sender: TObject);
    procedure Scale_3DClick(Sender: TObject);
  private
    { Private declarations }
  public
    procedure Propagate;
    procedure Initialise(first:boolean);
    procedure RecalcElectric(scr:smallint);
    procedure UpdateBitmap(scr:smallint);
    procedure DisplayScreen(scr:smallint);
    procedure InputFields(scr:smallint;Time:extended);
    function sign(number: extended): shortint;
    function Gradient(val1, val2: extended): extended;
    function Gradient_3point(val1, val2, val3: extended): extended;
    procedure RecalcMagnetic(scr: smallint);
    function Gauss(x: extended): extended;
    procedure UpdateDetails;
    procedure UpdateAxisColours(which: string);
    procedure Auto_Scale(scr: smallint);
    procedure MaxCheck(element: PointPtr);
    function VectSize(vect: Vector): extended;
    function E_Energy(E_amp: extended): extended;
    function B_Energy(B_amp: extended): extended;
```

```
    procedure DrawArrow(ThisBitmap: Tbitmap;x,y: smallint; arrow:
Vector);
    procedure NewBitmap(BmapPtr: BitmapPtr);
    procedure PlotPoint(Bmap: TBitmap; x, y: smallint);
    function RGB_Val(colour: Tcolor; primary: integer): byte;
    function EdgeCase(x, y, Xmin, Ymin, Xmax, Ymax: smallint): byte;
    function VectorInterpolate(v1, v2, v3, v4: Vector; Xfrac,
      Yfrac: extended): Vector;
    function Interpolate(val1, val2, val3, val4, Xfrac,
      Yfrac: extended): extended;
    procedure PlotQuadrant(Bmap: TBitmap; Quadrant: byte; RealX,
RealY: extended);
    function GetActualX(x: smallint): smallint;
    function GetActualY(y: smallint): smallint;
    procedure SetTileSize;
    function Round2(realval: extended): int64;
    function GetRealX(x: extended): extended;
    function GetRealY(y: extended): extended;
    function ColourRange(value: extended; ScaleFactor: extended):
byte;
    function VectToColours(vect: vector; ScaleFactor: extended):
vector;
    function ByteLimit(value: extended): smallint;
    function VectByteLimit(vect: Vector): Vector;
    function VectorCross(v1, v2: Vector): Vector;
    function PowerFlow(Apoint: point): vector;
    function VectorProperty(field: byte; Apoint: point): vector;
    function PointNull(Apoint: point): boolean;
    function VectorNull(vect: Vector): boolean;
    function ReverseTColor(input: TColor): Tcolor;
    function MouseZplane: smallint;
    procedure TileCursor(Bmap: TBitmap; Tile: smallint; colour:
TColor);
    procedure PlotPixel(x, y: smallint; Colour: TColor);
    procedure UpdateE_Energy(scr: smallint);
    procedure UpdateB_Energy(scr: smallint);
    procedure SetAspectRatio;
    procedure SetGridGlobals;
    procedure ReAllocGridMemory;
    function VectDiv(VectGroup: VectorGrp): Vector;
    function VectCurl(VectGroup: VectorGrp): Vector;
    function PointGroup(scr, x, y, z: smallint): PointGrp;
    function VectorGroup(PntGroup: PointGrp; Field: smallint):
VectorGrp;
    procedure FindMaxVal(scr, Field: smallint);
    function VectorDot(v1, v2: Vector): extended;
    function ScalarGrad(ScalarGroup: ScalarGrp): Vector;
    { Public declarations }
  end;
const
{EdgeArray is an array used to determine which points exist within the
range of}
{a rectangular region (such as a grid x/y plane) given a known edge
condition}
```

```
{The edge condition is determined elsewhere by EdgeCase; its values
range from}
{1 to 9 corresponding to the point in question residing at the topleft
corner,}
{somewhere along the top edge, the topright corner... etc.}
{The values in the sub-arrays indicate which of the points surrounding
the }
{point in question are in range: 1=in range, 0=out of range}

  EdgeArray : EdgeMasks = (((0,0,0),     {Top Left corner}
                            (0,1,1),
                            (0,1,1)),

                           ((0,0,0),     {Top Edge}
                            (1,1,1),
                            (1,1,1)),

                           ((0,0,0),     {Top Right Corner}
                            (1,1,0),
                            (1,1,0)),

                           ((0,1,1),     {Left Edge}
                            (0,1,1),
                            (0,1,1)),

                           ((1,1,1),     {Not on an Edge or Corner}
                            (1,1,1),
                            (1,1,1)),

                           ((1,1,0),     {Right Edge}
                            (1,1,0),
                            (1,1,0)),

                           ((0,1,1),     {Bottom Left Corner}
                            (0,1,1),
                            (0,0,0)),

                           ((1,1,1),     {Bottom Edge}
                            (1,1,1),
                            (0,0,0)),

                           ((1,1,0),     {Bottom Right Corner}
                            (1,1,0),
                            (0,0,0)));
const
  Default_GridWidth=100;     {Default Number of Pixels wide}
  Default_GridHeight=100;    {Default Number of Pixels high}
  Default_GridDepth=100;     {Default Number of Pixels Deep}
  Million=1000000; {One million (10^6) used for other definitions}
  SpeedOfLight=299.8*Million;   {Speed of Light in meters per second}
  Permittivity=8.854/1000000000000; {Permittivity of free space
(farad/m)}
  Permeability=4*Pi/(10*Million); {Permeability of free space
(Henry/m)}
  PPM=10000000000000000; {Points Per Metre}
```

```
  PointSize=1/PPM; {size, in metres, that each point represents}
  ElectronComptonWavelength=2.4263102175/1000000000000;
  WavePeriod1=400;  {Number of Time iterations for wave period}
  WaveNumber=1; {number of wave cycles to start with}
  E_Amplitude=1000*PointSize;    {Max amplitude of input E field - 
Volts/m }
  B_Amplitude=1000*PointSize;    {Max amplitude of input B field - 
Tesla  }
  Max_E=0;  {Max value of Electric field to allow per pixel - Volts/m 
(0 value disables it)}
  Max_B=0;  {Max value of Magnetic field to allow per pixel - Tesla (0 
value disables it)}
  RED=$1;      {Multiplication factor for 4-byte RGB colour}
  GREEN=$100;  {Multiplication factor for 4-byte RGB colour}
  BLUE=$10000; {Multiplication factor for 4-byte RGB colour}
  BLACK=$0;    {Definition for black}
  RedMask=$FF;        {Masking value for colour separation}
  GreenMask=$FF00;    {Masking value for colour separation}
  BlueMask=$FF0000;   {Masking value for colour separation}
  START=1;     {AutoScaling constants}
  CONTINUAL=2; {AutoScaling constants}
  NEVER=3;     {AutoScaling constants}
  OneToOne=1;  {Rendering Option Constant - for 1 pixel per point}
  Chunky=2;    {Rendering Option Constant - for filled rectangle per 
point}
  Blend=3;     {Rendering Option Constant - colour blending betweem 
points}
  EdgeSize=3;  {Size of border between Tiled Z Planes & screen edge}
var
  Form1: TForm1;
  Bitmap1,Bitmap2: TBitmap;
  BitmapRed,BitmapGreen,BitmapBlue,BitmapBlack: TBitmap;

  // Note: the Grid (0,0,0) point is at the Top, Left & Back point of 
the Grid
  //       So X increments to the right, Y increments downwards, & Z 
increments out of the screen
  points: array of array of array of array of point; 
{[1..2,1..GridWidth,1..GridHeight,1..GridDepth] of point;}

  ColourArray: array of array of TColor; {[1..GridWidth,1..GridHeight] 
of Tcolor;}
  SignArray: array of array of array of shortint; 
{[1..GridWidth,1..GridHeight,1..3] of shortint;}
  ColArray: array[1..9] of Tcolor;
  PntArray: array[1..9] of Vector;
  YLinePtrs: array of PByteArray;
  GridWidth,GridHeight,GridDepth: integer;
  New_GridWidth,New_GridHeight,New_GridDepth: integer;
  screen: smallint;
  StartOption,New_StartOption: smallint;
  New_DisplayField,DisplayField : smallint;
  Time,TimeStep : extended;
  RescaleFactor, New_ReScale : double;
  Z_Plane, New_ZPlane : smallint;
```

```
  FreezeTime, New_FreezeTime, GridTransformed: Boolean;
  TileZ, New_TileZ: Boolean;
  ShowColour, New_ShowColour: Boolean;
  IsGrey,DoUpdate : Boolean;
  E_Energy_Tot,B_Energy_Tot,MaxVal : extended;
  Amplification : extended;
  X_RGB,Y_RGB,Z_RGB : Tcolor;
  New_AutoScale,AutoScale: smallint;
  ZplanePos,OriginX,OriginY : smallint;
  FirstPass : boolean;
  ArrowScale: extended;
  ScrScaleX,ScrScaleY,HalfX,HalfY : extended;
  BitmapX,BitmapY: smallint;
  TileX,TileY,TileXcount,TileYcount : smallint;
  Aspect: extended;
  NullVect: Vector;
  NullPoint: Point;
  NullVectGrp: VectorGrp;
  NullPointGrp: PointGrp;
  MyMouse: TMouse;
  AxisColours,New_AxisColours: string;
  Display_Level,New_DisplayLevel: integer;
  Rate_Of_Time,New_RateOfTime: integer;
  Arrows, New_Arrows: boolean;
  UpdateColours,ReDraw: boolean;
  CurrentBitmap: TBitmap;
  New_Render,Render: smallint;
  New_Rendered,Rendered: boolean;
  New_VectSpacing,VectSpacing: integer;
  TileScrScaleX,TileScrScaleY: double;
  TileHalfX,TileHalfY: extended;
  MaintainAspect,New_MaintainAspect,VectorChange: boolean;
  ActualWidth,ActualHeight,ActualDepth: extended;
  PointArea,PointVolume: extended;
  Waveperiod2: extended;
  ScreenAspect: extended;
  dx,dy,dz: extended;
  ArrowsUnitsChange,AutoWarnState: boolean;
  TileRect: TRect;
  Quit: boolean;
  Timestep_count: smallint;
  FrameCount: integer;
  save_frames,save_3D,New_Flip_YZ,Flip_YZ: boolean;
  arrow_step: extended;
  New_ArrowScaleFactor: integer;
  ArrowScaleFactor: single;
  Restart: boolean;
  ViewTop: boolean;

implementation

{$R *.DFM}
{$G-} {Disable Data Importation from Units - Improves Memory access
efficiency}
```

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Initialise(true);
  Propagate();
end;

procedure TForm1.Initialise(first:boolean);
var
  scr,i,j,k : smallint;
begin
  if first then begin
    Restart:=true;
    FreezeTime:=false;
    New_FreezeTime:=false;
    GridTransformed:=false;
    StartOption:=0;                {No Starting conditions untill
selected}
    New_StartOption:=9;           {default to start config 10}
    MyMouse:=TMouse.Create;        {Create a mouse variable}
    BitmapX:=Image1.Width;         {get width of picture control on
the form}
    BitmapY:=Image1.Height;        {get height of picture control on
the form}
    ScreenAspect:=BitmapY/BitmapX; {Calc Screen's aspect ratio}
    SetLength(YLinePtrs,BitmapY);  {determine size for array of bitmap
line pointers}
    New_DisplayField:=1;     {set default to displaying the Electric
field}
    NewBitmap(@Bitmap1);     {create a new blank 24bit bitmap}
    NewBitmap(@Bitmap2);     {create a new blank 24bit bitmap}
    BitmapRed := TBitmap.Create;     {create a bitmap for the red
square}
    BitmapGreen := TBitmap.Create;   {create a bitmap for the green
square}
    BitmapBlue := TBitmap.Create;    {create a bitmap for the blue
square}
    BitmapBlack := TBitmap.Create;    {create a bitmap for the black
square}
    BitmapRed.LoadFromFile('red.bmp');      {load the red square from
disk}
    BitmapGreen.LoadFromFile('green.bmp');  {load the green square
from disk}
    BitmapBlue.LoadFromFile('blue.bmp');    {load the blue square from
disk}
    BitmapBlack.LoadFromFile('black.bmp');  {load the black square
from disk}

    New_ShowColour:=true;            {default to displaying a colour
image}
    AutoScale:=CONTINUAL;             {default autoscale option to at
Continual}
    New_TileZ:=false;                {default Z plane Tiling option}
    New_Arrows:=true;                {default to Vector Arrows On}
    New_Render:=Blend;               {default to Colour Blended
Rendering}
```

```
    New_Rendered:=true;              {default to Rendered Display On}
    New_DisplayLevel:=1900;          {default Level slider to 0%
(centre)}
    New_RateOfTime:=RateOfTime.Position; {default Rate of Time slider}
    New_MaintainAspect:=true;        {Default Aspect Ratio Control
setting}
    New_AutoScale:=CONTINUAL;        {Default AutoScale option to
Continual}
    GridWidth:=0;                    {Ensure a full update occurs}
    GridHeight:=0;
    GridDepth:=0;
    New_GridWidth:=Default_GridWidth;     {set default Grid
Dimensions}
    New_GridHeight:=Default_GridHeight;
    New_GridDepth:=Default_GridDepth;
    VectorChange:=false;              {initialise flag}
    FrameCount:=1;
    save_frames:=false;
    save_3D:=false;
    New_Flip_YZ:=false;
    Flip_YZ:=false;
    New_ArrowScaleFactor := ArrowScaleScroll.Position;
    ViewTop := ViewFromTop.Checked;
    dx:=PointSize;
    dy:=PointSize;
    dz:=PointSize;
  end;
  with NullVect do begin
    x:=0;
    y:=0;
    z:=0;
  end;
  with NullPoint do begin
    Electric:=NullVect;
    Magnetic:=NullVect;
  end;
  with NullVectGrp do begin
    v0:=NullVect;
    v1:=NullVect;
    v2:=NullVect;
    v3:=NullVect;
    v4:=NullVect;
    v5:=NullVect;
    v6:=NullVect;
  end;
  with NullPointGrp do begin
    p0:=NullPoint;
    p1:=NullPoint;
    p2:=NullPoint;
    p3:=NullPoint;
    p4:=NullPoint;
    p5:=NullPoint;
    p6:=NullPoint;
  end;
  Form1.visible:=true;       {show the Form (user interface)}
```

```
  Image1.visible:=true;      {ensure the Image is visible}
  Screen:=0;                 {start by displaying bitmap1}
  DisplayScreen(Screen);     {make it visible}
  Time:=0;                   {set time to zero}
  Timestep_count:=0;
  MaxVal:=0;                 {ensure MaxVal is zero}
  E_Energy_Tot:=0;           {ensure Electric field energy total=0}
  B_Energy_Tot:=0;           {ensure Magnetic field energy total=0}
  New_AxisColours:='ALL';    {set axis colour display boxes to match
selections}
  UpdateDetails;             {Update all changed values & displayed
text}
  DoUpdate:=false;           {Initial conditions set up via FirstPass}
  FirstPass:=true;

  if not first then          {if first initialisation then Grid
zero'ed already}
    for scr:=0 to 1 do             {scan both copies of the grid's
points}
      for i:=0 to GridWidth-1 do     {and set all values to zero}
        for j:=0 to GridHeight-1 do
          for k:=0 to GridDepth-1 do
            points[scr,i,j,k]:=NullPoint;
  ReDraw:=true;               {ensure redraw is enabled}
end;

procedure TForm1.RecalcMagnetic(scr: smallint);
var
  dBx_dt,dBy_dt,dBz_dt : extended;
  Current_Bx,Current_By,Current_Bz : extended;
  NewScreen : smallint;
  xpos,ypos,zpos : smallint;
  ThisGroup: PointGrp;
  CurlVect: Vector;
  VectGrp: VectorGrp;
begin
  if scr=0 then NewScreen:=1 else NewScreen:=0;  {determine which data
to update}
  for xpos:=0 to GridWidth-1 do begin            {scan grid's x
coords}
    for ypos:=0 to GridHeight-1 do begin         {scan grid's y
coords}
      for zpos:=0 to GridDepth-1 do begin        {scan grid's z
coords}

        ThisGroup:=PointGroup(scr, xpos, ypos, zpos);

        { ThisGroup's points are assigned as follows:    P3
P5
                                                      P1 P0 P2
                                                         P4        P6

                        Where P5 & P6 are in the Z plane }

        with ThisGroup do begin
```

```
          Current_Bx:=P0.Magnetic.x;        {current x component of
Magnetic field}
          Current_By:=P0.Magnetic.y;        {current y component of
Magnetic field}
          Current_Bz:=P0.Magnetic.z;        {current z component of
Magnetic field}
        end;

        if not Flip_YZ then begin
          VectGrp:=VectorGroup(ThisGroup, 1);
          CurlVect:=VectCurl(VectGrp);

          {Calculate required change in Magnetic field components
based on}
          {the observed gradients in the Electric field where the grid
point is}
          dBx_dt:=-CurlVect.x;    {change in x component of B}
          dBy_dt:=-CurlVect.y;    {change in y component of B}
          dBz_dt:=-CurlVect.z;    {change in z component of B}

          with points[NewScreen,xpos,ypos,zpos].Magnetic do begin
            x:=Current_Bx+(dBx_dt*TimeStep);  {current Bx +
Changerate*Timestep}
            y:=Current_By+(dBy_dt*TimeStep);  {current By +
Changerate*Timestep}
            z:=Current_Bz+(dBz_dt*TimeStep);  {current Bz +
Changerate*Timestep}
          end;

        //       MaxCheck(@points[NewScreen,xpos,ypos,zpos]); {limit
new E & H vals if req'd}
        end
        else begin

          {Save new Magnetic field component values in new grid -
swapping Y & Z planes}
          with points[NewScreen,xpos,zpos,ypos].Magnetic do begin
            x:=Current_Bx; {current Bx}
            y:=Current_Bz; {current Bz}
            z:=Current_By; {current By}
          end;
        end;

      end;
    end;
  end;

end;

procedure TForm1.RecalcElectric(scr:smallint);
var
  dEx_dt,dEy_dt,dEz_dt : extended;
  Current_Ex,Current_Ey,Current_Ez: extended;
  NewScreen : smallint;
  xpos,ypos,zpos:smallint;
```

```
  ThisGroup: PointGrp;
  vect,CurlVect,DivVect: vector;
  VectGrp: VectorGrp;
begin
  if scr=0 then NewScreen:=1 else NewScreen:=0;  {determine which data
to update}
  for xpos:=0 to GridWidth-1 do begin            {scan grid's x
coords}
    for ypos:=0 to GridHeight-1 do begin         {scan grid's y
coords}
      for zpos:=0 to GridDepth-1 do begin        {scan grid's z
coords}

        ThisGroup:=PointGroup(scr, xpos, ypos, zpos);

        { ThisGroup's points are assigned as follows:    P3
P5
                                                      P1 P0 P2
                                                         P4        P6

                        Where P5 & P6 are in the Z plane }

        with ThisGroup do begin
          Current_Ex:=P0.Electric.x;        {current x component of
Electric field}
          Current_Ey:=P0.Electric.y;        {current y component of
Electric field}
          Current_Ez:=P0.Electric.z;        {current z component of
Electric field}
        end;

        if not Flip_YZ then begin
          VectGrp:=VectorGroup(ThisGroup, 2);
          CurlVect:=VectCurl(VectGrp);

          {Calculate required change in Electric field components
based on}
          {the observed gradients in the Magnetic field where the grid
point is}
          dEx_dt:=CurlVect.x/(Permittivity*Permeability); {change in x
component of E}
          dEy_dt:=CurlVect.y/(Permittivity*Permeability); {change in y
component of E}
          dEz_dt:=CurlVect.z/(Permittivity*Permeability); {change in z
component of E}

          with points[NewScreen,xpos,ypos,zpos].Electric do begin
            x:=Current_Ex+(dEx_dt*TimeStep);  {current Ex +
Changerate*Timestep}
            y:=Current_Ey+(dEy_dt*TimeStep);  {current Ey +
Changerate*Timestep}
            z:=Current_Ez+(dEz_dt*TimeStep);  {current Ez +
Changerate*Timestep}
          end;
        end
```

```
        else begin

          {Save new Electric field component values in new grid -
swapping Y & Z planes}
          with points[NewScreen,xpos,zpos,ypos].Electric do begin
            x:=Current_Ex; {current Ex}
            y:=Current_Ez; {current Ez}
            z:=Current_Ey; {current Ey}
          end;
        end;
      end;
    end;
  end;

if false then begin
  for xpos:=0 to GridWidth-1 do begin            {scan grid's x
coords}
    for ypos:=0 to GridHeight-1 do begin         {scan grid's y
coords}
      for zpos:=0 to GridDepth-1 do begin        {scan grid's z
coords}

        ThisGroup:=PointGroup(scr, xpos, ypos, zpos);

        { ThisGroup's points are assigned as follows:    P3
P5
                                                      P1 P0 P2
                                                         P4        P6

                        Where P5 & P6 are in the Z plane }

        if not Flip_YZ then begin
          VectGrp:=VectorGroup(ThisGroup, 1);
          DivVect:=VectDiv(VectGrp);

          with points[NewScreen,xpos,ypos,zpos].Electric do begin
            x:=x - DivVect.x ; {change in x component of E}
            y:=y - DivVect.y ; {change in y component of E}
            z:=z - DivVect.z ; {change in z component of E}
          end;
        end
      end;
    end;
  end;
end;

end;

procedure TForm1.Propagate;
begin
  repeat

  Application.ProcessMessages;
```

```
  if (New_StartOption<>StartOption) or Restart then begin  {Restart
with new start option}
    Restart:=false;
    Time:=0;
    StartOption:=New_StartOption;
    Initialise(False);
    InputFields(Screen,Time);                {set initial/continuing
input}
    UpdateDetails;           {ensure the screen is displaying up-to-
date info}
    InputFields(Screen,Time);                  {set initial/continuing
input}
    Auto_Scale(Screen); {determine the scaling factor required for
current data}
    UpdateDetails;           {ensure the screen is displaying up-to-
date info}
    UpdateBitmap(Screen); {Redraw the new data on the current
screen(bitmap)}
    DisplayScreen(Screen);   {Display the current screen}
  end;

  if DoUpdate then begin
    ReDraw:=false;           {Prevent re-draw unless specifically
asked to}
    UpdateDetails;           {ensure the screen is displaying up-to-
date info}

    if (not FreezeTime) or Flip_YZ then begin   {if time is running}

      if not GridTransformed then begin
        {Analyse the current grid & recalculate the new}
        {values for the E & H fields for the TimeStep}
        RecalcElectric(Screen);
        RecalcMagnetic(Screen);
      end;

      if Screen=0 then Screen:=1 else Screen:=0;  {initiate screen
swap}
      ReDraw:=true;
    end;

    Auto_Scale(Screen);     {determine the scaling factor required for
current data}
    UpdateDetails;           {ensure the screen is displaying up-to-
date info}
    if ( Form1.Timesteps.Value <> 0 ) then begin
      if Redraw and (Timestep_count mod Form1.Timesteps.Value = 0)
then begin
        UpdateBitmap(Screen); {Redraw the new data on the current
screen(bitmap)}
        DisplayScreen(Screen);   {Display the current screen}
        Timestep_count:=0;
      end;
    end;
```

```
    DoUpdate:=false;
  end;

  if ((not FreezeTime) and (not GridTransformed)) and (StartOption<>0)
and (not Flip_YZ) then begin
     Time:=Time+TimeStep;  {if time is running, increase it by
TimeStep}
     Inc(Timestep_count);
     DoUpdate:=true;
  end;

  Flip_YZ:=false;

  until Quit;
  Application.Terminate;
end;

procedure TForm1.UpdateBitmap(scr:smallint);
{Starting with a new, black, 24bit Colour bitmap, this routine takes
all the }
{relevent selected display options into account and produces a
complete bitmap}
{image ready to be displayed to screen. The main display options it
considers }
{are: (a)which field to display. (b)Colour or Greyscale. (c)Vector
arrows on/off.}
{(d)Z Plane tiling on/off. (e)Colours for each axis.}
var
  colour : longint;
  i,j,k,count :smallint;
  xyzcol: byte;
  value : double;
  VectorType : boolean;
  ArrowsX,ArrowsY,OffsetX,OffsetY: smallint;
  ArrowsOn,FirstY: boolean;
  ThisBitmap: Tbitmap;
  ActualX,ActualY: smallint;
  pointX,pointY,Xfrac,Yfrac: extended;
  Xleft,Ytop,imax,jmax,linescan: smallint;
  v1,v2,v3,v4,ArrowVect,ColVect,vect: vector;
  p1,p2,p3,p4: point;
  Xstep,Ystep,step_int: smallint;
  GridX,GridY: extended;
  JpegImage: TJpegImage;
  OutStream: TFileStream;
  path,fname,zstr: string;
  Point1,Point2,CopyPoint1,CopyPoint2: Point;
  xpos,ypos,zpos : smallint;

begin

  if scr=0 then begin      {get a new bitmap}
    NewBitmap(@Bitmap1);
    ThisBitmap:=Bitmap1;
  end
```

```
  else begin
    NewBitmap(@Bitmap2);
    ThisBitmap:=Bitmap2;
  end;

  if TileZ then begin
    Xstep:=1;{round2(1/(TileScaleX*ScrScaleX))-1;   {Don't bother with
points which map to}
    Ystep:=1; {round2(1/(TileScaleY*ScrScaleY))-1;   {the same screen
pixel}
  end        {needs to be one less than theory to prevent beat freq}
  else begin
    Xstep:=1;                      {Full size picture so use every
point}
    Ystep:=1;
  end;

    if (TileZ or (save_frames and save_3D)) then k:=0 else k:=Z_Plane;
{set Z axis position. If Z Plane tiling,}
repeat

  Application.ProcessMessages;

  if ( save_frames and save_3D ) then begin
    if scr=0 then begin      {get a new bitmap}
      if (k = Z_Plane) then begin
        NewBitmap(@Bitmap1);
        ThisBitmap:=Bitmap1;
      end
      else begin
        NewBitmap(@Bitmap2);
        ThisBitmap:=Bitmap2;
      end;
    end
    else begin
      if (k = Z_Plane) then begin
        NewBitmap(@Bitmap2);
        ThisBitmap:=Bitmap2;
      end
      else begin
        NewBitmap(@Bitmap1);
        ThisBitmap:=Bitmap1;
      end;
    end;
  end;

  if ViewTop then begin

    if not GridTransformed then begin

      if FreezeTime then GridTransformed:=true;

      for xpos:=0 to GridWidth-1 do begin            {scan grid's x
coords}
```

```
        for ypos:=0 to GridHeight-1 do begin         {scan grid's y
coords}
          for zpos:=0 to GridDepth-1 do begin        {scan grid's z
coords}
            Point1:=points[scr,xpos,ypos,zpos];
            Point2:=points[scr,xpos,zpos,ypos];
            CopyPoint1:=Point1;
            CopyPoint2:=Point2;

            {Swap Electric & Magnetic field component values in grid -
swapping Y & Z planes}
            {Note: Y plane zero point is at the Top of the screen,
hence the minus signs below}
            Point1.Electric.y := CopyPoint1.Electric.z;
            Point1.Electric.z := -CopyPoint1.Electric.y;
            Point1.Magnetic.y := CopyPoint1.Magnetic.z;
            Point1.Magnetic.z := -CopyPoint1.Magnetic.y;

            Point2.Electric.y := CopyPoint2.Electric.z;
            Point2.Electric.z := -CopyPoint2.Electric.y;
            Point2.Magnetic.y := CopyPoint2.Magnetic.z;
            Point2.Magnetic.z := -CopyPoint2.Magnetic.y;

            if (ypos = zpos) then begin
              points[scr,xpos,zpos,ypos] := Point1;
            end
            else if (ypos <= zpos) then begin
              points[scr,xpos,ypos,zpos] := Point2;
              points[scr,xpos,zpos,ypos] := Point1;
            end;
          end;
        end;
      end;

    end; // if not GridTransformed
  end // if ViewTop
  else begin

    if GridTransformed then begin
      GridTransformed:=false;
      for xpos:=0 to GridWidth-1 do begin            {scan grid's x
coords}
        for ypos:=0 to GridHeight-1 do begin         {scan grid's y
coords}
          for zpos:=0 to GridDepth-1 do begin        {scan grid's z
coords}
            Point1:=points[scr,xpos,ypos,zpos];
            Point2:=points[scr,xpos,zpos,ypos];
            CopyPoint1:=Point1;
            CopyPoint2:=Point2;

            {Swap Electric & Magnetic field component values in grid -
swapping Y & Z planes}
            {Note: Y plane zero point is at the Top of the screen,
hence the minus signs below}
```

```
          Point1.Electric.y := -CopyPoint1.Electric.z;
          Point1.Electric.z := CopyPoint1.Electric.y;
          Point1.Magnetic.y := -CopyPoint1.Magnetic.z;
          Point1.Magnetic.z := CopyPoint1.Magnetic.y;

          Point2.Electric.y := -CopyPoint2.Electric.z;
          Point2.Electric.z := CopyPoint2.Electric.y;
          Point2.Magnetic.y := -CopyPoint2.Magnetic.z;
          Point2.Magnetic.z := CopyPoint2.Magnetic.y;

          if (ypos = zpos) then begin
            points[scr,xpos,ypos,zpos] := Point2;
          end
          else if (ypos < zpos) then begin
            points[scr,xpos,zpos,ypos] := Point1;
            points[scr,xpos,ypos,zpos] := Point2;
          end;
        end;
      end;
    end; // if GridTransformed

  end; // else begin

end; // if ViewTop

repeat                                {all Z values are drawn, one
at a time }
  VectorType:=false;    {initialize working variables}
  vect:=NullVect;
  ColVect:=NullVect;
  value:=0;
  OffsetX:=0;
  OffsetY:=0;
  colour:=0;

  for j:=0 to BitmapY-1 do   {build array of bitmap line pointers}
    YLinePtrs[j]:=ThisBitmap.ScanLine[j];
  IsGrey:=not ShowColour;    {set local boolean values}
  ArrowsOn:=Arrows and (VectorX.Checked or VectorY.checked or
VectorZ.checked);
  if (not TileZ) and ArrowsOn then begin {if vector arrows are
required, do some calculations}
    arrow_step:=VectSpacing;    {using spacing value from control on
Form}
    GridX:=(GridWidth+1)*ScrScaleX;     {width of active region in
pixels}
    GridY:=(GridHeight+1)*ScrScaleY;    {height of active region in
pixels}

    if Spacing_Metres.Checked then      {if spacing defined as 'n'
per meter}
      arrow_step:=(GridX/(GridWidth/PPM))/arrow_step; {rescale the
step value}
```

```
      ArrowsX:=Trunc(GridX/arrow_step);  {determine number of arrows
across screen}
      ArrowsY:=Trunc(GridY/arrow_step); {determine number of arrows
up/down screen}
      OffsetX:=round2(Frac(GridX/arrow_step)*arrow_step/2);  {calc
offsets for 1st arrow}
      OffsetY:=round2(Frac(GridY/arrow_step)*arrow_step/2);
      step_int:=Round2(arrow_step);
      if step_int=0 then Inc(step_int);
    end
    else step_int:=1; {prevent compiler giving 'uninitialised' warning
message}

   i:=0;
    repeat    {scan through each (x,y) in Z plane of grid}
      j:=0;
      with points[scr,i,j,k] do repeat
       vect:=VectorProperty(DisplayField,points[scr,i,j,k]);

       case DisplayField of   {depending on which field is required to
display}
          1,2,5: begin {Show Electric Field, Magnetic Field, or Power
flow}
                   VectorType:=true;
                 end;
          3: value:=E_Energy(VectSize(vect));   {Show energy in
Electric Field}
          4: value:=B_Energy(VectSize(vect));   {Show energy in
Magnetic Field}
        end;

        if VectorType then begin {convert component values to colour
values}
          ColVect:=VectToColours(vect,Amplification);
          with ColVect do begin   {Combine colours}

colour:=(Round2(abs(x))*X_RGB)+(Round2(abs(y))*Y_RGB)+(Round2(abs(z))*
Z_RGB);
            SignArray[i,j,0]:=Sign(x);
            SignArray[i,j,1]:=Sign(y);
            SignArray[i,j,2]:=Sign(z);
          end;
        end;
        if (not VectorType) or IsGrey then begin {if greyscale display
required}
          if VectorType then value:=VectSize(vect); {if vector use
vector's size}
          xyzcol:=ColourRange(value,Amplification);   {set equal RGB
colour values}
          colour:=xyzcol*RED+xyzcol*GREEN+xyzcol*BLUE;    {Combine
colours}
          SignArray[i,j,0]:=Sign(xyzcol);
          SignArray[i,j,1]:=Sign(xyzcol);
          SignArray[i,j,2]:=Sign(xyzcol);
        end;
```

```
        if Rendered then {if colour or grey rendered display req'd}
          ColourArray[i,j]:=colour;   {save point's colour in array}
        Inc(j,Ystep);
      until j>GridHeight-1;
      Inc(i,Xstep);
    until i>GridWidth-1;

      ZplanePos:=k; {set global variable for current Z plane used by
PlotPoint}

      i:=0;
      if Rendered then    {if colour or greyscale Rendered display
req'd}
        repeat
          j:=0;
          repeat
            PlotPoint(ThisBitmap,i,j);  {plot each point onto bitmap}
            Inc(j,Ystep);
          until j>GridHeight-1;
          Inc(i,Xstep);
        until i>Gridwidth-1;

      if TileZ then                    {if Z Plane tiling required}
        TileCursor(ThisBitmap,ZplanePos,clGray);

      if VectorType and ArrowsOn and (not TileZ) then begin  {if
Vector arrows required}

        if Spacing_gridpoints.Checked then begin

          if ( ScrScaleX > ScrScaleY ) then
arrow_step:=VectSpacing*ScrScaleY;
          if ( ScrScaleY >= ScrScaleX ) then
arrow_step:=VectSpacing*ScrScaleX;

          ArrowScale:=ArrowScaleFactor*arrow_step/$FF;    {set scaling
factor for arrow size}

          i:=0;
          repeat
            j:=0;
            repeat
              ActualX:=round2(GetRealX((i+0.5)));  {the actual screen
x location (on the image control)}
              ActualY:=round2(GetRealY((j+0.5)));  {the actual screen
y location (on the image control)}
              p1:=points[scr,i,j,k];
              ArrowVect:=VectorProperty(DisplayField,p1);
              with ArrowVect do begin
                x:=x*ReScaleFactor*ArrowScale;          {re-scale raw
values}
                y:=y*ReScaleFactor*ArrowScale;
                z:=z*ReScaleFactor*ArrowScale;
              end;
```

```
              DrawArrow(ThisBitmap,ActualX,ActualY,ArrowVect);
{draw the vector arrow}
              Inc(j,VectSpacing);
            until j>GridHeight-1;
            Inc(i,VectSpacing);
          until i>Gridwidth-1;

        end
        else begin

          ArrowScale:=ArrowScaleFactor*arrow_step/$FF;    {set scaling
factor for arrow size}

          i:=OriginX+OffsetX;      {calc screen x coord of arrow
start}
          imax:=round2(OriginX+GridWidth*ScrScaleX);   {calculate
screen coords}
          jmax:=round2(OriginY+GridHeight*ScrScaleY);  {of edge of
active region}
          repeat
            pointX:=(i-OriginX)/ScrScaleX;  {calc grid x coord for
screen (i,j) }
            Xleft:=Trunc(pointX);    {find x coord of top left point}
            Xfrac:=pointX-Xleft;     {calc fractional distance to next
point}
            j:=OriginY+OffsetY;    {calc screen y coord of arrow
start}
            repeat
              pointY:=(j-OriginY)/ScrScaleY; {calc grid y coord for
screen (i,j) }
              Ytop:=Trunc(pointY);   {find y coord of top left point}
              Yfrac:=pointY-Ytop;    {calc fractional distance to next
point}
              p1:=points[scr,Xleft,Ytop,k];
              if Xleft<(GridWidth-1) then      {get relevant point
vectors}
                p2:=points[scr,Xleft+1,Ytop,k] else p2:=NullPoint;
              if YTop<(GridHeight-1) then
                p3:=points[scr,Xleft,Ytop+1,k] else p3:=NullPoint;
              if (Xleft<(GridWidth-1)) and (YTop<(GridHeight-1)) then
                p4:=points[scr,Xleft+1,Ytop+1,k] else p4:=NullPoint;
              if not PointNull(p1) then
                v1:=VectorProperty(DisplayField,p1) else v1:=NullVect;
              if not PointNull(p2) then
                v2:=VectorProperty(DisplayField,p2) else v2:=NullVect;
              if not PointNull(p3) then
                v3:=VectorProperty(DisplayField,p3) else v3:=NullVect;
              if not PointNull(p4) then
                v4:=VectorProperty(DisplayField,p4) else v4:=NullVect;
              ArrowVect:=VectorInterpolate(v1,v2,v3,v4,Xfrac,Yfrac);
              with ArrowVect do begin
                x:=x*ReScaleFactor*ArrowScale;          {re-scale raw
values}
                y:=y*ReScaleFactor*ArrowScale;
                z:=z*ReScaleFactor*ArrowScale;
```

```
              end;
              DrawArrow(ThisBitmap,i,j,ArrowVect);    {draw the vector
arrow}
              Inc(j,step_int);
            until j>=jmax;     {do next arrow till done}
            Inc(i,step_int);
          until i>=imax;
        end;
      end;
      Inc(k);                {do next Z Plane}
  until (not TileZ) or (k>GridDepth-1); {only repeat if tiling req'd
and Z plane in range}

  if TileZ then   {draw a red rectangle around the currently selected
Z plane}
    TileCursor(ThisBitmap,Z_Plane,clRed);

  if MaintainAspect and (not TileZ) then with ThisBitmap.Canvas do
begin
    Pen.Width:=1;                         {set pen to thin white line}
    Pen.Style:=psSolid;
    Pen.Color:=clGray;
    if OriginX<>0 then begin                    {Draw boundary lines
for Grid within the }
      MoveTo(OriginX-2,0);          {Screen space if the Aspect ratio
is being}
      LineTo(OriginX-2,BitmapY);    {preserved and onlypart of the
screen space}
      MoveTo(BitmapX-OriginX,0);  {is active.}
      LineTo(BitmapX-OriginX,BitmapY);
    end;
    if OriginY<>0 then begin
      MoveTo(0,OriginY-2);
      LineTo(BitmapX,OriginY-2);
      MoveTo(0,BitmapY-OriginY);
      LineTo(BitmapX,BitmapY-OriginY);
    end;
  end;

  if ( save_frames ) then begin
    // convert to jpeg and save
    try
      JpegImage := TJpegImage.Create;
      JpegImage.Assign(ThisBitmap);

      path := '.\Frames\';

      case DisplayField of   {depending on which field is required to
display}
          1: fname := 'Electric';
          2: fname := 'Magnetic';
          5: fname := 'Power';
          3: fname := 'E_Energy';
          4: fname := 'B_Energy';
      end;
```

```
      if (FrameCount < 10) then fname:=fname+'00000'
      else if (FrameCount < 100) then fname:=fname+'0000'
      else if (FrameCount < 1000) then fname:=fname+'000'
      else if (FrameCount < 10000) then fname:=fname+'00'
      else if (FrameCount < 100000) then fname:=fname+'0';

      if (k < 10) then zstr:='00000'
      else if (k < 100) then zstr:='0000'
      else if (k < 1000) then zstr:='000'
      else if (k < 10000) then zstr:='00'
      else zstr:='0';

      if not DirectoryExists(path) then CreateDir(path);

      if ( save_3D ) then begin
        OutStream :=
TFileStream.Create(path+fname+IntToStr(FrameCount)+'_z'+zstr+IntToStr(
k)+'.jpg',fmOpenWrite or fmCreate);
      end
      else begin
        OutStream :=
TFileStream.Create(path+fname+IntToStr(FrameCount)+'.jpg',fmOpenWrite
or fmCreate);
      end;

      JpegImage.SaveToStream(OutStream);
    finally
      JpegImage.Free;
      OutStream.Free;
    end;
  end;

until (not (save_frames and save_3D)) or (k>GridDepth-1); {only repeat
if tiling req'd and Z plane in range}

Inc(FrameCount);

  if ViewTop and not ( FreezeTime and GridTransformed ) then begin
    GridTransformed:=false;
    for xpos:=0 to GridWidth-1 do begin            {scan grid's x
coords}
      for ypos:=0 to GridHeight-1 do begin         {scan grid's y
coords}
        for zpos:=0 to GridDepth-1 do begin        {scan grid's z
coords}
          Point1:=points[scr,xpos,ypos,zpos];
          Point2:=points[scr,xpos,zpos,ypos];
          CopyPoint1:=Point1;
          CopyPoint2:=Point2;

          {Swap Electric & Magnetic field component values in grid -
swapping Y & Z planes}
          {Note: Y plane zero point is at the Top of the screen, hence
the minus signs below}
```

```
          Point1.Electric.y := -CopyPoint1.Electric.z;
          Point1.Electric.z := CopyPoint1.Electric.y;
          Point1.Magnetic.y := -CopyPoint1.Magnetic.z;
          Point1.Magnetic.z := CopyPoint1.Magnetic.y;

          Point2.Electric.y := -CopyPoint2.Electric.z;
          Point2.Electric.z := CopyPoint2.Electric.y;
          Point2.Magnetic.y := -CopyPoint2.Magnetic.z;
          Point2.Magnetic.z := CopyPoint2.Magnetic.y;

          if (ypos = zpos) then begin
            points[scr,xpos,ypos,zpos] := Point2;
          end
          else if (ypos < zpos) then begin
            points[scr,xpos,zpos,ypos] := Point1;
            points[scr,xpos,ypos,zpos] := Point2;
          end;
         end;
      end;
    end;
  end;

end;

procedure TForm1.DisplayScreen(scr:smallint);  {display the current
bitmap}
begin
  if scr=0 then
    CurrentBitmap:= Bitmap1  {set the picture control to display
bitmap1}
  else
    CurrentBitmap:= Bitmap2;  {set the picture control to display
bitmap2}
  Image1.Picture.Graphic:=CurrentBitmap
end;

procedure TForm1.InputFields(scr:smallint;Time:extended);
{provide grid with some initial starting values for the Electric &}
{Magnetic fields. A number of different starting options are
available}
var

value,len_sqr,len,temp,E_phase,amplitude,phase_scale,in_out_freq_facto
r : extended;
  x,y,z,midx,midy,midz,starty,endy,startz,endz,r : smallint;
  xpos,ypos,zpos: smallint;
  rangey,rangez : smallint;
begin
  midx:=round2(GridWidth/2)-1;   {determine mid point of grid}
  midy:=round2(GridHeight/2)-1;
  midz:=round2(GridDepth/2)-1;
  case StartOption of
  1: begin
       if Time<(WaveNumber*WavePeriod1) then begin
         value:=Sin((Time/WavePeriod1)*2*Pi);
```

```
          for y:=round(midy-(GridHeight/8))-1 to
round(midy+(GridHeight/8))-1 do begin
            with points[scr,round(GridWidth/2)-1,y,midz] do begin
              Electric.z:=value*E_Amplitude;
              Magnetic.y:=value*B_Amplitude;
            end;
          end;
        end;
      end;
   2: begin  {several cycles of a sin wave pattern along the x
direction}
        if Time=0 then begin
          starty:=0;{round(midy-(GridHeight/4));}
          endy:=GridHeight-1;{round(midy+(GridHeight/4));}
          startz:=0;{round(midz-(GridDepth/4));}
          endz:=GridDepth-1;{round(midz+(GridDepth/4));}
          rangey:=endy-starty;
          rangez:=endz-startz;
          for x:=round(midx-(WaveNumber*WavePeriod2))-1 to
round(midx+(WaveNumber*WavePeriod2))-1 do begin
            value:=Sin(((x-midx)/WavePeriod2)*2*Pi);
           for y:= starty to endy do begin
             for z:= startz to endz do begin
               temp:=-value*E_Amplitude*Gauss(4*((z-startz-
rangez/2)/rangez));
               with points[scr,x,y,z] do begin
                 Electric.z:=temp;
        {         Magnetic.z:=temp;       }
               end;
            {     Magnetic.y:=-value*B_Amplitude*Gauss(4*((y-starty-
rangey/2)/rangey));}
        {       with points[1,x,y,z] do    }
        {         Electric.z:=temp;        }
              end;
            end;
          end;
      {   ReCalcMagnetic(1);        }
        end;
      end;
   3:begin
        if Time<(WaveNumber*WavePeriod1) then begin
          value:=Sin((Time/WavePeriod1)*2*Pi);
          starty:=round(midy-(GridHeight/8))-1;
          endy:= round(midy+(GridHeight/8))-1;
          for y:= starty to endy do
            with points[scr,midx,y,midz] do begin
              Electric.z:=value*E_Amplitude*Sin((y-starty)/(endy-
starty)*2*Pi);
              Magnetic.x:=value*B_Amplitude*Sin((y-starty)/(endy-
starty)*2*Pi);
            end;
        end;
      end;
   4:begin
        if Time<(WaveNumber*WavePeriod1) then begin
```

```
          y:=round((GridHeight/8)*Sin((Time/WavePeriod1)*2*Pi))-1;
          with points[scr,midx,midy-y,midz] do begin
            Electric.z:=E_Amplitude;
            Magnetic.x:=B_Amplitude*Sin((Time/WavePeriod1)*2*Pi);
          end;
          with points[scr,midx,midy+y,midz] do begin
            Electric.z:=-E_Amplitude;
            Magnetic.x:=B_Amplitude*Sin((Time/WavePeriod1)*2*Pi);
          end;
        end;
     end;
   5:begin
       if Time=0 then with
         points[scr,midx,midy,midz] do begin
           Electric.z:=1;
         end;
     end;
   6:begin
       if Time=0 then
         for Y:=round(GridHeight/6)-1 to round(GridHeight-
GridHeight/6)-1 do with
           points[scr,y,y,midz] do
             Electric.z:=1;
     end;
   7:begin
       if Time=0 then
         for x:=0 to GridWidth-1 do begin
           for y:=0 to GridHeight-1 do
             points[scr,x,y,midz].Electric.z:=E_Amplitude;
{           y:=round(midy+midy*Gauss(4*((x-midx)/GridWidth)));
           points[scr,x,y,midz].Electric.z:=E_Amplitude;}
         end;
     end;
   8:begin    {Model an Electron Type E & H field setup}
       if Time=0 then begin
         r:=Min(midx,midy);
         r:=Min(r,midz)-1;
         for x:=-r to r do
           for y:=-r to r do
             for z:=-r to r do with points[scr,midx+x,midy+y,midz+z] do
begin
               len:=sqrt(sqr(x)+sqr(y)+sqr(z));
               if len<>0 then begin
//                   Electric.x:=(x/len)*E_Amplitude*(-
1)*Gauss(4*len/(2*r));
//                   Electric.y:=(y/len)*E_Amplitude*(-
1)*Gauss(4*len/(2*r));
//                   Electric.z:=(z/len)*E_Amplitude*(-
1)*Gauss(4*len/(2*r));
                 Electric.x:=(x/len)*E_Amplitude;//*(-
1)*Gauss(4*len/(2*r));
                 Electric.y:=(y/len)*E_Amplitude;//*(-
1)*Gauss(4*len/(2*r));
                 Electric.z:=(z/len)*E_Amplitude;//*(-
1)*Gauss(4*len/(2*r));
```

```
//                 Magnetic.z:=(z/len)*B_Amplitude*(-
1)*Gauss(4*len/(2*r));
              end
              else begin
                Electric.x:=E_Amplitude/3;
                Electric.y:=E_Amplitude/3;
                Electric.z:=E_Amplitude/3;
//                Magnetic.x:=B_Amplitude/3;
//                Magnetic.y:=B_Amplitude/3;
//                Magnetic.z:=B_Amplitude/3;
              end;
            end;
      end;
    end;

 9:begin   { A much better model of an Electron - as a Spherical
standing wave }
    r:=Min(midx,midy);
    r:=Min(r,midz)-1;
    // calculate the phase scaling factor for the Compton wavelength
of an Electron's standing wave.
    phase_scale:=0.5*(r/PPM)/ElectronComptonWavelength;
    if r<>0 then begin
      for x:=-r to r do
        for y:=-r to r do
         for z:=-r to r do with points[scr,midx+x,midy+y,midz+z] do
begin
            len:=sqrt(sqr(x)+sqr(y)+sqr(z));
            E_phase:=2*(len/r)*2*Pi;
            amplitude:=sqr((r-len)/r);

Electric.x:=amplitude*E_Amplitude*(0.9*Sin(phase_scale*E_phase)+Sin(ph
ase_scale*(2*Pi-E_phase)));

Electric.y:=amplitude*E_Amplitude*(0.9*Sin(phase_scale*E_phase)+Sin(ph
ase_scale*(2*Pi-E_phase)));

Electric.z:=amplitude*E_Amplitude*(0.9*Sin(phase_scale*E_phase)+Sin(ph
ase_scale*(2*Pi-E_phase)));
            Magnetic.z:=((r-len)/r)*B_Amplitude;
            Magnetic.y:=0;
            Magnetic.z:=0;
          end
      end;
   end;

 10:begin   { A much better model of an Electron - as a Spherical
standing wave }
    r:=Min(GridWidth,GridHeight);
    r:=Min(r,GridDepth);
    // calculate the phase scaling factor for the Compton wavelength
of an Electron's standing wave.
    phase_scale:=(r/PPM)/ElectronComptonWavelength;
    if r<>0 then begin
      for x:=0 to (GridWidth-1) do
```

```
        for y:=0 to (GridHeight-1) do
         for z:=0 to (GridDepth-1) do begin
          xpos:=x - Trunc(3*GridWidth/8);
          ypos:=y - Trunc(3*GridHeight/8);
          zpos:=z - Trunc(GridDepth/2);

          len:=sqrt(sqr(xpos)+sqr(ypos)+sqr(zpos));
          E_phase:=2*(len/r)*2*Pi;
          amplitude:=sqr(1/(len+1));

          with points[scr,x,y,z] do begin
            with Electric do begin

x:=amplitude*E_Amplitude*(Sin(phase_scale*E_phase)+Sin(phase_scale*(2*
Pi-E_phase)));

y:=amplitude*E_Amplitude*(Sin(phase_scale*E_phase)+Sin(phase_scale*(2*
Pi-E_phase)));

z:=amplitude*E_Amplitude*(Sin(phase_scale*E_phase)+Sin(phase_scale*(2*
Pi-E_phase)));
            end;
            with Magnetic do begin
              x:=0;
              y:=0;
              z:=(1/(len+1))*amplitude*B_Amplitude;
            end;
          end;
        end;

      for x:=0 to (GridWidth-1) do
        for y:=0 to (GridHeight-1) do
         for z:=0 to (GridDepth-1) do begin
          xpos:=x - Trunc(5*GridWidth/8);
          ypos:=y - Trunc(5*GridHeight/8);
          zpos:=z - Trunc(GridDepth/2);

          len:=sqrt(sqr(xpos)+sqr(ypos)+sqr(zpos));
          E_phase:=2*(len/r)*2*Pi;
          amplitude:=sqr(1/(len+1));

          with points[scr,x,y,z] do begin
            with Electric do begin
              x:=x +
amplitude*E_Amplitude*(Sin(phase_scale*E_phase)+Sin(phase_scale*(2*Pi-
E_phase)));
              y:=y +
amplitude*E_Amplitude*(Sin(phase_scale*E_phase)+Sin(phase_scale*(2*Pi-
E_phase)));
              z:=z +
amplitude*E_Amplitude*(Sin(phase_scale*E_phase)+Sin(phase_scale*(2*Pi-
E_phase)));
            end;
            with Magnetic do begin
              x:=0;
```

```
              y:=0;
              z:=z + (1/(len+1))*amplitude*B_Amplitude;
            end;
          end;
        end;
    end;
  end;

  11:begin   { A much better model of an Electron - as a Spherical
standing wave }
    r:=Min(midx,midy);
    r:=Min(r,midz)-1;
    // calculate the phase scaling factor for the Compton wavelength
of an Electron's standing wave.
    phase_scale:=0.5*(r/PPM)/ElectronComptonWavelength;
    in_out_freq_factor := 0.9;
    if r<>0 then begin
      for x:=-r to r do
        for y:=-r to r do
         for z:=-r to r do with points[scr,midx+x,midy+y,midz+z] do
begin
            len_sqr:=sqr(x)+sqr(y)+sqr(z);
            len:=sqrt(len_sqr);
            E_phase:=2*(len/r)*2*Pi;
            if ( len_sqr ) < 0.0001 then begin
              len_sqr := r - 0.0001;
              len := sqrt(len_sqr);
            end;

amplitude:=abs((1/len_sqr)*E_Amplitude*(in_out_freq_factor*Sin(phase_s
cale*E_phase)+Sin(in_out_freq_factor*phase_scale*(2*Pi-E_phase))));
            Electric.x:=amplitude * sqrt( len_sqr - (sqr(y) + sqr(z))
);
            Electric.y:=amplitude * sqrt( len_sqr - (sqr(x) + sqr(z))
);
            Electric.z:=amplitude * sqrt( len_sqr - (sqr(x) + sqr(y))
);
            if ( x < midx ) then Electric.x := -Electric.x;
            if ( y < midy ) then Electric.y := -Electric.y;
            if ( z < midz ) then Electric.z := -Electric.z;
            Magnetic.z:=(1/len)*B_Amplitude;
            Magnetic.y:=0;
            Magnetic.z:=0;
          end
      end;
    end;
  end;
end;

procedure TForm1.Start1Click(Sender: TObject);
{The 1st Start option has been selected}
begin
  New_StartOption:=1;
  DoUpdate:=true;
end;
```

```
procedure TForm1.Start2Click(Sender: TObject);
{The 2nd Start option has been selected}
begin
  New_StartOption:=2;
  DoUpdate:=true;
end;

procedure TForm1.start3Click(Sender: TObject);
{The 3rd Start option has been selected}
begin
  New_StartOption:=3;
  DoUpdate:=true;
end;

procedure TForm1.Start4Click(Sender: TObject);
{The 4th Start option has been selected}
begin
  New_StartOption:=4;
  DoUpdate:=true;
end;

procedure TForm1.Start5Click(Sender: TObject);
{The 5th Start option has been selected}
begin
  New_StartOption:=5;
  DoUpdate:=true;
end;

procedure TForm1.Start6Click(Sender: TObject);
{The 6th Start option has been selected}
begin
  New_StartOption:=6;
  DoUpdate:=true;
end;

procedure TForm1.Start7Click(Sender: TObject);
{The 7th Start option has been selected}
begin
  New_StartOption:=7;
  DoUpdate:=true;
end;

procedure TForm1.Start8Click(Sender: TObject);
{The 8th Start option has been selected}
begin
  New_StartOption:=8;
  DoUpdate:=true;
end;

procedure TForm1.Start9Click(Sender: TObject);
{The 9th Start option has been selected}
begin
  New_StartOption:=9;
  DoUpdate:=true;
```

```
end;

procedure TForm1.Start10Click(Sender: TObject);
{The 10th Start option has been selected}
begin
  New_StartOption:=10;
  DoUpdate:=true;
end;

function TForm1.sign(number: extended): shortint;
{This function takes a real number and returns either -1 or +1 }
{If: number<0, -1 is returned }
{    number>=0, +1 is returned }
var
  signval: shortint;
begin
  if number>=0 then signval:=1
  else signval:=-1;
  Result:=signval;
end;

function TForm1.Gradient(val1, val2: extended): extended;
{Determines the gradient from val1 to val2 assuming a horizontal
separation}
{between the points of one unit}
begin
  result:=Val2-Val1;
end;

function TForm1.Gradient_3point(val1, val2, val3: extended): extended;
{Determines the gradient from val1 to val2 assuming a horizontal
separation}
{between the points of one unit}
begin
  result:= (Gradient(val1, val2) + Gradient(val2, val3))/2;
end;

procedure TForm1.Field1Click(Sender: TObject);
{The Electric field has been selected to be displayed}
begin
  New_DisplayField:=1;
  DoUpdate:=true;
end;

procedure TForm1.Field2Click(Sender: TObject);
{The Magnetic field has been selected to be displayed}
begin
  New_DisplayField:=2;
  DoUpdate:=true;
end;

procedure TForm1.Field3Click(Sender: TObject);
{The Electric field Energy has been selected to be displayed}
begin
  New_DisplayField:=3;
```

```
  DoUpdate:=true;
end;

procedure TForm1.Field4Click(Sender: TObject);
{The Magnetic field Energy has been selected to be displayed}
begin
  New_DisplayField:=4;
  DoUpdate:=true;
end;

procedure TForm1.Field5Click(Sender: TObject);
{The Power flow field has been selected to be displayed}
begin
  New_DisplayField:=5;
  DoUpdate:=true;
end;

procedure TForm1.TimeFreezeClick(Sender: TObject);
{The Freeze Time control has been changed, so toggle button's text}
begin
  if FreezeTime then New_FreezeTime:=false else New_FreezeTime:=true;
  DoUpdate:=true;
end;

function TForm1.Gauss(x: extended): extended;
{return a value (between 0 and -1) corresponding to a gaussian
distribution}
{based of the x value supplied. The whole curve (essentially) is
defined }
{between x=-2 to x=2}
begin
  result:=Exp(-sqr(x));
end;

procedure TForm1.UpdateDetails();
var
  ReCalcTileSize,ReCalcAspectRatio: boolean;
begin
  ReCalcTileSize:=false;          {Default re-calc to Off}
  ReCalcAspectRatio:=false;       {Default re-calc to Off}

  Flip_YZ:=New_Flip_YZ;
  New_Flip_YZ:=false;

  if ViewTop<>ViewFromTop.Checked then begin
    ViewTop := ViewFromTop.Checked;
    ReDraw:=true;                       {Trigger a screen re-draw}
  end;

  // If flipping Y & Z planes, need to make sure they have the same
dimensions first.
  if Flip_YZ then begin
    if GridHeight > GridDepth then New_GridHeight := GridDepth;
    if GridDepth > GridHeight then New_GridDepth := GridHeight;
  end;
```

```
  {Reset the Grid Dimensions if required (retains existing data)}
  if ((New_GridWidth<>GridWidth) or (New_GridHeight<>GridHeight) or
      (New_GridDepth<>GridDepth)) or FirstPass then begin
    ReAllocGridMemory;                 {Re-allocate memory for new
Grid size}
    GridWidth:=New_GridWidth;          {Adopt new Dimensions}
    GridHeight:=New_GridHeight;
    GridDepth:=New_GridDepth;

    if ( GridHeight <> GridDepth ) then ViewFromTop.Enabled := false
else ViewFromTop.Enabled := true;

    GridX.Text:=IntToStr(GridWidth);   {Ensure Controls reflect
Current}
    GridY.Text:=IntToStr(GridHeight);  {program settings.}
    GridZ.Text:=IntToStr(GridDepth);
    SetGridGlobals;                    {Re-Calc Globals for new Grid
size}
    ReCalcAspectRatio:=true;           {Trigger an Aspect Ratio
recalculation}
    ReCalcTileSize:=true;              {Trigger a Tile Size
recalculation}
    ReDraw:=true;                      {Trigger a screen re-draw}
  end;

  {Update Time Related Displays}
  TimeDisplay.Text:='Time (Secs) : '+FloatToStr(Time); {update time
display}
  if (FreezeTime<>New_FreezeTime) or FirstPass then begin
    FreezeTime:=New_FreezeTime;
    with TimeFreeze do
      if FreezeTime then
        Caption:='Unfreeze Time'
      else
        Caption:='Freeze Time';
  end;

  {Update the Auto-Scale selection}
  if (New_AutoScale<>AutoScale) or FirstPass then begin
    if AutoScale=CONTINUAL then begin {Re-Enable control incase it was
out of}
      ReScale.Enabled:=true;          {range whilst autoscaling.}
      New_ReScale:=ReScaleFactor;     {Prevent the next update, as the
control}
      New_DisplayLevel:=Display_Level;{was turned off.}
    end;
    AutoScale:=New_AutoScale;         {Adopt the new control state.}
    case AutoScale of
      START:     begin
                   if Time=0 then begin
                     AutoWarn.Top:=Auto1.Top+2; {Show Auto-scale
warning indicator}
                     AutoWarn.Visible:=true;    {next to the relevant
button.}
```

```
                    AutoWarnTimer.Enabled:=true;
                  end
                  else begin
                    AutoWarn.Visible:=false;
                    AutoWarnTimer.Enabled:=false;
                  end;
                end;
      CONTINUAL: begin
                  AutoWarn.Top:=Auto2.Top+2; {Show Auto-scale warning
indicator}
                  AutoWarn.Visible:=true;    {next to the relevant
button.}
                  AutoWarnTimer.Enabled:=true;
                end;
      NEVER:     begin
                  AutoWarn.Visible:=false;
                  AutoWarnTimer.Enabled:=false;
                end;
    end;
  end;

  {Turn off Auto Scale indicator after time started if req'd}
  if (Time<>0) and (AutoScale=START) then begin
    AutoWarn.Visible:=false;
    AutoWarnTimer.Enabled:=false;
  end;

  {Update the display mode Colour/GreyScale}
  if New_ShowColour<>ShowColour then begin
    ShowColour:=New_ShowColour;
    ReDraw:=true;                      {Trigger a screen re-draw}
  end;

  {Update Axis Colour Allocation}
  if UpdateColours or FirstPass then begin
    UpdateAxisColours(New_AxisColours);
    UpdateColours:=false;
    ReDraw:=true;                      {Trigger a screen re-draw}
  end;

  if (New_Rendered<>Rendered) or FirstPass then begin
    Rendered:=New_Rendered;
    if Rendered then begin
      RendGroup.Enabled:=true;
      RenderOption1.Enabled:=true;
      RenderOption2.Enabled:=true;
      RenderOption3.Enabled:=true;
      ColourButton.Enabled:=true;
      GreyScaleButton.Enabled:=true;
    end
    else begin
      RendGroup.Enabled:=false;
      RenderOption1.Enabled:=false;
      RenderOption2.Enabled:=false;
      RenderOption3.Enabled:=false;
```

```
      ColourButton.Enabled:=false;
      GreyScaleButton.Enabled:=false;
    end;
    ReDraw:=true;                      {Trigger a screen re-draw}
  end;

  {Update the Maintain Aspect Ratio Checkbox control}
  if (New_MaintainAspect<>MaintainAspect) or FirstPass then begin
    MaintainAspect:=New_MaintainAspect;
    ReCalcAspectRatio:=true;           {Trigger an Aspect Ratio
recalculation}
    ReCalcTileSize:=true;              {Trigger a Tile Size
recalculation}
    ReDraw:=true;                      {Trigger a screen re-draw}
  end;

  {Update the screen rendering Option}
  if (New_Render<>Render) or FirstPass then begin
    Render:=New_Render;
    ReCalcTileSize:=true;     {Reset Tiling size for new Rendering
selection}
    ReDraw:=true;                      {Trigger a screen re-draw}
  end;

  {Update Z Tiling Option Variable and button display}
  if (TileZ<>New_TileZ) or FirstPass then begin
    TileZ:=New_TileZ;
    ReDraw:=true;                      {Trigger a screen re-draw}
    with Z_Tiling do
      if TileZ then
        Caption:='Single Z Plane'
      else
        Caption:='Tile Z Planes';
  end;

  {Update Currently displayed Z plane, the control's value and its
display}
  if (Z_Plane<>New_ZPlane) or FirstPass then begin
    {Shift Tile Cursor if new tile selected}
    if TileZ then begin
      TileCursor(CurrentBitmap,Z_Plane,clGray);   {return old Tile's
border to Grey}
      TileCursor(CurrentBitmap,New_ZPlane,clRed); {Outline new Tile's
border in Red}
    end
    else ReDraw:=true;                  {Trigger a screen re-draw}
    Z_Plane:=New_ZPlane;
    ZPlane.Position:=Z_Plane+1;
    Z_Plane_Number.Text:='Z Plane :  '+IntToStr(Z_Plane+1)+'  (of
'+IntToStr(GridDepth)+')';
  end;

  if ReCalcAspectRatio then SetAspectRatio;    {Re-Calc if flagged}
  if ReCalcTileSize then SetTileSize;
```

```
  {Update the Re-scaling factor value and ensure control's value
agrees}
  if (ReScaleFactor<>New_ReScale) or FirstPass then begin
    ReScaleFactor:=New_ReScale;
    ReScale.Enabled:=true;
    ReDraw:=true;                      {Trigger a screen re-draw}
  end;

  {Update the DisplayLevel control's position}
  if (Display_Level<>New_DisplayLevel) or FirstPass then begin
    Display_Level:=New_DisplayLevel;
    if ((AutoScale=CONTINUAL) or ((AutoScale=START) and (Time=0)))
then
      DisplayLevel.Position:=Display_Level;
    ReDraw:=true;                      {Trigger a screen re-draw}
  end;

  if (Rate_Of_Time<>New_RateOfTime) or FirstPass then begin
    Rate_Of_Time:=New_RateOfTime;
    TimeStep:=(Rate_Of_Time/1000)*PointSize/SpeedOfLight; {Increment
of Time per iteration - Secs }
  end;

  {Calc the Amplification factor & update its display}
//  CentrePos:=Round2(DisplayLevel.Max/2);
  Amplification:=DisplayLevel.Max - Display_Level;
  AmpDisplay.Text:='Amplification :
'+IntToStr(Trunc(Amplification))+'%';
  Amplification:=ReScaleFactor*Amplification/100;

  {Update which Field is being shown}
  if DisplayField<>New_DisplayField then begin
    DisplayField:=New_DisplayField;
    ReDraw:=true;                      {Trigger a screen re-draw}
  end;

  {Update all the Field's statistical values}
  UpdateE_Energy(Screen);
  UpdateB_Energy(Screen);
  Energy1.Text:=FloatToStr(E_Energy_Tot); {display the Electric field
energy total}
  Energy2.Text:=FloatToStr(B_Energy_Tot); {display the Magnetic field
energy total}
  Energy3.Text:=FloatToStr(E_Energy_Tot+B_Energy_Tot); {display total
field energy}

  ActualGridWidth.Text:=FloatToStr(ActualWidth); {display actual size
in metres that grid represents}

  try
    New_VectSpacing:=StrToInt(VectorSpacing.Text);
  except                       {catch any invalid integer conditions}
    VectorSpacing.Text:='11';
    New_VectSpacing:=11;
  end;
```

```
  {Update the Vector Arrows' spacing value}
  if (New_VectSpacing<>VectSpacing) or FirstPass then begin
    VectSpacing:=New_VectSpacing;
    ReDraw:=true;                      {Trigger a screen re-draw}
  end;

  {The X,Y or Z vector arrows have been selected/de-selected, so re-
draw req'd}
  if VectorChange then begin
    VectorChange:=false;
    ReDraw:=true;
  end;

  {if the Arrows separation units have been changed, trigger a re-
draw}
  if ArrowsUnitsChange then begin
    ArrowsUnitsChange:=false;
    ReDraw:=true;
  end;

  {Update Arrows variable & status of Vector Arrows controls}
  if (Arrows<>New_Arrows) or FirstPass then begin
    Arrows:=New_Arrows;
    if Arrows then begin
      VectorX.Enabled:=true;
      VectorY.Enabled:=true;
      VectorZ.Enabled:=true;
      Spacing_Text.Enabled:=true;
      VectorSpacing.Enabled:=true;
      Spacing_pixels.Enabled:=true;
      Spacing_metres.Enabled:=true;
      Spacing_gridpoints.Enabled:=true;
    end
    else begin
      VectorX.Enabled:=false;
      VectorY.Enabled:=false;
      VectorZ.Enabled:=false;
      Spacing_Text.Enabled:=false;
      VectorSpacing.Enabled:=false;
      Spacing_pixels.Enabled:=false;
      Spacing_metres.Enabled:=false;
      Spacing_gridpoints.Enabled:=false;
    end;
    ReDraw:=true;                      {Trigger a screen re-draw}
  end;

  if (ArrowScaleFactor <> New_ArrowScaleFactor) or FirstPass then
begin
    ArrowScaleFactor := New_ArrowScaleFactor/20;
    ReDraw:=true;                      {Trigger a screen re-draw}
  end;

  FirstPass:=false;
end;
```

```
procedure TForm1.ZPlaneChange(Sender: TObject);
{The Z plane control has been changed, so re-validate display values}
begin
  New_ZPlane:=ZPlane.Position-1;
  DoUpdate:=true;
end;

procedure TForm1.DisplayLevelChange(Sender: TObject);
{The displaylevel slider control has been changed, so re-validate
display values}
begin
  New_DisplayLevel:=DisplayLevel.Position;
  DoUpdate:=true;
end;

procedure TForm1.ReScaleChange(Sender: TObject);
{The rescale factor control has been changed, so re-validate display
values}
begin
  try
    New_ReScale:=ReScale.value;
  except                       {catch any invalid integer conditions}
    New_ReScale:=ReScaleFactor;
  end;
  DoUpdate:=true;
end;

procedure TForm1.X_RedClick(Sender: TObject);
{The x axis display colour has been changed, so set it to new colour}
begin
  New_AxisColours:='X';
  UpdateColours:=true;
  DoUpdate:=true;
end;

procedure TForm1.X_GreenClick(Sender: TObject);
{The x axis display colour has been changed, so set it to new colour}
begin
  New_AxisColours:='X';
  UpdateColours:=true;
  DoUpdate:=true;
end;

procedure TForm1.X_BlueClick(Sender: TObject);
{The x axis display colour has been changed, so set it to new colour}
begin
  New_AxisColours:='X';
  UpdateColours:=true;
  DoUpdate:=true;
end;

procedure TForm1.X_noneClick(Sender: TObject);
begin
  New_AxisColours:='X';
```

```
  UpdateColours:=true;
  DoUpdate:=true;
end;

procedure TForm1.Y_RedClick(Sender: TObject);
{The y axis display colour has been changed, so set it to new colour}
begin
  New_AxisColours:='Y';
  UpdateColours:=true;
  DoUpdate:=true;
end;

procedure TForm1.Y_GreenClick(Sender: TObject);
{The y axis display colour has been changed, so set it to new colour}
begin
  New_AxisColours:='Y';
  UpdateColours:=true;
  DoUpdate:=true;
end;

procedure TForm1.Y_BlueClick(Sender: TObject);
{The y axis display colour has been changed, so set it to new colour}
begin
  New_AxisColours:='Y';
  UpdateColours:=true;
  DoUpdate:=true;
end;

procedure TForm1.Y_noneClick(Sender: TObject);
begin
  New_AxisColours:='Y';
  UpdateColours:=true;
  DoUpdate:=true;
end;

procedure TForm1.Z_RedClick(Sender: TObject);
{The z axis display colour has been changed, so set it to new colour}
begin
  New_AxisColours:='Z';
  UpdateColours:=true;
  DoUpdate:=true;
end;

procedure TForm1.Z_GreenClick(Sender: TObject);
{The z axis display colour has been changed, so set it to new colour}
begin
  New_AxisColours:='Z';
  UpdateColours:=true;
  DoUpdate:=true;
end;

procedure TForm1.Z_BlueClick(Sender: TObject);
{The z axis display colour has been changed, so set it to new colour}
begin
  New_AxisColours:='Z';
```

```
  UpdateColours:=true;
  DoUpdate:=true;
end;

procedure TForm1.Z_noneClick(Sender: TObject);
begin
  New_AxisColours:='Z';
  UpdateColours:=true;
  DoUpdate:=true;
end;

procedure TForm1.GreyscaleButtonClick(Sender: TObject);
{The greyscale display button has been selected, so set global
variable}
begin
  if GreyscaleButton.Checked then
    New_ShowColour:=false else New_ShowColour:=true;
  DoUpdate:=true;
end;

procedure TForm1.ColourButtonClick(Sender: TObject);
{The colour gradient display button has been selected, so set global
variable}
begin
  if ColourButton.Checked then
    New_ShowColour:=true else New_ShowColour:=false;
  DoUpdate:=true;
end;

procedure TForm1.UpdateAxisColours(which: string);
{Ensures that the colour rectangles showing the primary colour
currently}
{being used to depict that axis, are correct. If ALL is passed as a
parameter}
{all the colour bitmaps are updated. Likewise if X,Y or Z are passed
then only}
{that axis colour is updated.}
begin
  if (which='ALL') or (which='X') then begin
    if X_Red.Checked then X_RGB:=RED;      {only one of these can be
true}
    if X_Green.Checked then X_RGB:=GREEN;
    if X_Blue.Checked then X_RGB:=BLUE;
    if X_none.Checked then X_RGB:=BLACK;
    with X_Colour.Picture do
      case X_RGB of                        {point to correct bitmap}
        RED:   Graphic:=BitmapRed;
        GREEN: Graphic:=BitmapGreen;
        BLUE:  Graphic:=BitmapBlue;
        BLACK: Graphic:=BitmapBlack;
      end;
  end;
  if (which='ALL') or (which='Y') then begin
    if Y_Red.Checked then Y_RGB:=RED;      {only one of these can be
true}
```

```
    if Y_Green.Checked then Y_RGB:=GREEN;
    if Y_Blue.Checked then Y_RGB:=BLUE;
    if Y_none.Checked then Y_RGB:=BLACK;
    with Y_Colour.Picture do
      case Y_RGB of                        {point to correct bitmap}
        RED:   Graphic:=BitmapRed;
        GREEN: Graphic:=BitmapGreen;
        BLUE:  Graphic:=BitmapBlue;
        BLACK: Graphic:=BitmapBlack;
      end;
  end;
  if (which='ALL') or (which='Z') then begin
    if Z_Red.Checked then Z_RGB:=RED;      {only one of these can be
true}
    if Z_Green.Checked then Z_RGB:=GREEN;
    if Z_Blue.Checked then Z_RGB:=BLUE;
    if Z_none.Checked then Z_RGB:=BLACK;
    with Z_Colour.Picture do
      case Z_RGB of                        {point to correct bitmap}
        RED:   Graphic:=BitmapRed;
        GREEN: Graphic:=BitmapGreen;
        BLUE:  Graphic:=BitmapBlue;
        BLACK: Graphic:=BitmapBlack;
      end;
  end;
end;

procedure TForm1.Auto1Click(Sender: TObject);
{The AutoScale at Start Only button has been selected}
begin
  New_AutoScale:=START;
  DoUpdate:=true;
end;

procedure TForm1.Auto2Click(Sender: TObject);
{The AutoScale Continual button has been selected}
begin
  New_AutoScale:=CONTINUAL;
  DoUpdate:=true;
end;

procedure TForm1.Auto3Click(Sender: TObject);
{The AutoScale Never button has been selected}
begin
  New_AutoScale:=NEVER;
  DoUpdate:=true;
end;

procedure TForm1.Auto_Scale(scr: smallint);  {Calculate scaling factor
for data display}
{If AutoScale is not set to NEVER, then the colours (RGB values) are
scaled}
{such that the grid point of maximum absolute value for the quantity
currently}
```

```
{being displayed is given the maximum colour value (255 or $FF). The
other }
{points are given colours determined by a linear transition from zero
to that}
{maximum value. The absolute values are used for determining the
colour, so}
{that for example: -100 and +100 will be displayed as the same colour}
const
  PixMax=$FF;
begin
  if AutoScale<> NEVER then begin     {only autoscale if option
selected}
    if (AutoScale=CONTINUAL) or ((AutoScale=START) and (Time=0)) then
begin
      FindMaxVal(scr,DisplayField);  {Find the Maximum value of the
field concerned}

      if MaxVal<>0 then begin
        New_ReScale:=PixMax/MaxVal;
      end
      else begin
        New_ReScale:=0;           {if all screen points have zero
value,}
      end;                        {reset default scaling control
positions}

      DoUpdate:=true;
    end;
  end;
end;

procedure TForm1.MaxCheck(element: PointPtr);
begin
  with element.Electric do          {for Electric field of current
grid point}
    if Max_E<>0 then begin          {if a maximum allowable value is
defined}
      if x>Max_E then x:=Max_E;     {restrict the point's values to
within this}
      if y>Max_E then y:=Max_E;     {range of values}
      if z>Max_E then z:=Max_E;
      if x<-Max_E then x:=-Max_E;
      if y<-Max_E then y:=-Max_E;
      if z<-Max_E then z:=-Max_E;
    end;
  with element.Magnetic do          {for Magnetic field of current
grid point}
    if Max_B<>0 then begin          {if a maximum allowable value is
defined}
      if x>Max_B then x:=Max_B;     {restrict the point's values to
within this}
      if y>Max_B then y:=Max_B;     {range of values}
      if z>Max_B then z:=Max_B;
      if x<-Max_B then x:=-Max_B;
      if y<-Max_B then y:=-Max_B;
```

```
      if z<-Max_B then z:=-Max_B;
    end;
end;

function TForm1.VectSize(vect: Vector): extended;
{Calculate the total vector size by combining the three component
vectors}
begin
  with vect do
    Result:=sqrt(sqr(x)+sqr(y)+sqr(z));
end;

function TForm1.E_Energy(E_amp: extended): extended;
{calculate the energy density in the electric field at the point with
amplitude E_amp }
begin
  Result:=0.5*Permittivity*sqr(E_amp);
end;

function TForm1.B_Energy(B_amp: extended): extended;
{calculate the energy density in the magnetic field at the point with
amplitude B_amp }
begin
  Result:=0.5*Permeability*sqr(B_amp);
end;

procedure TForm1.Vect_ArrowsClick(Sender: TObject);
{The Vector Arrows button has been changed, so either enable or
disable}
{all the vector option controls}
begin
  if Vect_Arrows.Checked then
    New_Arrows:=true
  else
    New_Arrows:=false;
  DoUpdate:=true;
end;

procedure TForm1.DrawArrow(ThisBitmap: Tbitmap;x,y: smallint;arrow:
Vector);
var      {draws a vector corresponding to the relevant x,y & z values}
         {x & y vectors are depicted as lines, z as crosses or
circles}
  xpos,ypos,zsize: smallint;
begin
  with ThisBitmap.Canvas do begin    {using the appropriate bitmap}
    with Pen do begin
      Width:=1;                        {set pen to thin white line}
      Color:=clWhite;
      Style:=psSolid;

      if VectorX.Checked then begin
        if (arrow.x > arrow_step) then begin
          arrow.x := arrow_step;
          Style:=psDot;
```

```
        end;

        if (arrow.x < -arrow_step) then begin
          arrow.x := -arrow_step;
          Style:=psDot;
        end;
      end;

      if VectorY.Checked then begin
        if (arrow.y > arrow_step) then begin
          arrow.y := arrow_step;
          Style:=psDot;
        end;

        if (arrow.y < -arrow_step) then begin
          arrow.y := -arrow_step;
          Style:=psDot;
        end;
      end;

      if VectorZ.Checked then begin
        if (arrow.z > arrow_step) then begin
          arrow.z := arrow_step;
          Style:=psDot;
        end;

        if (arrow.z < -arrow_step) then begin
          arrow.z := -arrow_step;
          Style:=psDot;
        end;
      end;
    end;

   MoveTo(x,y);                    {start the vector from the point
(x,y) }
    if VectorX.Checked then          {if x axis vectors required calc
xpos}
      xpos:=round2(x+(arrow.x))
    else xpos:=x;                    {else use x value of starting
point}
    if VectorY.Checked then          {if y axis vectors required calc
ypos}
      ypos:=round2(y+(arrow.y))
    else ypos:=y;                    {else use y value of starting
point}
    LineTo(xpos,Ypos);               {draw the vector arrow for the x
and/or y displacement}
    if VectorZ.Checked then begin    {if z axis vectors required,}
      zsize:=round2(arrow.z);        {calc vector size}
      if arrow.z>0 then begin        {if z>0 (coming out of the page)
draw circle of appropriate size}
        MoveTo(x,y);
        brush.Style:=bsClear;        {ensure circle is not filled}
        Ellipse(x-zsize,y-zsize,x+zsize,y+zsize);
      end;
```

```
      if arrow.z<0 then begin        {if z<0 (going into the page)
draw cross of appropriate size}
        MoveTo(x-zsize,y-zsize);
        LineTo(x+zsize-1,y+zsize-1); {slight correction added to fix
arrow's}
        MoveTo(x-zsize,y+zsize);     {look - make both sides look
even}
        LineTo(x+zsize-1,y-zsize+1);
      end;
    end;
  end;
end;

procedure TForm1.NewBitmap(BmapPtr: BitmapPtr);
{given a pointer to a bitmap, erase the bitmap it points to (if any) }
{and return the pointer pointing to a new blank 24bit bitmap with }
{width and height the same as the Image control}
begin
  if BmapPtr<>nil then BmapPtr.free;        {free memory of current
bitmap if req'd}
  BmapPtr^:=TBitmap.Create;                 {create a new bitmap}
  if BmapPtr<> nil then with BmapPtr^ do begin                 {using
the new bitmap's pointer}
    Height:=Image1.Height;                  {set bitmap height}
    Width:=Image1.Width;                    {set bitmap width}
    Canvas.Brush.Color:=clBlack;
    Canvas.FloodFill(1,1,clBlack,fsBorder); {ensure black filled}
    PixelFormat:=pf24bit;                   {set 24bit colour}
  end;
end;

procedure TForm1.PlotPoint(Bmap: TBitmap; x, y: smallint);
{Using the colour array built by UpdateBitmap, display the colour
information}
{of the point (x,y) either as: (a) a extended pixel. (b) a circle of
uniform }
{colour with diametre equal to the smallest of ScrScaleX & ScrScaleY.
}
{ (c) a rectangle of width ScrScaleX and height ScrScaleY with the
original }
{colour value in the centre, and a graduated change to match the other
grid }
{points around it - the colour graduation is provided by
VectorInterpolate }
{and uses a linear interpolation in the X and Y directions for each of
the }
{three colour planes (Red Green Blue).}
var
  ScreenX,ScreenY: extended;
  Edge,PointNum: byte;
  DoQuad1,DoQuad2,DoQuad3,DoQuad4: boolean;
  TileOfsX,TileOfsY: smallint;
  newx,newy: smallint;
  Signs: array[1..9,1..3] of shortint;
  Colour: TColor;
```

```
  XCol,Yrow: Longint;
  Xtop,Ytop,Xbot,Ybot: extended;
  PixelRender: smallint;
  DrawPoint: boolean;
begin
  TileOfsX:=0;
  TileOfsY:=0;
  DrawPoint:=true;
  PixelRender:=Render;
  Colour:=ColourArray[x,y];
  begin
    if TileZ then begin      {if the Z planes are required to be
tiled}
      Yrow:=ZplanePos div TileXcount;
      Xcol:=ZplanePos mod TileXcount;
      TileRect:=TileGrid.CellRect(Xcol,Yrow);
      with TileRect do begin
        TileOfsX:=Left+EdgeSize;  {add a bit bypass tile's border}
        TileOfsY:=Top+EdgeSize;
      end;
      TileHalfX:=TileScrScaleX/2;        {calc midway points}
      TileHalfY:=TileScrScaleY/2;
      ScreenX:=TileOfsX+(x*TileScrScaleX)+TileHalfX; {calc x coord for
pixel}
      ScreenY:=TileOfsY+(y*TileScrScaleY)+TileHalfY; {calc y coord for
pixel}
      if ((TileScrScaleX<=1) and (TileScrScaleY<=1)) then
        PixelRender:=OneToOne; {prevent uneccesary work}
     { if (GridDepth>50) and (PixelRender=Blend) then
PixelRender:=Chunky;  }
        {Blend takes too long if too many Z Planes}
      if not ((ScreenX<TileRect.Right) and (ScreenY<TileRect.Bottom))
then
        DrawPoint:=false;    {Don't draw pixel if outside tile's
space}
    end
    else begin
      ScreenX:=GetRealX((x+0.5));  {the actual screen x location (on
the image control)}
      ScreenY:=GetRealY((y+0.5));  {the actual screen y location (on
the image control)}
      if ((ScrScaleX<=1) and (ScrScaleY<=1)) then
        PixelRender:=OneToOne; {prevent uneccesary work}
    end;
  if DrawPoint then
    if (PixelRender=OneToOne) then  {Display 1 point as one pixel}
      PlotPixel(Round2(ScreenX),Round2(ScreenY),Colour)
    else if PixelRender=Chunky then with Bmap.Canvas do begin  {if
'chunky' pixels of uniform colour req'd}
      if TileZ then begin
        Xtop:=ScreenX-TileHalfX;
        Ytop:=ScreenY-TileHalfY;
        Xbot:=Xtop+TileScrScaleX+1;
        Ybot:=Ytop+TileScrScaleY+1;
        if Xtop<TileOfsX then Xtop:=TileOfsX;
```

```
          if Xbot>=TileRect.Right then Xbot:=TileRect.Right-1;
          if Ytop<TileOfsY then Ytop:=TileOfsY;
          if Ybot<=TileRect.Bottom then Ybot:=TileRect.Bottom-1;
        end
        else begin
          Ytop:=ScreenY-halfY-1;
          Xtop:=ScreenX-halfX-1;
          Xbot:=Xtop+ScrScaleX+1;
          Ybot:=Ytop+ScrScaleY+1;
        end;
        Brush.Color:=Colour;

FillRect(rect(Round2(Xtop),Round2(Ytop),Round2(Xbot),Round2(Ybot)));
      end
      else if PixelRender=Blend then begin

{The colour graduation for the rectangle depicting the grid point in
question}
{is determined by examining the 8 grid points surrounding it and
noting the }
{Red, Green, and Blue values for each (and the point itself). A linear
}
{interpolation is then carried out in each quadrant around the central
point}
{The points are numbered as:      1   2   3
}
{                                 4   5   6
}
{                                 7   8   9
}

{Fill ColArray with combined RGB value for each point}
        Edge:=EdgeCase(x,y,0,0,GridWidth-1,GridHeight-1);  {determine
point's location}
        if EdgeArray[Edge,1,1]=1 then begin             {if not at top
left}
          ColArray[1]:=ColourArray[x-1,y-1];
          Signs[1,1]:=SignArray[x-1,y-1,0];
          Signs[1,2]:=SignArray[x-1,y-1,1];
          Signs[1,3]:=SignArray[x-1,y-1,2];
        end
        else ColArray[1]:=0;

        if EdgeArray[Edge,1,2]=1 then begin             {if not at top}
          ColArray[2]:=ColourArray[x,y-1];
          Signs[2,1]:=SignArray[x,y-1,0];
          Signs[2,2]:=SignArray[x,y-1,1];
          Signs[2,3]:=SignArray[x,y-1,2];
        end
        else ColArray[2]:=0;

        if EdgeArray[Edge,1,3]=1 then begin             {if not at top
right}
          ColArray[3]:=ColourArray[x+1,y-1];
          Signs[3,1]:=SignArray[x+1,y-1,0];
```

```
        Signs[3,2]:=SignArray[x+1,y-1,1];
        Signs[3,3]:=SignArray[x+1,y-1,2];
      end
      else ColArray[3]:=0;

      if EdgeArray[Edge,2,1]=1 then begin            {if not at left}
        ColArray[4]:=ColourArray[x-1,y];
        Signs[4,1]:=SignArray[x-1,y,0];
        Signs[4,2]:=SignArray[x-1,y,1];
        Signs[4,3]:=SignArray[x-1,y,2];
      end
      else ColArray[4]:=0;

      ColArray[5]:=ColourArray[x,y];                {refers to point
itself}
      Signs[5,1]:=SignArray[x,y,0];
      Signs[5,2]:=SignArray[x,y,1];
      Signs[5,3]:=SignArray[x,y,2];

      if EdgeArray[Edge,2,3]=1 then begin            {if not at
right}
        ColArray[6]:=ColourArray[x+1,y];
        Signs[6,1]:=SignArray[x+1,y,0];
        Signs[6,2]:=SignArray[x+1,y,1];
        Signs[6,3]:=SignArray[x+1,y,2];
      end
      else ColArray[6]:=0;

      if EdgeArray[Edge,3,1]=1 then begin            {if not at
bottom left}
        ColArray[7]:=ColourArray[x-1,y+1];
        Signs[7,1]:=SignArray[x-1,y+1,0];
        Signs[7,2]:=SignArray[x-1,y+1,1];
        Signs[7,3]:=SignArray[x-1,y+1,2];
      end
      else ColArray[7]:=0;

      if EdgeArray[Edge,3,2]=1 then begin            {if not at
bottom}
        ColArray[8]:=ColourArray[x,y+1];
        Signs[8,1]:=SignArray[x,y+1,0];
        Signs[8,2]:=SignArray[x,y+1,1];
        Signs[8,3]:=SignArray[x,y+1,2];
      end
      else ColArray[8]:=0;

      if EdgeArray[Edge,3,3]=1 then begin            {if not at
bottom right}
        ColArray[9]:=ColourArray[x+1,y+1];
        Signs[9,1]:=SignArray[x+1,y+1,0];
        Signs[9,2]:=SignArray[x+1,y+1,1];
        Signs[9,3]:=SignArray[x+1,y+1,2];
      end
      else ColArray[9]:=0;
```

```
{Determine if it is worth doing calculation for each quadrant. If all
four points}
{which define the quadrant have a zero value, nothing need be done.}

DoQuad1:=not ((ColArray[1]=0) and (ColArray[2]=0) and (ColArray[4]=0)
and (ColArray[5]=0));
DoQuad2:=not ((ColArray[2]=0) and (ColArray[3]=0) and (ColArray[5]=0)
and (ColArray[6]=0));
DoQuad3:=not ((ColArray[4]=0) and (ColArray[5]=0) and (ColArray[7]=0)
and (ColArray[8]=0));
DoQuad4:=not ((ColArray[5]=0) and (ColArray[6]=0) and (ColArray[8]=0)
and (ColArray[9]=0));


   {   if TileZ then begin          {prevent Tile over-run}
   {     if EdgeArray[Edge,1,1]=0 then DoQuad1:=false;
        if EdgeArray[Edge,1,3]=0 then DoQuad2:=false;
        if EdgeArray[Edge,3,1]=0 then DoQuad3:=false;
        if EdgeArray[Edge,3,3]=0 then DoQuad4:=false;
      end;
{Process ColArray to provide signed R,G & B values for each point}
{each R,G or B value lies in the range -255 to +255 due to the
signing. }
{Once the interpolation has been done, the results can then be turned
into}
{an absolute value to give all positive values once more. The reason
for }
{this process is to get the correct gradient between points of equal
absolute}
{value but of different signs.}

      if (DoQuad1 or DoQuad2 or DoQuad3 or DoQuad4) then begin
        for PointNum:=1 to 9 do   {determine RGB values for each of
the 9 points}
          with PntArray[PointNum] do begin
            x:=abs(Signs[PointNum,1])*RGB_Val(ColArray[PointNum],RED);
{set x to point's Red value}

y:=abs(Signs[PointNum,2])*RGB_Val(ColArray[PointNum],GREEN);  {set y
to point's Green value}

z:=abs(Signs[PointNum,3])*RGB_Val(ColArray[PointNum],BLUE);   {set z
to point's Blue value}
          end;


{Call the routine which calculates and draws to screen the graduated
quadrants}
        if DoQuad1 then PlotQuadrant(Bmap,1,ScreenX,ScreenY);
        if DoQuad2 then PlotQuadrant(Bmap,2,ScreenX,ScreenY);
        if DoQuad3 then PlotQuadrant(Bmap,3,ScreenX,ScreenY);
        if DoQuad4 then PlotQuadrant(Bmap,4,ScreenX,ScreenY);
      end;
    end;
```

```
  end;
end;

function TForm1.RGB_Val(colour: Tcolor; primary: integer): byte;
{Given a combined RGB value (3 bytes of type TColor) and a parameter}
{determining which colour is required, this function returns the }
{byte value of that colour component.}
var
  Mask: integer;
begin
  Mask:=0;
  case primary of           {select appropriate mask value}
    RED:    Mask:=RedMask;
    GREEN:  Mask:=GreenMask;
    BLUE:   Mask:=BlueMask;
  end;
  if primary<>0 then
    Result:=round((colour and Mask)/primary)  {mask out unwanted
colours}
  else
    Result:=0;
end;

function TForm1.EdgeCase(x, y, Xmin, Ymin, Xmax, Ymax: smallint):
byte;
     {determines if the point (x,y) lies at the edge of the rectangle
}
     {defined by (Xmin,Ymin) and (Xmax,Ymax), and if so, what sort of
}
     {edge is it on (ie. left, right, bottom left corner etc...).
This}
     {information is required during calculations which require
comparisons}
     {between adjacent grid points, where such points may lie out of
the }
     {grid's bounds. The case values correspond to the following
locations:}
     {       (top left)  1   2   3
}
     {                   4   5   6
}
     {                   7   8   9  (bottom right)
}
var
  Xcase,Ecase: byte;
begin
   Ecase:=0;
   if x=Xmin then Xcase:=1        {calc X coord boundary state}
   else if x=Xmax then  Xcase:=3  {1=left boundary, 3=right boundary}
   else Xcase:=2;

   if y=Ymin then        {calc overall boundary state using y and
xcase}
     case Xcase of       {if y is at top boundary}
       1: Ecase:=1;        {x at left boundary}
```

```
        2: Ecase:=2;         {x not at a boundary}
        3: Ecase:=3;         {x at right boundary}
      end
    else if y=Ymax then    {if y is at bottom boundary}
      case Xcase of
        1: Ecase:=7;         {x at left boundary}
        2: Ecase:=8;         {x not at a boundary}
        3: Ecase:=9;         {x at right boundary}
      end
    else
      case Xcase of        {if y is not at a boundary}
        1: Ecase:=4;         {x at left boundary}
        2: Ecase:=5;         {x not at a boundary}
        3: Ecase:=6;         {x at right boundary}
      end;
  Result:=Ecase;           {report finding}
end;

function TForm1.VectorInterpolate(v1, v2, v3, v4: Vector; Xfrac,
  Yfrac: extended): Vector;
var
  NewVect: Vector;
begin
  NewVect.x:=Interpolate(v1.x,v2.x,v3.x,v4.x,Xfrac,Yfrac);
  NewVect.y:=Interpolate(v1.y,v2.y,v3.y,v4.y,Xfrac,Yfrac);
  NewVect.z:=Interpolate(v1.z,v2.z,v3.z,v4.z,Xfrac,Yfrac);
  Result:=NewVect;
end;

function TForm1.Interpolate(val1, val2, val3, val4, Xfrac,
  Yfrac: extended): extended;
{provide a value for a point somewhere with four other points which
define}
{the corners of a rectangle. The position of the point in question is
given as}
{a fraction along the x axis and a fraction along the y axis}
{The value is calculated by linear interpolation along each axis
direction}
var
  Xgrad,Xgrad1,Xgrad2,Ygrad2,newval: extended;
begin
  Xgrad1:=val2-val1;   {gradient along top of rectangle}
  Xgrad2:=val4-val3;   {gradient along bottom of rectangle}
  Ygrad2:=val3-val1;   {gradient along left of rectangle}
  Xgrad:=Xgrad1+Yfrac*(Xgrad2-Xgrad1);   {how x axis gradient varies
with y}
  newval:=val1+(Yfrac*Ygrad2)+(Xfrac*Xgrad); {top-left corner value
plus }
  Result:=newval;      {contributions from travel down y axis and
across x axis}
end;

procedure TForm1.PlotQuadrant(Bmap: TBitmap; Quadrant: byte; RealX,
RealY: extended);
```

```
{Use the PntArray configured by PlotPoint and the supplied variables
such as }
{Quadrant number (1,2,3 or 4) and actual (x,y) location for grid
point. The }
{four points defining the quadrant are sent to VectorInterpolate to
determine}
{the new RGB value for each pixel in the quadrant which is to be
plotted.}
{Note: only a quarter of the area defined by the quadrant is actually
}
{calculated and drawn, as adjacent points will do the other three
quarters}
{when they are calculated.}
var
  xstart,ystart,xend,yend,i,j: smallint;
  v1,v2,v3,v4,pix: Vector;
  colour: Tcolor;
  Xofs,Yofs: extended;
  Rval,Gval,Bval: byte;
  Xpos,Ypos: smallint;
  Width,Height,HalfWidth,HalfHeight: extended;
begin
  Xofs:=0;
  Yofs:=0;
  xstart:=0;
  ystart:=0;
  xend:=0;
  yend:=0;
  if TileZ then begin
    HalfWidth:=TileHalfX;
    HalfHeight:=TileHalfY;
    Width:=TileScrScaleX;
    Height:=TileScrScaleX;
  end
  else begin
    HalfWidth:=halfX;
    HalfHeight:=halfY;
    Width:=ScrScaleX;
    Height:=ScrScaleY;
  end;
{Determine boundary conditions for relevant Quadrant. The origin of
the four}
{Quadrants is condidered to be point (0,0) for the purposes of the
calculation}
  case Quadrant of                {Quadrant 1}
    1: begin                      {x<=0, y<=0}
      xstart:=-Trunc(HalfWidth)-1;
      ystart:=-Trunc(HalfHeight)-1;
      xend:=0;
      yend:=0;
      Xofs:=Width;
      Yofs:=Height;
    end;                          {Quadrant 2}
    2: begin                      {x>0, y<=0}
      xstart:=0;
```

```
      ystart:=-Trunc(HalfHeight)-1;
      xend:=Trunc(Width-HalfWidth)+1;
      yend:=0;
      Xofs:=0;
      Yofs:=Height;
    end;                          {Quadrant 3}
    3: begin                      {x<=0, y>0}
      xstart:=-Trunc(HalfWidth)-1;
      ystart:=0;
      xend:=0;
      yend:=Trunc(Height-HalfHeight)+1;
      Xofs:=Width;
      Yofs:=0;
    end;                          {Quadrant 4}
    4: begin                      {x>0, y>0}
      xstart:=0;
      ystart:=0;
      xend:=Trunc(Width-HalfWidth)+1;
      yend:=Trunc(Height-HalfHeight)+1;
      Xofs:=0;
      Yofs:=0;
    end;
  end;

  if TileZ then with TileRect do begin   {Trim edges, so it fits 
exactly}
    if (RealX+xstart)<=(Left+EdgeSize) then 
xstart:=Round2(Left+EdgeSize-RealX);
    if (RealX+xend)>=(Right-1) then xend:=Round2(Right-1-RealX);
    if (RealY+ystart)<=(Top+EdgeSize) then 
ystart:=Round2(Top+EdgeSize-RealY);
    if (RealY+yend)>=(Bottom-1) then yend:=Round2(Bottom-1-RealY);
  end;

  if Quadrant>2 then Inc(Quadrant); {Algorithm for selecting the 
appropriate points}
  v1:=PntArray[Quadrant];
  v2:=PntArray[Quadrant+1];
  v3:=PntArray[Quadrant+3];
  v4:=PntArray[Quadrant+4];

  for j:=ystart to yend do begin {scan each horizontal line in the 
Quadrant}
    Ypos:=Round2(RealY+j);       {Determine actual y coord for point 
in Quadrant}
    for i:=xstart to xend do begin {scan along horizontal line in 
Quadrant}
      {do the linear interpolation for each colour of point in 
Quadrant}

pix:=VectorInterpolate(v1,v2,v3,v4,((i+Xofs)/Width),((j+Yofs)/Height))
;
      with pix do begin
        Rval:=byte(Round2(abs(x)));    {convert real values to 
integers}
```

```
        Gval:=byte(Round2(abs(y)));    {round values to nearest
integer (down if 0.5)}
        Bval:=byte(Round2(abs(z)));
      end;
      {if greyscale display is required, the R,G & B values are all
equalized}
      {prior to building the ColourArray and calling the PlotPoint and
}
      {PlotQuadrant routines, so the resulting colour is guaranteed to
also}
      {be a greyscale.}

      Colour:=Tcolor((Rval*RED) + (Gval*GREEN) + (Bval*BLUE));
{combine values to give one RGB value}
      Xpos:=Round2(RealX+i);   {calc x coord of the point on the Image
control}
      PlotPixel(Xpos,Ypos,Colour);
    end;
  end;
end;

function TForm1.GetActualX(x: smallint): smallint;
{return the actual screen x coordinate (in the picture control) of the
x }
{coord of a point in the grid (assumes bitmap to be displayed at full
size)}
begin
  Result:=round2(GetRealX(x)); {ScrScaleX=number of pixels b/w points}
end;

function TForm1.GetActualY(y: smallint): smallint;
{return the actual screen y coordinate (in the picture control) of the
y }
{coord of a point in the grid (assumes bitmap to be displayed at full
size)}
begin
  Result:=round2(GetRealY(y)); {ScrScaleY=number of pixels b/w points}
end;

procedure TForm1.VectorSpacingChange(Sender: TObject);
{Redraw screen with different arrow spacing, unless Z plane tiling is
}
{on in which case arrows are not shown anyhow.}
var
  step: integer;
begin
  step:=VectSpacing;

  try
    step:=StrToInt(VectorSpacing.Text);
  except                        {catch any invalid integer conditions}
    VectorSpacing.Text:='';
  end;

  New_VectSpacing:=step;
```

```
  if not TileZ then DoUpdate:=true;
end;

procedure TForm1.SetTileSize;
{If the Z Plane tiling option is selected, each of the Z coordinate
X/Y planes}
{is to be displayed simultaneously on the Image control (screen). So,
as the }
{dimensions of the grid (including the number of Z planes) can vary,
the }
{optimum tile size must be calculated so that all planes can be
displayed }
{with their correct aspect ratios and at the largest possible tile
size}
{Note: A tile is a small version of one of the X/Y planes which are
normally}
{      displayed as a full screen image.}
var
  nx,ny: smallint;
  height: integer;
  TileAspect: extended;
begin
    if MaintainAspect then
      TileAspect:=Aspect
    else
      TileAspect:=ScreenAspect;

    TileX:=(BitmapX-EdgeSize*2);  {start with one tile the width of
the screen}
    repeat                        {TileX will be width of tile}
      nx:=(BitmapX-EdgeSize*2) div TileX;      {nx is the number of
tiles across the screen}
      TileY:=Trunc(TileAspect*TileX); {TileY will be height of tile}
      ny:=GridDepth div nx;       {ny is number of rows of tiles down
the screen}
      if (GridDepth mod nx)<>0 then Inc(ny);
      height:=ny*TileY;           {total height used by tiles}
      Dec(TileX);                 {Try smaller tile size}
{height must fit screen. Don't allow vanishingly small. Don't exceed
max Z plane}
    until (height<=(BitmapY-EdgeSize*2)) or (TileX<10) or
(nx>GridDepth);
    Inc(TileX);       {reverse the last decrement}
    if Render=OneToOne then begin    {If one to one representation
required}
      if TileX>GridWidth then begin  {restrict max tile size so that}
        TileX:=GridWidth;            {one grid point becomes one
pixel}
        TileY:=Trunc(TileAspect*TileX);
      end;
      if TileY>GridHeight then begin
        TileY:=GridHeight;
        TileX:=Trunc(TileY/TileAspect);
      end;
```

```
    end;
    TileXcount:=(BitmapX-EdgeSize*2) div TileX;  {determine number
across...}
    TileYcount:=GridDepth div TileXcount; {and down screen.}
    if GridDepth mod TileXcount<>0 then Inc(TileYcount); {add one for
an incmplete row}
    if MaintainAspect then begin
      TileScrScaleX:=(TileX-EdgeSize)/GridWidth;  {calc scaling
factors for reducing full}
      TileScrScaleY:=(TileY-EdgeSize)/GridHeight; {screen to tile size
(allow for border)}
    end
    else begin
      TileScrScaleX:=((TileX-EdgeSize)/BitmapX)*ScrScaleX;  {calc
scaling factors for reducing full}
      TileScrScaleY:=((TileY-EdgeSize)/BitmapY)*ScrScaleY;  {screen to
tile size (allow for border)}
    end;
    with TileGrid do begin
      ColCount:=TileXcount;
      RowCount:=TileYcount;
      DefaultColWidth:=TileX;
      DefaultRowHeight:=TileY;
    end;
end;

function TForm1.Round2(realval: extended): int64;
{This function provided an equal, consistent rounding of real numbers}
{such that they are always rounded to the nearest integer, and when }
{exactly 0.5 between integers, they are rounded towards zero.}
{The standard Round function provided rounds to the EVEN integer!!}
var
  intval: int64;
  fraction: extended;
begin
  intval:=Trunc(realval);           {first round real value towards
zero}
  fraction:=(realval-intval);       {subtract this result from the
real}
  if fraction>0.5 then Inc(intval)  {if the fraction>0.5, add one to
intval}
  else if fraction<-0.5 then Dec(intval); {cater for negative
fractions too}
  Result:=intval;                   {return the integer value
obtained}
end;

function TForm1.GetRealX(x: extended): extended;
{performs same function as GetActualX but returns a real value rather
than }
{an integer. This is so culuative rounding errors don't occur in some
calcs}
begin
  Result:=OriginX+(x*ScrScaleX);
end;
```

```
function TForm1.GetRealY(y: extended): extended;
{performs same function as GetActualY but returns a real value rather
than }
{an integer. This is so culuiative rounding errors don't occur in some
calcs}
begin
  Result:=OriginY+(y*ScrScaleY);
end;

function TForm1.ColourRange(value: extended; ScaleFactor: extended):
byte;
{Convert a real value to a colour value (between 0 and 255)}
var
  ColourVal: smallint;
begin
  ColourVal:=abs(ByteLimit(value*ScaleFactor));
  Result:=byte(ColourVal);
end;

function TForm1.VectToColours(vect: vector; ScaleFactor: extended):
vector;
{Takes a vector of real numbers and converts it to a vector of colour}
{values (one for each component: x,y & z). Negative values become }
{positive colours.}
var
  ColourVect: Vector;
begin
  with ColourVect do begin
    x:=ColourRange(vect.x,ScaleFactor); {convert component values to
colour values}
    y:=ColourRange(vect.y,ScaleFactor);
    z:=ColourRange(vect.z,ScaleFactor);
  end;
  Result:=ColourVect;       {return a vector of colour values}
end;


function TForm1.ByteLimit(value: extended): smallint;
{Take a real value and return it as a smallint value between the
limits}
{-255 and +255}
var
  newval: smallint;
begin
  if abs(value)>$FF then        {if out of range, clip it at the
limit}
    newval:=Sign(value)*$FF
  else newval:=round2(value);
  Result:=newval;               {return as a smallint}
end;

function TForm1.VectByteLimit(vect: Vector): Vector;
{Process each component vector in vect such that the resulting
vector's}
```

```
{component vectors are all in the range -255 to 255}
var
  ResultVect: Vector;
begin
  with ResultVect do begin
    x:=ByteLimit(vect.x);
    y:=ByteLimit(vect.y);
    z:=ByteLimit(vect.z);
  end;
  Result:=ResultVect;
end;

function TForm1.VectorCross(v1, v2: Vector): Vector;
var
  ResultVect: Vector;
begin
  with ResultVect do begin
    x:=((v1.y)*(v2.z) - (v1.z)*(v2.y));
    y:=((v1.z)*(v2.x) - (v1.x)*(v2.z));
    z:=((v1.x)*(v2.y) - (v1.y)*(v2.x));
  end;
  Result:=ResultVect;
end;

function TForm1.PowerFlow(Apoint: point): vector;
{Calculate the Power flow vector at a point in the grid by taking }
{the vector cross product of the Electric & Magnetic fields, and  }
{re-scaling the result by the area of a grid point.}
var
  vect: Vector;
begin
  with Apoint do
    vect:=VectorCross(Electric,Magnetic);
  with vect do begin
    x:=x*PointArea;         {Rescale values by area of grid point}
    y:=y*PointArea;
    z:=z*PointArea;
  end;
  Result:=vect;
end;

function TForm1.VectorProperty(field: byte; Apoint: point): Vector;
{Return a vector quantity of the field which is required to be
displayed}
{using the Electric and Magnetic field vectors at a grid point.}
var
  Vect: Vector;
begin
  Vect:=NullVect;
  with Apoint do
    case Field of   {depending on which field is required to display}
      1,3: Vect:=Electric;                    {Show Electric Field}
      2,4: Vect:=Magnetic;                    {Show Magnetic Field}
      5: Vect:=PowerFlow(Apoint);             {Show Power flow}
    end;
```

```
  Result:=Vect;
end;

function TForm1.PointNull(Apoint: point): boolean;
{See if the point's information is all zeros}
begin
  with Apoint do
    Result:=(VectorNull(Electric) and VectorNull(Magnetic));
end;

function TForm1.VectorNull(vect: Vector): boolean;
{Look at each component vector to see if the vector is overall a}
{Null vector.}
begin
  with vect do
    if (x=0) and (y=0) and (z=0) then
      Result:=true else Result:=false;
end;

function TForm1.ReverseTColor(input: TColor): Tcolor;
{Function reverses the byte order of a TColor type. This is required}
{prior to storing in bitmap's memory using the pointers provided by}
{the ScanLine function.}
var
  Byte1,Byte2,Byte3: byte;
begin
  if input<>0 then begin
      Byte1:=PByteArray(@input)[0];
      Byte2:=PByteArray(@input)[1];
      Byte3:=PByteArray(@input)[2];
      Result:=Byte3*RED + Byte2*Green + Byte1*Blue;
    end
    else Result:=0;
end;

procedure TForm1.Z_TilingClick(Sender: TObject);
begin
   if TileZ then New_TileZ:=false else New_TileZ:=true;
   DoUpdate:=true;
end;

function TForm1.MouseZplane: smallint;
{If Z plane tiling is being used, return the tile number which}
{the mouse cursor is over. Zero returned if not over a tile.}
var
  x,y: longint;
  Coord: TPoint;
  Coord2: TGridCoord;
  Tile: smallint;
begin
  if TileZ then  begin
    with Form1 do begin
      x:=left+MainGroup.left+TileGrid.left+3;
      y:=top+(Height-ClientHeight)+MainGroup.top+TileGrid.top-3;
    end;
```

```
    Coord:=MyMouse.CursorPos;
    Coord2:=TileGrid.MouseCoord(Coord.x-x,Coord.y-y);
    Tile:=Coord2.x+Coord2.y*TileXcount;
    if (Tile>GridDepth-1) or (Tile<0) then Tile:=-1;       {ensure in
range}
    Result:=Tile;
  end
  else
    Result:=-1;
end;

procedure TForm1.Image1Click(Sender: TObject);
var
  Tile: smallint;
begin
  Tile:=MouseZplane;
  if Tile>=0 then New_ZPlane:=Tile;
  DoUpdate:=true;
end;

procedure TForm1.Image1DblClick(Sender: TObject);
var
  Tile: smallint;
begin
  Tile:=MouseZplane;
  if Tile>=0 then New_ZPlane:=Tile;
  New_TileZ:=false;
  DoUpdate:=true;
end;

procedure TForm1.VectorXClick(Sender: TObject);
begin
  VectorChange:=true;
  DoUpdate:=true;
end;

procedure TForm1.VectorYClick(Sender: TObject);
begin
  VectorChange:=true;
  DoUpdate:=true;
end;

procedure TForm1.VectorZClick(Sender: TObject);
begin
  VectorChange:=true;
  DoUpdate:=true;
end;

procedure TForm1.TileCursor(Bmap: TBitmap; Tile: smallint; colour:
TColor);
{Draw a coloured rectangle around the selected Z Plane Tile}
var
  Xcol,Yrow: Longint;
begin
  Yrow:=Tile div TileXcount;
```

```
  Xcol:=Tile mod TileXcount;
  with Bmap.Canvas do begin
    brush.Color:=colour;
    FrameRect(TileGrid.CellRect(Xcol,Yrow));
  end;
end;

procedure TForm1.Spacing_pixelsClick(Sender: TObject);
{Redraw screen with different arrow spacing, unless Z plane tiling is
}
{on in which case arrows are not shown anyhow.}
begin
  if not TileZ then begin
    ArrowsUnitsChange:=true;
    DoUpdate:=true;
  end;
end;

procedure TForm1.Spacing_metresClick(Sender: TObject);
{Redraw screen with different arrow spacing, unless Z plane tiling is
}
{on in which case arrows are not shown anyhow.}
begin
  if not TileZ then begin
    ArrowsUnitsChange:=true;
    DoUpdate:=true;
  end;
end;

procedure TForm1.Spacing_gridpointsClick(Sender: TObject);
{Redraw screen with different arrow spacing, unless Z plane tiling is
}
{on in which case arrows are not shown anyhow.}
begin
  if not TileZ then begin
    ArrowsUnitsChange:=true;
    DoUpdate:=true;
  end;
end;

procedure TForm1.PlotPixel(x, y: smallint; Colour: TColor);
{Using the current array of bitmap's horizontal line pointers, plot}
{a point of a certain 24bitr colour value into memory. The colour
must}
{by reversed prior to storage as the values are stored in byte
reversed}
{order.}
var
  YLine: PByteArray;
begin
  if (x>0) and (y>0) and (x<=BitmapX) and (y<=BitmapY) then begin
    Yline:=YLinePtrs[y-1];    {find array of colour values for line y
in bitmap}
    TColorPtr(@Yline[3*(x-1)])^:=ReverseTColor(Colour);    {get
pointer to 24bit RGB value for point (x,y) }
```

```
  end;
end;

procedure TForm1.RenderOption1Click(Sender: TObject);
begin
  if RenderOption1.Checked then begin
    New_Render:=OneToOne;
    DoUpdate:=true;
  end;
end;

procedure TForm1.RenderOption2Click(Sender: TObject);
begin
  if RenderOption2.Checked then begin
    New_Render:=Chunky;
    DoUpdate:=true;
  end;
end;

procedure TForm1.RenderOption3Click(Sender: TObject);
begin
  if RenderOption3.Checked then begin
    New_Render:=Blend;
    DoUpdate:=true;
  end;
end;

procedure TForm1.RendDisplayClick(Sender: TObject);
begin
  if RendDisplay.Checked then
    New_Rendered:=true
  else
    New_Rendered:=false;
  DoUpdate:=true;
end;

procedure TForm1.UpdateE_Energy(scr: smallint);
var
  i,j: smallint;
begin
  E_Energy_Tot:=0;
  for i:=0 to GridWidth-1 do
    for j:=0 to GridHeight-1 do with points[scr,i,j,Z_Plane] do
      E_Energy_Tot:=E_energy_Tot+E_Energy(VectSize(Electric));
  E_Energy_Tot:=E_Energy_Tot*PointVolume; {adjust for unit volume}
end;

procedure TForm1.UpdateB_Energy(scr: smallint);
var
  i,j: smallint;
begin
  B_Energy_Tot:=0;
  for i:=0 to GridWidth-1 do
    for j:=0 to GridHeight-1 do with points[scr,i,j,Z_Plane] do
      B_Energy_Tot:=B_Energy_Tot+B_Energy(VectSize(Magnetic));
```

```
  B_Energy_Tot:=B_Energy_Tot*PointVolume; {adjust for unit volume}
end;

procedure TForm1.AspectControlClick(Sender: TObject);
begin
  if AspectControl.Checked then New_MaintainAspect:=true
  else New_MaintainAspect:=false;
  DoUpdate:=true;
end;

procedure TForm1.SetAspectRatio;
begin
  Aspect:=GridHeight/GridWidth;   {Aspect ratio of X/Y plane in grid}
  ScrScaleX:=BitmapX/GridWidth;   {how many times does grid width fit
screen?}
  ScrScaleY:=BitmapY/GridHeight;  {how many times does grid height fit
screen?}
  if MaintainAspect then          {if grid aspect ratio is to be
preserved,}
    if ScreenAspect>Aspect then   {then adjust scaling values to
allow}
      ScrScaleY:=ScrScaleX*Aspect {the largest image with the same
aspect ratio}
    else
      ScrScaleX:=ScrScaleY/Aspect;
  halfX:=ScrScaleX/2; {determine the number of pixels from one point}
  halfY:=ScrScaleY/2; {to half way to the next point (in x & y
directions)}
  {Determine bitmap coord's of the Origin (top left) of the }
  {active area of the screen (picture control) where the image will
start}
  OriginX:=Round2((BitmapX-(GridWidth*ScrScaleX))/2);
  OriginY:=Round2((BitmapY-(GridHeight*ScrScaleY))/2);
end;

procedure TForm1.GridXChange(Sender: TObject);
begin
  try
    StrToInt(GridX.Text);
  except                       {catch any invalid integer conditions}
    GridX.Text:='';
  end;
end;

procedure TForm1.GridYChange(Sender: TObject);
begin
  try
    StrToInt(GridY.Text);
  except                       {catch any invalid integer conditions}
    GridY.Text:='';
  end;
end;

procedure TForm1.GridZChange(Sender: TObject);
begin
```

```
  try
    StrToInt(GridZ.Text);
  except                       {catch any invalid integer conditions}
    GridZ.Text:='';
  end;
end;

procedure TForm1.AcceptGridSizeClick(Sender: TObject);
{Read the new Grid dimensions and change them to their new values}
{if they are sensible.}
begin
  try
    New_GridWidth:=StrToInt(GridX.Text);
    if New_GridWidth<3 then begin     {Keep it in range}
      GridX.Text:='3';
      New_GridWidth:=3;
    end;
  except                     {catch any invalid integer conditions}
    New_GridWidth:=GridWidth;
  end;
  try
    New_GridHeight:=StrToInt(GridY.Text);
    if New_GridHeight<3 then begin     {Keep it in range}
      GridY.Text:='3';
      New_GridHeight:=3;
    end;
  except                     {catch any invalid integer conditions}
    New_GridHeight:=GridHeight;
  end;
  try
    New_GridDepth:=StrToInt(GridZ.Text);
    if New_GridDepth<3 then begin     {Keep it in range}
      GridZ.Text:='3';
      New_GridDepth:=3;
    end;
  except                     {catch any invalid integer conditions}
    New_GridDepth:=GridDepth;
  end;
  DoUpdate:=true;
end;

procedure TForm1.SetGridGlobals;
{Set the values of various global variables which are dependent on the
Grid}
{size chosen.}
begin
  ActualWidth:=GridWidth/PPM;   {Actual width the screen models - in
metres}
  ActualHeight:=GridHeight/PPM;  {Actual Height the screen models - in
metres}
  ActualDepth:=GridDepth/PPM;    {Actual Depth the screen models - in
metres}
  PointArea:=(ActualWidth*ActualHeight)/(GridWidth*GridHeight); {Area
each pixel represents}
```

```
  PointVolume:=PointArea*ActualDepth/GridDepth; {volume each pixel
represents}
  ZPlane.Max:=GridDepth;           {set the ZPlane control's max
position}
  New_ZPlane:=round(GridDepth/2);  {default start point is mid point
in Z axis}
  LastZ.Caption:=IntToStr(GridDepth);  {show max Z plane value under
ZPlane control}
  WavePeriod2:=GridWidth/(2*WaveNumber+2); {Number of pixels for wave
period}
end;

function GetLargestFreeMemRegion(var AAddressOfLargest: pointer):
LongWord;
var
  Si: TSystemInfo;
  P, dwRet: LongWord;
  Mbi: TMemoryBasicInformation;
begin
  Result := 0;
  AAddressOfLargest := nil;
  GetSystemInfo(Si);
  P := 0;
  while P < LongWord(Si.lpMaximumApplicationAddress) do begin
    dwRet := VirtualQuery(pointer(P), Mbi, SizeOf(Mbi));
    if (dwRet > 0) and (Mbi.State and MEM_FREE <> 0) then begin
      if Result < Mbi.RegionSize then begin
        Result := Mbi.RegionSize;
        AAddressOfLargest := Mbi.BaseAddress;
      end;
      Inc(P, Mbi.RegionSize);
    end else
      Inc(P, Si.dwPageSize);
  end;
end;

procedure TForm1.ReAllocGridMemory;
{Try and allocate memory for the Grid of data points. If successful}
{Adopt the new Dimensions. Only newly allocated memory is initialised}
var
  NewSizeOK: boolean;
  scr,i,j,k: integer;
  BaseAddr: pointer;
  MemSize: LongWord;
begin
  MemSize := GetLargestFreeMemRegion(BaseAddr);
  NewSizeOK:=true;
  try
    SetLength(Points,2,New_GridWidth,New_GridHeight,New_GridDepth);
{set size of Grid}
  except
    NewSizeOK:=false;
  end;
  try
    SetLength(ColourArray,New_GridWidth,New_GridHeight);
```

```
  except
    NewSizeOK:=false;
  end;
  try
    SetLength(SignArray,New_GridWidth,New_GridHeight,3);
  except
    NewSizeOK:=false;
  end;
  if NewSizeOK then begin
    for scr:=0 to 1 do              {Ensure new space is initialised}
      for i:=0 to New_GridWidth-1 do
        for j:=0 to New_GridHeight-1 do
          for k:=0 to New_GridDepth-1 do
            if (i>GridWidth-1) or (j>GridHeight-1) or
            (k>GridDepth-1) then
              Points[scr,i,j,k]:=NullPoint;
  end
  else begin
    New_GridWidth:=GridWidth;     {Memory allocation failed, so }
    New_GridHeight:=GridHeight;   {return to previous Dimensions}
    New_GridDepth:=GridDepth;
  end;
end;

procedure TForm1.FormDestroy(Sender: TObject);
{When the Form closes, De-allocate all the dynamic array memory}
{allocated during the program's execution.}
begin
  Points:=nil;          {Free all memory allocated}
  ColourArray:=nil;
  SignArray:=nil;
end;

function TForm1.VectDiv(VectGroup: VectorGrp): Vector;
{This function is the same as Del . Vect (Vector dot product) }

{Perform the Vector Div function on the vector group passed to it.}
{The value returned (as a vector) is for the central point of the
group}
var
  dVx_dx,dVy_dy,dVz_dz: extended;
  divVect: Vector;
begin
  with VectGroup do begin
    dVx_dx:=((v2.x - v0.x)/dx + (v0.x - v1.x)/dx)/2;
    dVy_dy:=((v4.y - v0.y)/dy + (v0.y - v3.y)/dy)/2;
    dVz_dz:=((v6.z - v0.z)/dz + (v0.z - v5.z)/dz)/2;
  end;

  with divVect do begin
    x:=dVx_dx;
    y:=dVy_dy;
    z:=dVz_dz;
  end;
```

```
  Result:=divVect;
end;

function TForm1.VectCurl(VectGroup: VectorGrp): Vector;
{This function is the same as Del x Vect (Vector cross product) }

{Perform the Vector Curl function on the vector group passed to it.}
{The value returned (as a vector) is for the central point of the
group}
var
  dVz_dy,dVy_dz,dVx_dz,dVz_dx,dVy_dx,dVx_dy: extended;
  CurlVect: Vector;
begin
  with VectGroup do begin
    dVz_dy:=((v4.z - v0.z)/dy + (v0.z - v3.z)/dy)/2;
    dVy_dz:=((v6.y - v0.y)/dz + (v0.y - v5.y)/dz)/2;
    dVx_dz:=((v6.x - v0.x)/dz + (v0.x - v5.x)/dz)/2;
    dVz_dx:=((v2.z - v0.z)/dx + (v0.z - v1.z)/dx)/2;
    dVy_dx:=((v2.y - v0.y)/dx + (v0.y - v1.y)/dx)/2;
    dVx_dy:=((v4.x - v0.x)/dy + (v0.x - v3.x)/dy)/2;
  end;

  with CurlVect do begin
    x:=dVz_dy - dVy_dz;
    y:=dVx_dz - dVz_dx;
    z:=dVy_dx - dVx_dy;
  end;

  Result:=CurlVect;
end;

function TForm1.PointGroup(scr, x, y, z: smallint): PointGrp;
{Return a group of point values for the point in question and all}
{the points immediately adjacent to it. If they are out of the }
{Grid, assign them zero values.}
{Assume the starting point (x,y,z) is valid}
var
  TheGroup: PointGrp;
  xless,yless,zless: boolean;
  xmore,ymore,zmore: boolean;
begin
  if x<=0 then xless:=true else xless:=false;
  if y<=0 then yless:=true else yless:=false;
  if z<=0 then zless:=true else zless:=false;

  if x>=(GridWidth-1) then xmore:=true else xmore:=false;
  if y>=(GridHeight-1) then ymore:=true else ymore:=false;
  if z>=(GridDepth-1) then zmore:=true else zmore:=false;

  with TheGroup do begin
    P0:=points[scr,x,y,z];
//    if xless then P1:=NullPoint else P1:=points[scr,x-1,y,z];
//    if xmore then P2:=NullPoint else P2:=points[scr,x+1,y,z];
//    if yless then P3:=NullPoint else P3:=points[scr,x,y-1,z];
//    if ymore then P4:=NullPoint else P4:=points[scr,x,y+1,z];
```

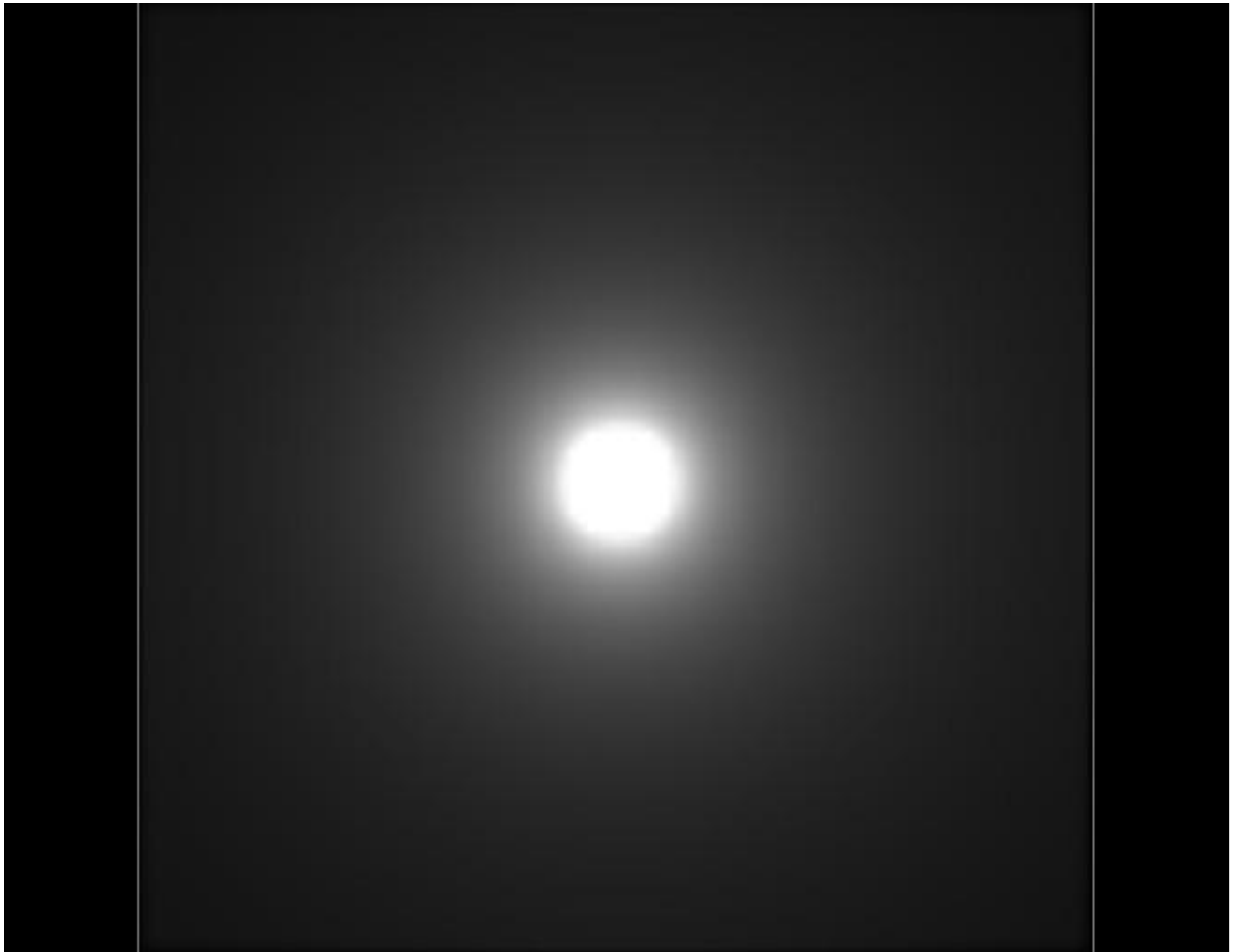
```
//    if zless then P5:=NullPoint else P5:=points[scr,x,y,z-1];
//    if zmore then P6:=NullPoint else P6:=points[scr,x,y,z+1];

    if xless then P1:=points[scr,x+1,y,z] else P1:=points[scr,x-
1,y,z];
    if xmore then P2:=points[scr,x-1,y,z] else
P2:=points[scr,x+1,y,z];
    if yless then P3:=points[scr,x,y+1,z] else P3:=points[scr,x,y-
1,z];
    if ymore then P4:=points[scr,x,y-1,z] else
P4:=points[scr,x,y+1,z];
    if zless then P5:=points[scr,x,y,z+1] else P5:=points[scr,x,y,z-
1];
    if zmore then P6:=points[scr,x,y,z-1] else
P6:=points[scr,x,y,z+1];
  end;
  Result:=TheGroup;
end;

function TForm1.VectorGroup(PntGroup: PointGrp; Field: smallint):
VectorGrp;
{Return a group of vectors for a particular field at, and around the }
{point in question.}
var
  VectGroup: VectorGrp;
begin
  with PntGroup do
    case Field of
      1: begin
           VectGroup.v0:=p0.Electric;
           VectGroup.v1:=p1.Electric;
           VectGroup.v2:=p2.Electric;
           VectGroup.v3:=p3.Electric;
           VectGroup.v4:=p4.Electric;
           VectGroup.v5:=p5.Electric;
           VectGroup.v6:=p6.Electric;
         end;
      2: begin
           VectGroup.v0:=p0.Magnetic;
           VectGroup.v1:=p1.Magnetic;
           VectGroup.v2:=p2.Magnetic;
           VectGroup.v3:=p3.Magnetic;
           VectGroup.v4:=p4.Magnetic;
           VectGroup.v5:=p5.Magnetic;
           VectGroup.v6:=p6.Magnetic;
         end;
      else VectGroup:=NullVectGrp;
    end;
  Result:=VectGroup;
end;

procedure TForm1.FindMaxVal(scr, Field: smallint);
{Find the maximum absolute value of the quantity being displayed, so}
{that the colour levels can be adjusted to give maximum brightness
for}
```

```
{that value.}
var
  i,j,k: smallint;
  vect: Vector;
  VectorType: boolean;
  value: extended;
  Zstart,Zend: smallint;
begin
  VectorType:=false;
  value:=0;
  MaxVal:=0;                {Set it to zero first}
  if TileZ or scale_3D.Checked then begin    {If Z Planes are tiled
find Max of whole volume}
    Zstart:=0;
    Zend:=GridDepth-1;
  end
  else begin
    Zstart:=Z_Plane;     {If not, find Max of current plane only}
    Zend:=Z_Plane;
  end;

  for i:=0 to GridWidth-1 do
    for j:=0 to GridHeight-1 do
      for k:=Zstart to Zend do begin
        case Field of   {depending on which field is required to use}
          1,2,5: begin {calc Electric Field, Magnetic Field, or Power
flow}
                   Vect:=VectorProperty(Field,points[scr,i,j,k]);
                   VectorType:=true;
                 end;
          3: value:=E_Energy(VectSize(points[scr,i,j,k].Electric));
{calc energy in Electric Field}
          4: value:=B_Energy(VectSize(points[scr,i,j,k].Magnetic));
{calc energy in Magnetic Field}
        end;

        if VectorType then with vect do begin
          MaxVal:=Max(MaxVal,abs(x));
          MaxVal:=Max(MaxVal,abs(y));
          MaxVal:=Max(MaxVal,abs(z));
        end
        else
          MaxVal:=Max(MaxVal,abs(value));
      end;
end;

function TForm1.VectorDot(v1, v2: Vector): extended;
var
  DotProduct: extended;
begin
  DotProduct:=(v1.x*v2.x) + (v1.y*v2.y) + (v1.z*v2.z);
  Result:=DotProduct;
end;

function TForm1.ScalarGrad(ScalarGroup: ScalarGrp): Vector;
```

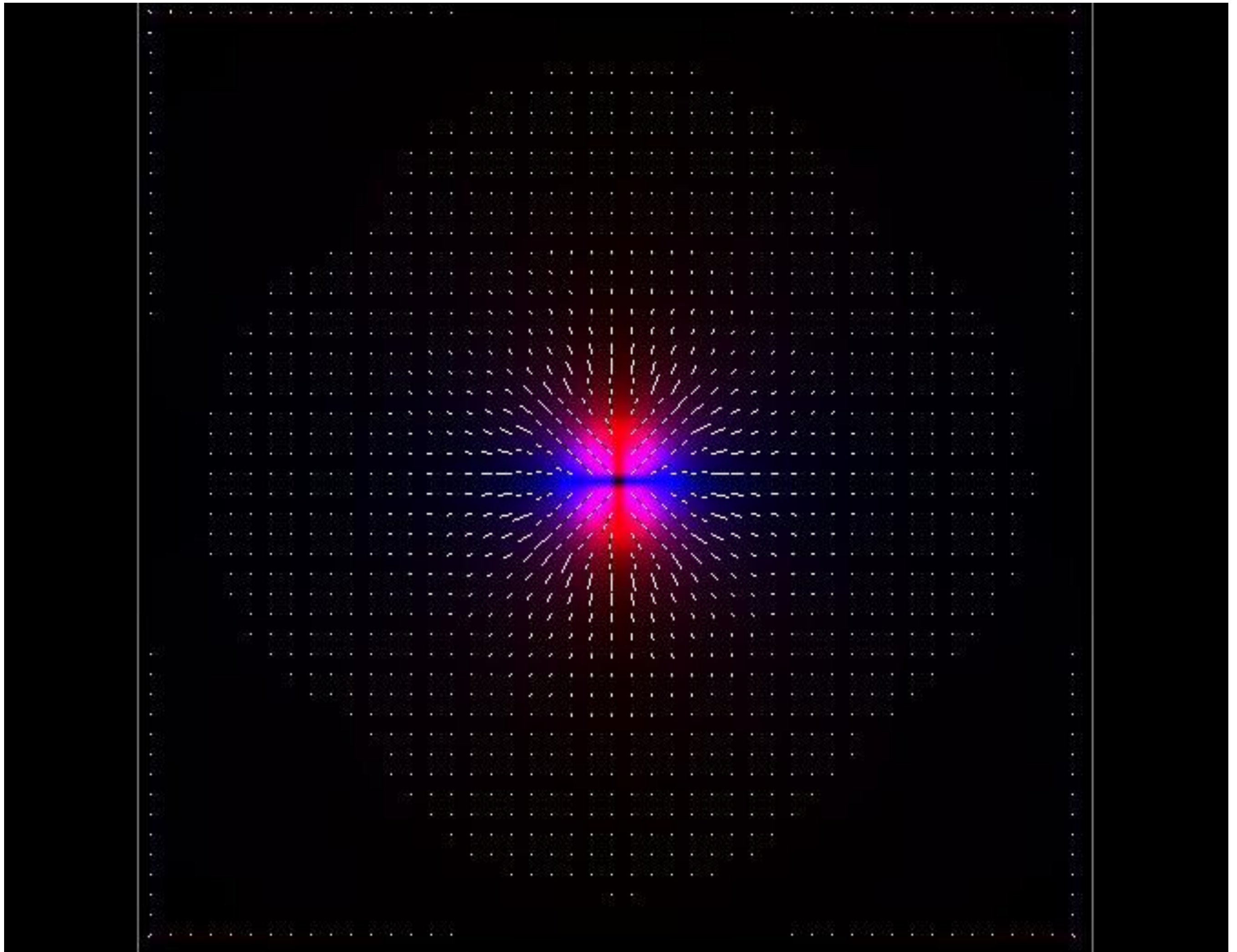
```
{This function is the same as delS }
{It gives the gradient vector of a Scalar field}
var
  GradVect: Vector;
begin
  with ScalarGroup do begin
    GradVect.x:=(s2-s1)/dx;
    GradVect.y:=(s4-s3)/dy;
    GradVect.z:=(s6-s5)/dz;
  end;
  Result:=GradVect;
end;

procedure TForm1.AutoWarnTimerTimer(Sender: TObject);
{If the auto-scale warning indicator's timer is active, toggle the}
{state of the warning indicator at each timer tick.}
begin
  AutoWarnState:=not AutoWarnState;
  if AutoWarnState then
    AutoWarn.Picture.Graphic:=BitmapRed
  else
    AutoWarn.Picture.Graphic:=BitmapBlack;
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Quit:=true;
end;

procedure TForm1.RateOfTimeChange(Sender: TObject);
begin
  New_RateOfTime:=RateOfTime.Position;
  DoUpdate:=true;
end;

procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  if CheckBox1.Checked then begin
    CheckBox2.Enabled:=true;
    save_frames:=true;
    FrameCount:=1;
  end
  else begin
    CheckBox2.Enabled:=false;
    save_frames:=false;
  end;
end;

procedure TForm1.CheckBox2Click(Sender: TObject);
begin
  if CheckBox2.Checked then begin
    save_3D:=true;
  end
  else begin
    save_3D:=false;
```

```
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  New_Flip_YZ:=true;
  DoUpdate:=true;
end;

procedure TForm1.ArrowScaleScrollChange(Sender: TObject);
begin
  New_ArrowScaleFactor := ArrowScaleScroll.Position;
  DoUpdate:=true;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Restart:=true;
  DoUpdate:=true;
end;

procedure TForm1.ViewFromTopClick(Sender: TObject);
begin
  DoUpdate:=true;
end;

procedure TForm1.Scale_3DClick(Sender: TObject);
begin
  DoUpdate:=true;
end;

End.
```

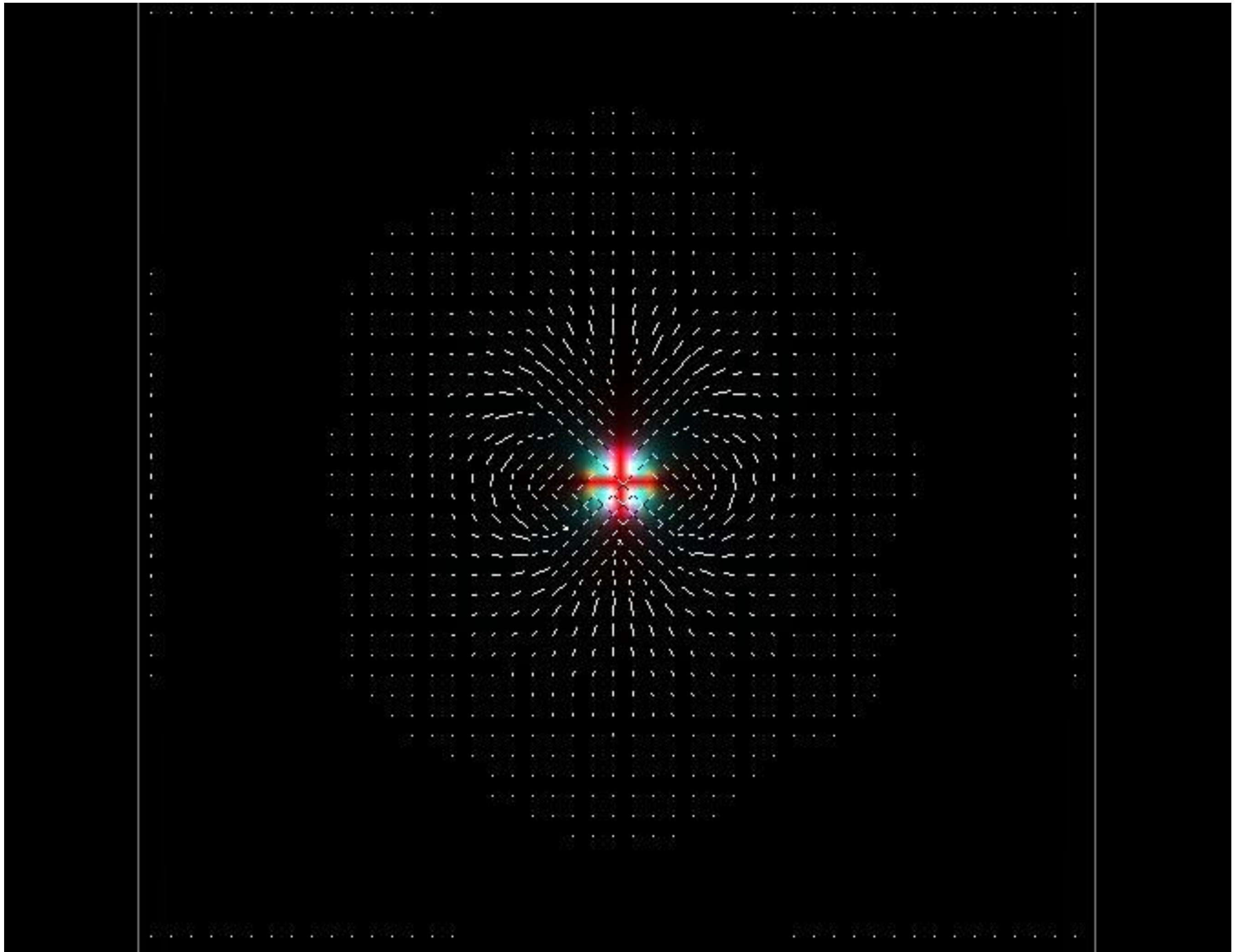
Appendix C - Images from Analytical Electron Model



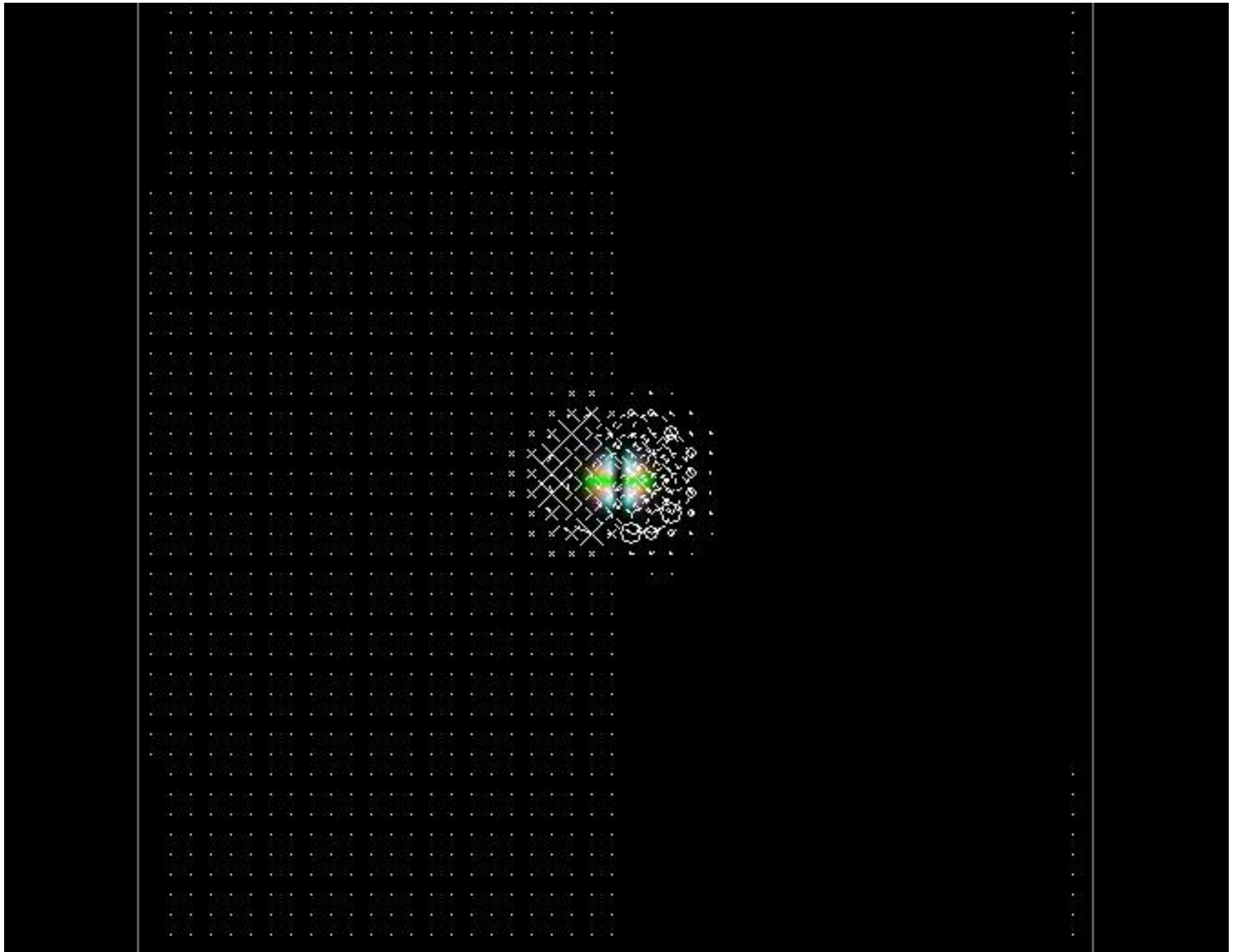
(1) Electric Potential (V) from the Side of the Electron



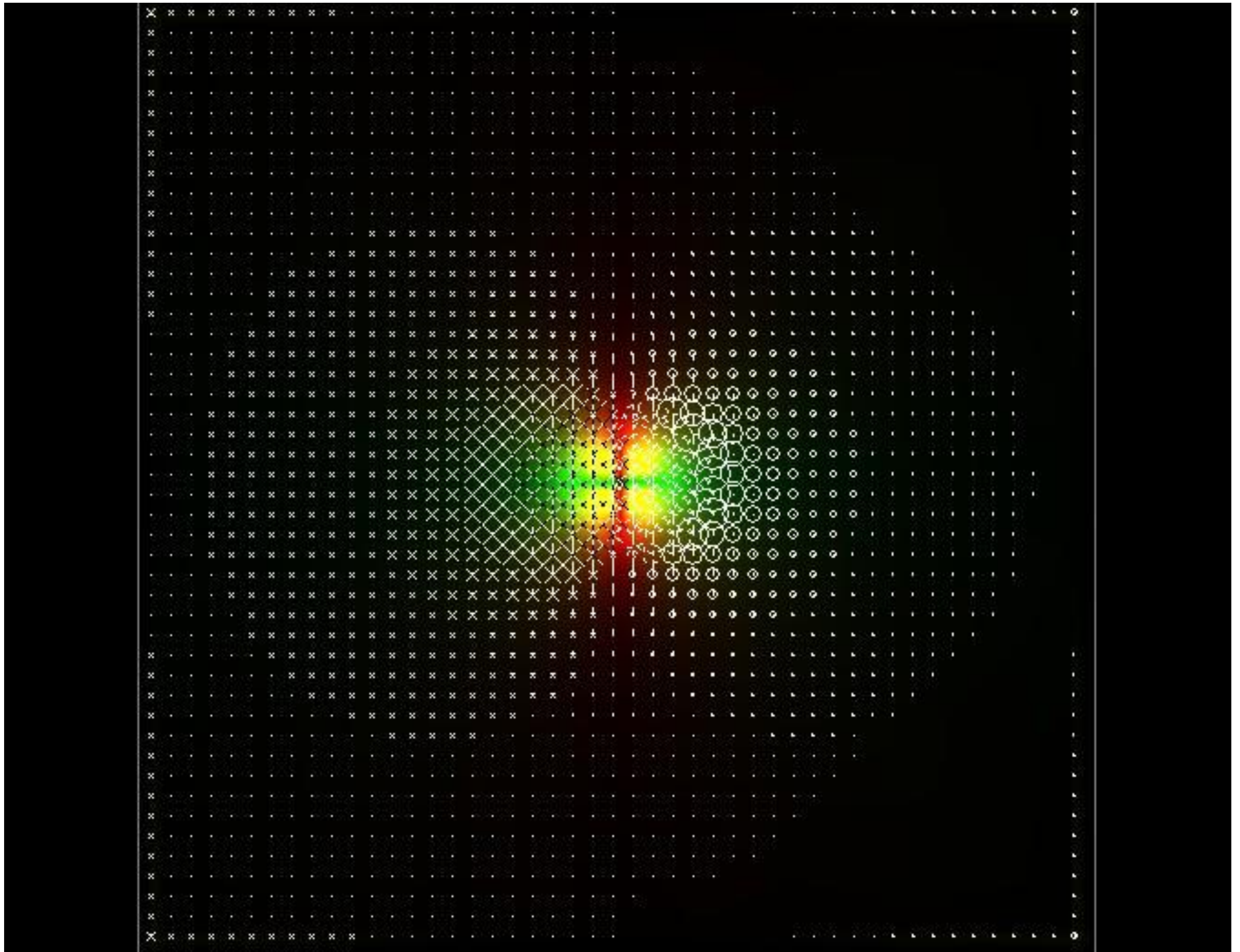
(2) Electric Field (E) from the Side of the Electron



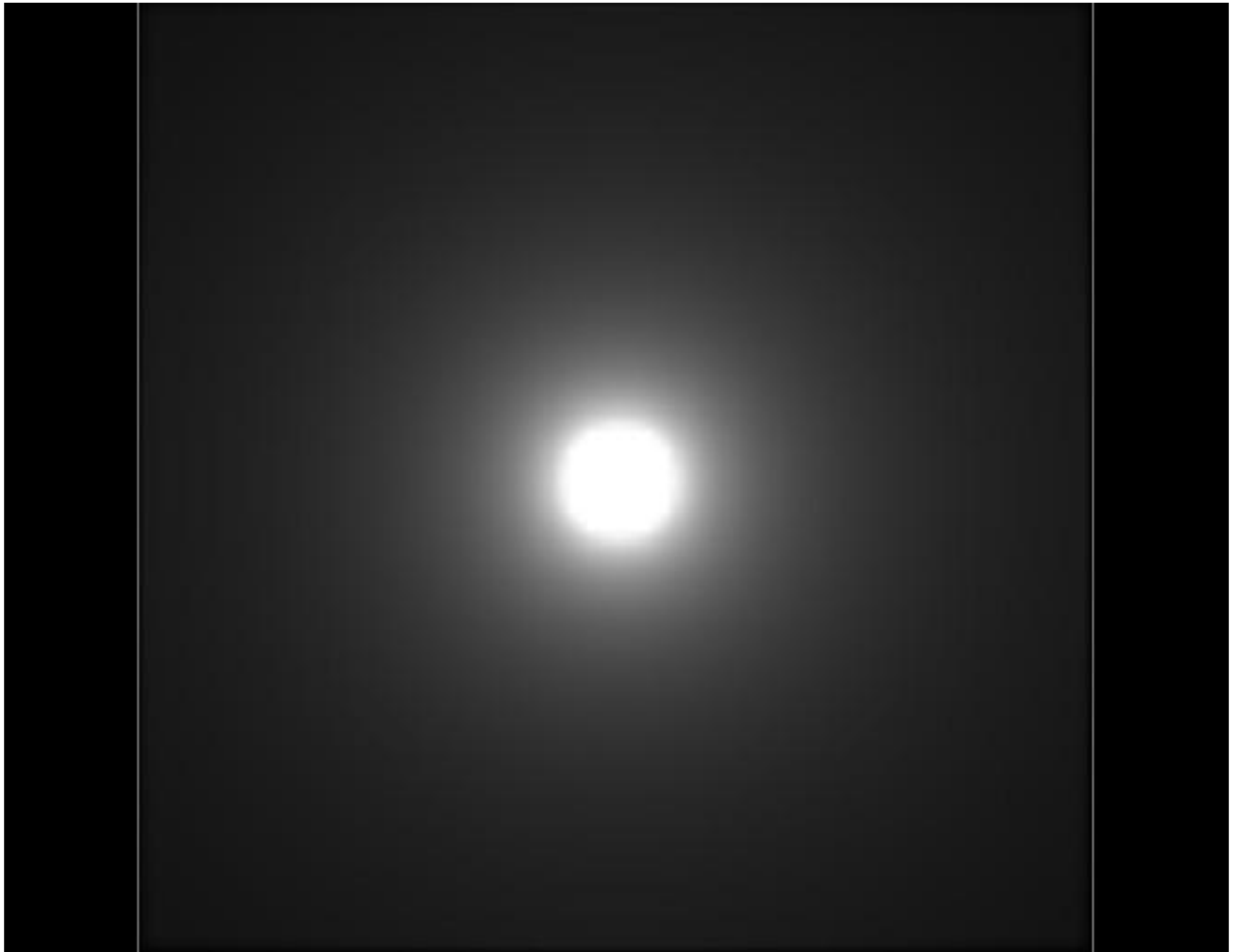
(3) Magnetic Field (H) from the Side of the Electron



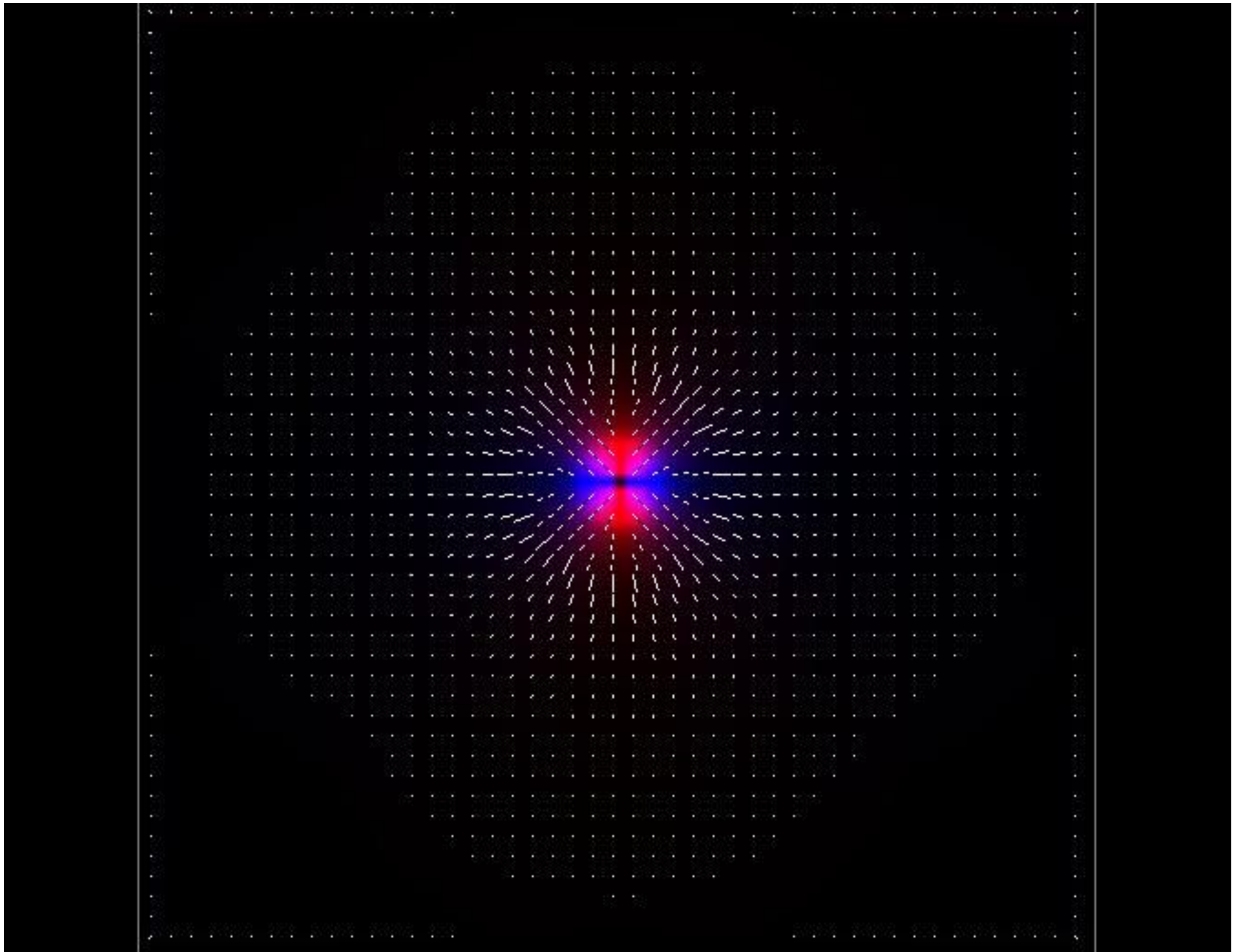
(4) Power Flow from the Side of the Electron



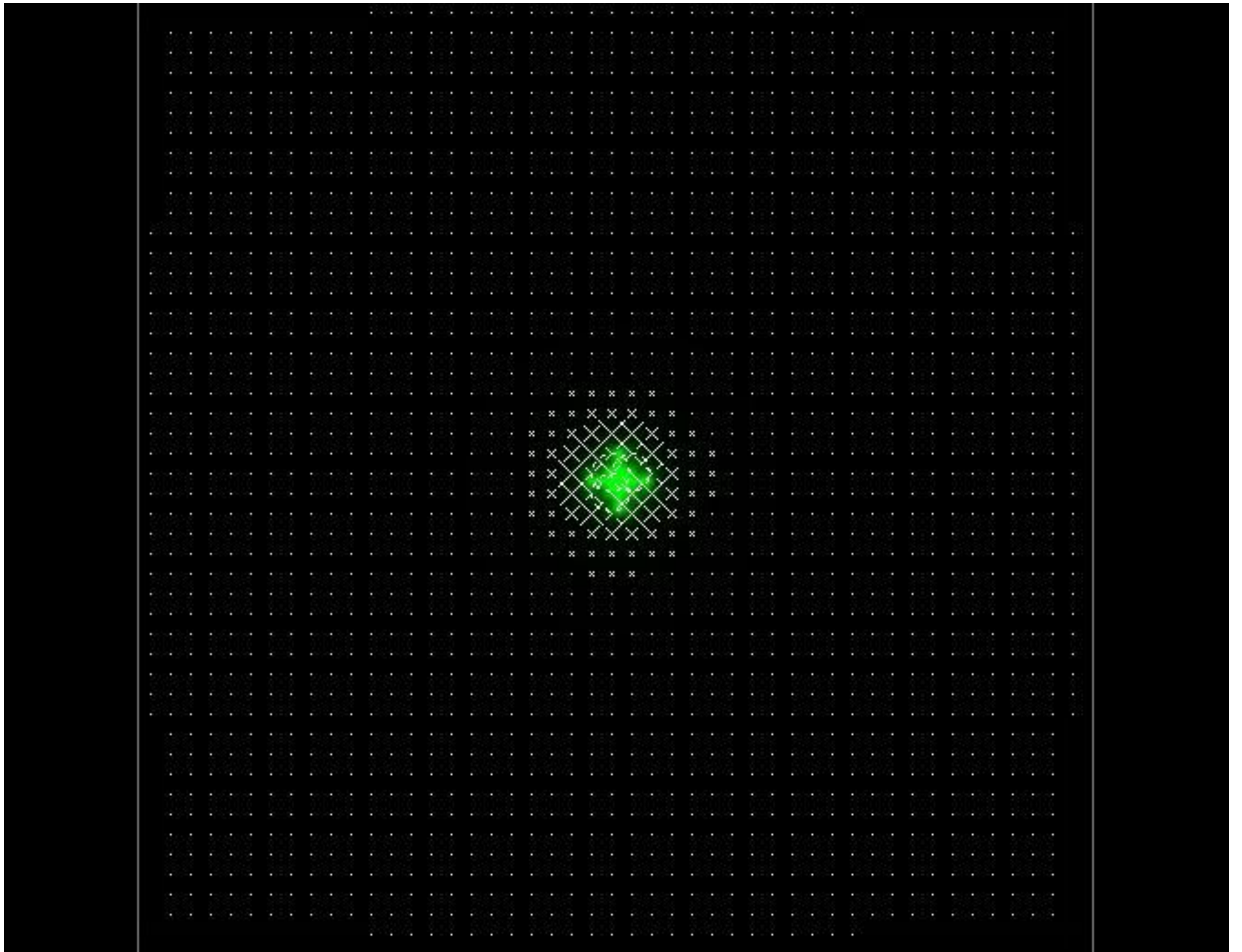
(5) Vector Potential (A) from the Side of the Electron



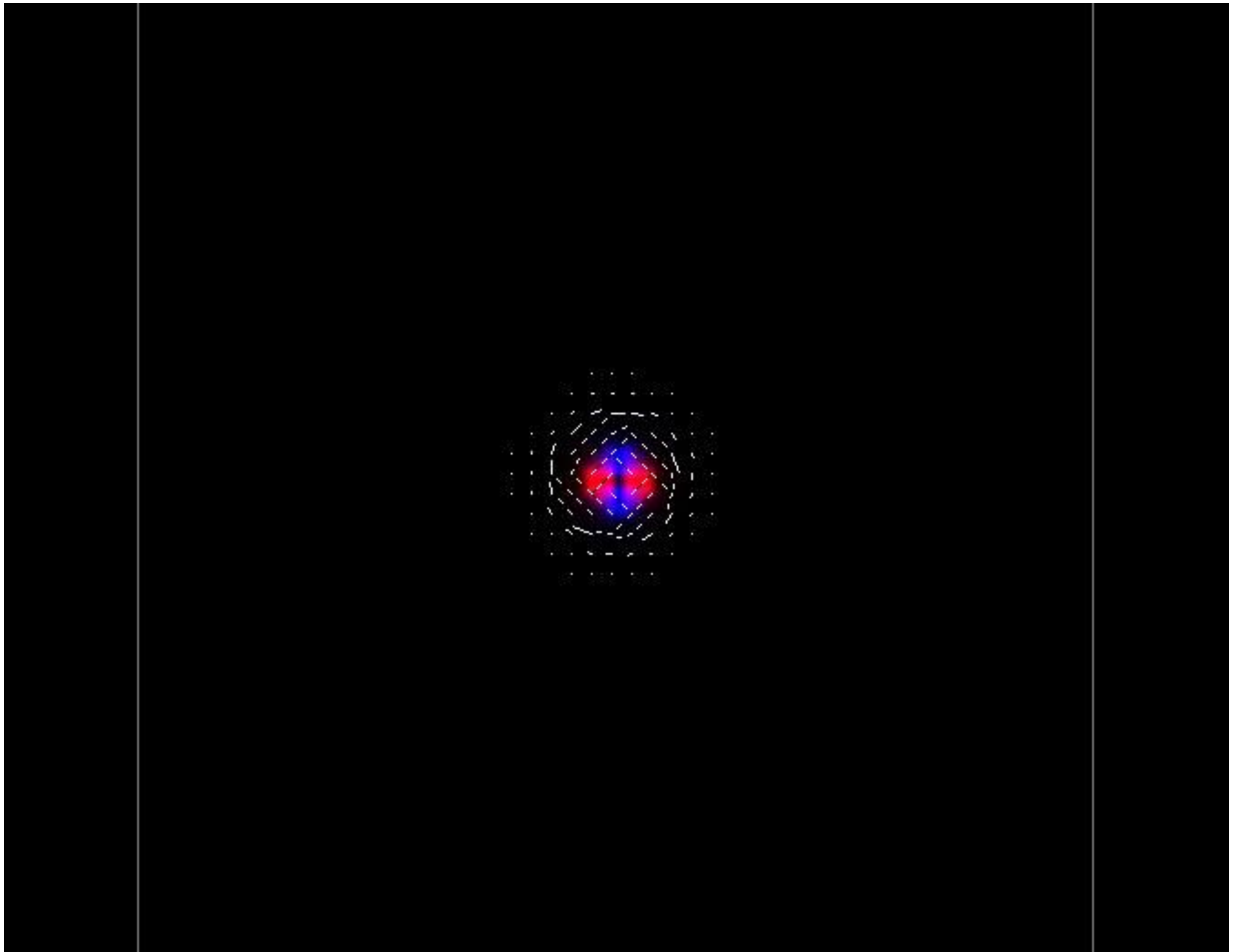
(6) Electric Potential (V) from the Top of the Electron



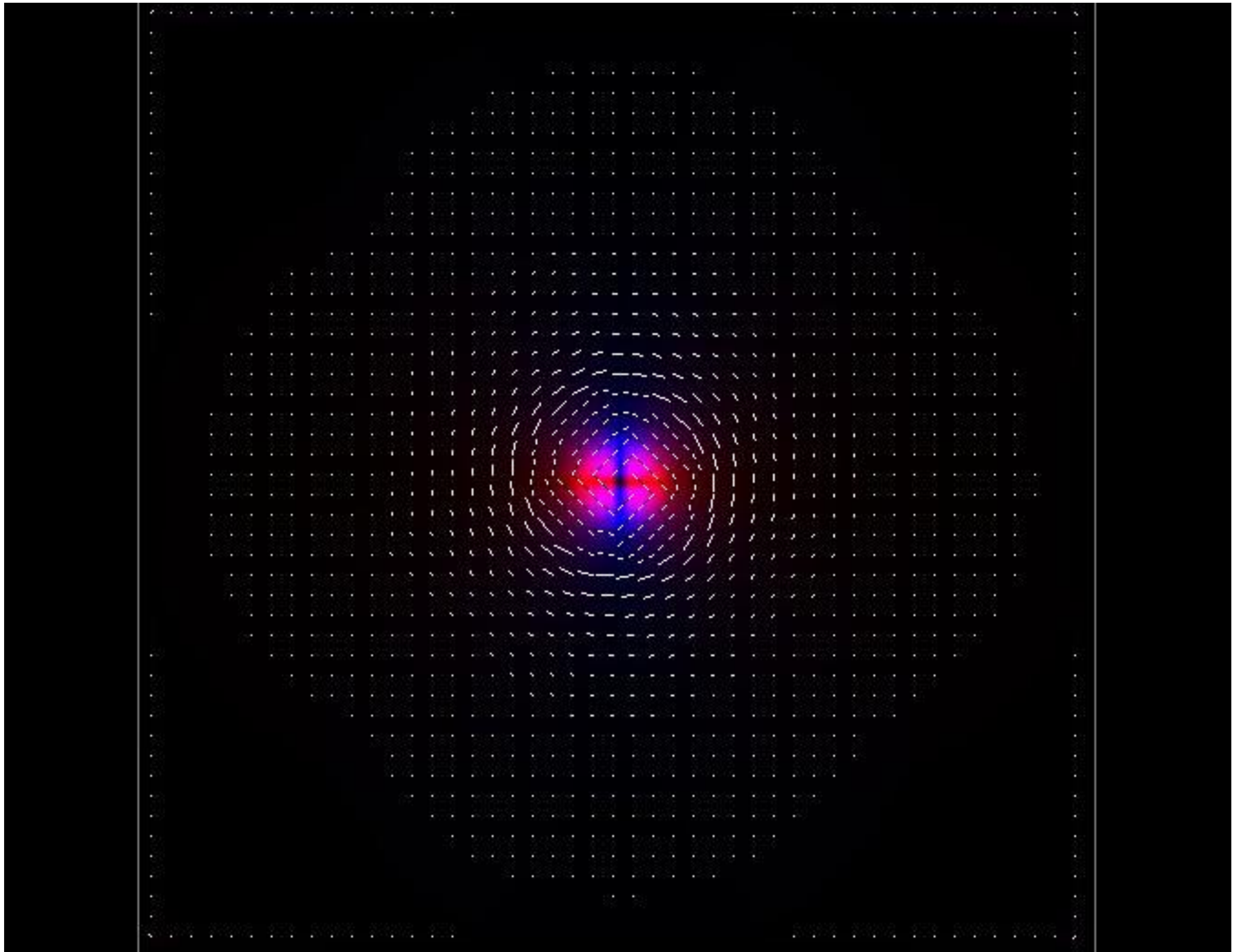
(7) Electric Field (E) from the Top of the Electron



(8) Magnetic Field (H) from the Top of the Electron



(9) Power Flow from the Top of the Electron



(10) Vector Potential (A) from the Top of the Electron

Appendix D: My Delphi Source Code for Images generated in Appendix A

```
//  VectPotential - 3D wave modelling platform.
//
//  (c) Copyright 1998 - 2011 : Declan Traill
//

unit VectorPotential;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  ExtCtrls, StdCtrls, Spin, ComCtrls, Math, Grids, Jpeg;

type            {Type declaration for a vector resolved into x,y & z
components}
  Vector = record
    x: extended;
    y: extended;
    z: extended;
  end;

  VectorGrp = record
    V0: Vector;
    V1: Vector;
    V2: Vector;
    V3: Vector;
    V4: Vector;
    V5: Vector;
    V6: Vector;
  end;

  ScalarGrp = record
    S0: extended;
    S1: extended;
    S2: extended;
    S3: extended;
    S4: extended;
    S5: extended;
    S6: extended;
  end;

  point = record      {Type declaration for a grid point}
    Phi: extended;
    Hertzian: Vector;
    Hertzian_scalar: Vector;
    ElectricPotential: Extended;
    VectorPotential: Vector;
```

```
    Electric: Vector;
    Magnetic: Vector;
  end;

  PointGrp = record
    P0: point;
    P1: point;
    P2: point;
    P3: point;
    P4: point;
    P5: point;
    P6: point;
  end;

  PointPtr = ^point;     {Define a type: pointer to a grid point}
  BitmapPtr = ^TBitmap;  {Define a type: pointer to a bitmap}
  TColorPtr = ^TColor;   {Define a type: pointer to a 24 bit colour
type (TColor)}

  EdgeMasks = array[1..9,1..3,1..3] of byte;  {Define the EdgeMasks
array type}

  TForm1 = class(TForm)
    StartGroup: TGroupBox;
    Start1: TRadioButton;
    Start2: TRadioButton;
    start3: TRadioButton;
    Start4: TRadioButton;
    Start5: TRadioButton;
    Start6: TRadioButton;
    Start7: TRadioButton;
    Start8: TRadioButton;
    Start9: TRadioButton;
    ZPlaneGroup: TGroupBox;
    ZPlane: TTrackBar;
    Z_Plane_Number: TEdit;
    MainGroup: TGroupBox;
    TimeDisplay: TEdit;
    ReScale: TSpinEdit;
    AmpDisplay: TEdit;
    TimeFreeze: TButton;
    Image1: TImage;
    DisplayLevel: TScrollBar;
    FieldGroup: TGroupBox;
    Field1: TRadioButton;
    Field2: TRadioButton;
    Field3: TRadioButton;
    Field4: TRadioButton;
    Field5: TRadioButton;
    StatsGroup: TGroupBox;
    Energy1: TEdit;
    Energy2: TEdit;
    Energy3: TEdit;
    Units1: TEdit;
    Units2: TEdit;
```

```
Units3: TEdit;
Energy_Msg1: TEdit;
Energy_Msg2: TEdit;
Energy_Msg3: TEdit;
ColourGroup: TGroupBox;
ColourX_Group: TGroupBox;
ColourY_Group: TGroupBox;
ColourZ_Group: TGroupBox;
X_Red: TRadioButton;
X_Green: TRadioButton;
X_Blue: TRadioButton;
Y_Red: TRadioButton;
Y_Green: TRadioButton;
Y_Blue: TRadioButton;
Z_Red: TRadioButton;
Z_Green: TRadioButton;
Z_Blue: TRadioButton;
X_Colour: TImage;
Y_Colour: TImage;
Z_Colour: TImage;
DisplayOptionsGroup: TGroupBox;
RendDisplay: TCheckBox;
Vect_Arrows: TCheckBox;
Vector_Group: TGroupBox;
VectorX: TCheckBox;
VectorY: TCheckBox;
VectorZ: TCheckBox;
Spacing_Text: TStaticText;
Spacing_metres: TRadioButton;
Spacing_pixels: TRadioButton;
Bevel1: TBevel;
FirstZ: TStaticText;
LastZ: TStaticText;
Z_Tiling: TButton;
TileGrid: TDrawGrid;
GridGroup: TGroupBox;
GridX: TEdit;
GridY: TEdit;
GridZ: TEdit;
GridXlabel: TLabel;
GridYlabel: TLabel;
GridZlabel: TLabel;
RendGroup: TGroupBox;
AspectControl: TCheckBox;
AcceptGridSize: TButton;
RendColour: TGroupBox;
GreyscaleButton: TRadioButton;
ColourButton: TRadioButton;
X_none: TRadioButton;
Y_none: TRadioButton;
Z_none: TRadioButton;
AutoWarnTimer: TTimer;
Timesteps: TSpinEdit;
Label1: TLabel;
RateOfTime: TScrollBar;
```

```
Label2: TLabel;
GroupBox1: TGroupBox;
GroupBox2: TGroupBox;
CheckBox1: TCheckBox;
CheckBox2: TCheckBox;
Spacing_gridpoints: TRadioButton;
Button1: TButton;
VectorSpacing: TEdit;
RenderOption1: TRadioButton;
RenderOption2: TRadioButton;
RenderOption3: TRadioButton;
ArrowScaleScroll: TScrollBar;
Label3: TLabel;
ActualGridWidth: TEdit;
Label4: TLabel;
Label5: TLabel;
Start10: TRadioButton;
Button2: TButton;
ViewFromTop: TCheckBox;
AutoScaleGroup: TGroupBox;
AutoWarn: TImage;
Auto1: TRadioButton;
Auto2: TRadioButton;
Auto3: TRadioButton;
Scale_3D: TCheckBox;
field6: TRadioButton;
Field7: TRadioButton;
Field8: TRadioButton;
Field9: TRadioButton;
CheckBox3: TCheckBox;
procedure FormCreate(Sender: TObject);
procedure Start1Click(Sender: TObject);
procedure Start2Click(Sender: TObject);
procedure Field1Click(Sender: TObject);
procedure Field2Click(Sender: TObject);
procedure Field3Click(Sender: TObject);
procedure Field4Click(Sender: TObject);
procedure Field5Click(Sender: TObject);
procedure Field6Click(Sender: TObject);
procedure Field7Click(Sender: TObject);
procedure Field8Click(Sender: TObject);
procedure Field9Click(Sender: TObject);
procedure start3Click(Sender: TObject);
procedure Start4Click(Sender: TObject);
procedure Start5Click(Sender: TObject);
procedure Start6Click(Sender: TObject);
procedure TimeFreezeClick(Sender: TObject);
procedure Start7Click(Sender: TObject);
procedure ZPlaneChange(Sender: TObject);
procedure DisplayLevelChange(Sender: TObject);
procedure ReScaleChange(Sender: TObject);
procedure Start8Click(Sender: TObject);
procedure Start9Click(Sender: TObject);
procedure Start10Click(Sender: TObject);
procedure X_RedClick(Sender: TObject);
```

```
    procedure X_GreenClick(Sender: TObject);
    procedure X_BlueClick(Sender: TObject);
    procedure Y_RedClick(Sender: TObject);
    procedure Y_GreenClick(Sender: TObject);
    procedure Y_BlueClick(Sender: TObject);
    procedure Z_RedClick(Sender: TObject);
    procedure Z_GreenClick(Sender: TObject);
    procedure Z_BlueClick(Sender: TObject);
    procedure GreyscaleButtonClick(Sender: TObject);
    procedure ColourButtonClick(Sender: TObject);
    procedure Auto1Click(Sender: TObject);
    procedure Auto2Click(Sender: TObject);
    procedure Auto3Click(Sender: TObject);
    procedure Vect_ArrowsClick(Sender: TObject);
    procedure VectorSpacingChange(Sender: TObject);
    procedure Z_TilingClick(Sender: TObject);
    procedure Image1Click(Sender: TObject);
    procedure Image1DblClick(Sender: TObject);
    procedure VectorXClick(Sender: TObject);
    procedure VectorYClick(Sender: TObject);
    procedure VectorZClick(Sender: TObject);
    procedure Spacing_pixelsClick(Sender: TObject);
    procedure Spacing_metresClick(Sender: TObject);
    procedure Spacing_gridpointsClick(Sender: TObject);
    procedure RenderOption1Click(Sender: TObject);
    procedure RenderOption2Click(Sender: TObject);
    procedure RenderOption3Click(Sender: TObject);
    procedure RendDisplayClick(Sender: TObject);
    procedure AspectControlClick(Sender: TObject);
    procedure GridXChange(Sender: TObject);
    procedure GridYChange(Sender: TObject);
    procedure GridZChange(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure AcceptGridSizeClick(Sender: TObject);
    procedure X_noneClick(Sender: TObject);
    procedure Y_noneClick(Sender: TObject);
    procedure Z_noneClick(Sender: TObject);
    procedure AutoWarnTimerTimer(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure RateOfTimeChange(Sender: TObject);
    procedure CheckBox1Click(Sender: TObject);
    procedure CheckBox2Click(Sender: TObject);
    procedure CheckBox3Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure ArrowScaleScrollChange(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure ViewFromTopClick(Sender: TObject);
    procedure Scale_3DClick(Sender: TObject);
  private
    { Private declarations }
  public
    procedure Propagate;
    procedure Initialise(first:boolean);
    procedure RecalcFields(scr:smallint);
    procedure UpdateBitmap(scr:smallint);
```

```
    procedure DisplayScreen(scr:smallint);
    function sign(number: extended): shortint;
    function Gradient(val1, val2: extended): extended;
    function Gradient_3point(val1, val2, val3: extended): extended;
    function Gauss(x: extended): extended;
    procedure UpdateDetails;
    procedure UpdateAxisColours(which: string);
    procedure Auto_Scale(scr: smallint);
    procedure MaxCheck(element: PointPtr);
    function VectSize(vect: Vector): extended;
    function E_Energy(E_amp: extended): extended;
    function B_Energy(B_amp: extended): extended;
    procedure DrawArrow(ThisBitmap: Tbitmap;x,y: smallint; arrow:
Vector);
    procedure NewBitmap(BmapPtr: BitmapPtr);
    procedure PlotPoint(Bmap: TBitmap; x, y: smallint);
    function RGB_Val(colour: Tcolor; primary: integer): byte;
    function EdgeCase(x, y, Xmin, Ymin, Xmax, Ymax: smallint): byte;
    function VectorInterpolate(v1, v2, v3, v4: Vector; Xfrac,
      Yfrac: extended): Vector;
    function Interpolate(val1, val2, val3, val4, Xfrac,
      Yfrac: extended): extended;
    procedure PlotQuadrant(Bmap: TBitmap; Quadrant: byte; RealX,
RealY: extended);
    function GetActualX(x: smallint): smallint;
    function GetActualY(y: smallint): smallint;
    procedure SetTileSize;
    function Round2(realval: extended): int64;
    function GetRealX(x: extended): extended;
    function GetRealY(y: extended): extended;
    function ColourRange(value: extended; ScaleFactor: extended):
byte;
    function VectToColours(vect: vector; ScaleFactor: extended):
vector;
    function ByteLimit(value: extended): smallint;
    function VectByteLimit(vect: Vector): Vector;
    function VectorCross(v1, v2: Vector): Vector;
    function PowerFlow(Apoint: point): vector;
    function VectorProperty(field: byte; Apoint: point): vector;
    function PointNull(Apoint: point): boolean;
    function VectorNull(vect: Vector): boolean;
    function ReverseTColor(input: TColor): Tcolor;
    function MouseZplane: smallint;
    procedure TileCursor(Bmap: TBitmap; Tile: smallint; colour:
TColor);
    procedure PlotPixel(x, y: smallint; Colour: TColor);
    procedure UpdateE_Energy(scr: smallint);
    procedure UpdateB_Energy(scr: smallint);
    procedure SetAspectRatio;
    procedure SetGridGlobals;
    procedure ReAllocGridMemory;
    function VectDiv(VectGroup: VectorGrp): extended;
    function VectCurl(VectGroup: VectorGrp): Vector;
    function PointGroup(scr, x, y, z: smallint): PointGrp;
```

```
    function VectorGroup(PntGroup: PointGrp; Field: smallint):
VectorGrp;
    function ScalarGroup(PntGroup: PointGrp; Field: smallint):
ScalarGrp;
    function IntegrateScalarGrp(ScalarGroup: ScalarGrp): extended;
    procedure FindMaxVal(scr, Field: smallint);
    function VectorDot(v1, v2: Vector): extended;
    function ScalarGrad(ScalarGroup: ScalarGrp): Vector;
    { Public declarations }
  end;
const
{EdgeArray is an array used to determine which points exist within the
range of}
{a rectangular region (such as a grid x/y plane) given a known edge
condition}
{The edge condition is determined elsewhere by EdgeCase; its values
range from}
{1 to 9 corresponding to the point in question residing at the topleft
corner,}
{somewhere along the top edge, the topright corner... etc.}
{The values in the sub-arrays indicate which of the points surrounding
the }
{point in question are in range: 1=in range, 0=out of range}

  EdgeArray : EdgeMasks = (((0,0,0),      {Top Left corner}
                            (0,1,1),
                            (0,1,1)),

                           ((0,0,0),      {Top Edge}
                            (1,1,1),
                            (1,1,1)),

                           ((0,0,0),      {Top Right Corner}
                            (1,1,0),
                            (1,1,0)),

                           ((0,1,1),      {Left Edge}
                            (0,1,1),
                            (0,1,1)),

                           ((1,1,1),      {Not on an Edge or Corner}
                            (1,1,1),
                            (1,1,1)),

                           ((1,1,0),      {Right Edge}
                            (1,1,0),
                            (1,1,0)),

                           ((0,1,1),      {Bottom Left Corner}
                            (0,1,1),
                            (0,0,0)),

                           ((1,1,1),      {Bottom Edge}
                            (1,1,1),
                            (0,0,0)),
```

```
                           ((1,1,0),     {Bottom Right Corner}
                            (1,1,0),
                            (0,0,0)));
const
  Default_GridWidth=100;     {Default Number of Pixels wide}
  Default_GridHeight=100;    {Default Number of Pixels high}
  Default_GridDepth=100;     {Default Number of Pixels Deep}

  PPM=1E8; {Points Per Metre}
  PointSize=1/PPM; {size, in metres, that each point represents}

  SpeedOfLight=2.998E8;   {Speed of Light in meters per second}
  Permittivity=8.854E-12; {Permittivity of free space (farad/m)}
  Permeability=4*Pi*1E-7; {Permeability of free space (Henry/m)}
  Hhat=1.054571596E-34; {Planck's constant}

  ElectronCharge=-(1.60217656535E-19);
  ElectronMass=9.10938188E-31;
  ElectronMassDivHhat=(ElectronMass/Hhat);
  ElectronComptonWavelength=2.4263102175/1000000000000;

  E_Amplitude=1000*PointSize;    {Max amplitude of input E field -
Volts/m }
  B_Amplitude=1000*PointSize;    {Max amplitude of input B field -
Tesla  }
  Max_E=0;  {Max value of Electric field to allow per pixel - Volts/m
(0 value disables it)}
  Max_B=0;  {Max value of Magnetic field to allow per pixel - Tesla (0
value disables it)}

  RED=$1;      {Multiplication factor for 4-byte RGB colour}
  GREEN=$100;  {Multiplication factor for 4-byte RGB colour}
  BLUE=$10000; {Multiplication factor for 4-byte RGB colour}
  BLACK=$0;    {Definition for black}
  RedMask=$FF;        {Masking value for colour separation}
  GreenMask=$FF00;    {Masking value for colour separation}
  BlueMask=$FF0000;   {Masking value for colour separation}
  START=1;     {AutoScaling constants}
  CONTINUAL=2; {AutoScaling constants}
  NEVER=3;     {AutoScaling constants}
  OneToOne=1;  {Rendering Option Constant - for 1 pixel per point}
  Chunky=2;    {Rendering Option Constant - for filled rectangle per
point}
  Blend=3;     {Rendering Option Constant - colour blending betweem
points}
  EdgeSize=3;  {Size of border between Tiled Z Planes & screen edge}

  HERTZIAN_SCALAR_FIELD=1;
  ELECTRIC_POTENTIAL_FIELD=2;
  HERTZIAN_FIELD=3;
  VECTOR_POTENTIAL_FIELD=4;
  ELECTRIC_FIELD=5;
  MAGNETIC_FIELD=6;
  E_ELECTRIC_FIELD=7;
```

```
  E_MAGNETIC_FIELD=8;
  POWER_FLOW_FIELD=9;

var
  Form1: TForm1;
  Bitmap1,Bitmap2: TBitmap;
  BitmapRed,BitmapGreen,BitmapBlue,BitmapBlack: TBitmap;

  // Note: the Grid (0,0,0) point is at the Top, Left & Back point of
the Grid
  //       So X increments to the right, Y increments downwards, & Z
increments out of the screen
  points: array of array of array of array of point;
{[1..2,1..GridWidth,1..GridHeight,1..GridDepth] of point;}

  ColourArray: array of array of TColor; {[1..GridWidth,1..GridHeight]
of Tcolor;}
  SignArray: array of array of array of shortint;
{[1..GridWidth,1..GridHeight,1..3] of shortint;}
  ColArray: array[1..9] of Tcolor;
  PntArray: array[1..9] of Vector;
  YLinePtrs: array of PByteArray;
  GridWidth,GridHeight,GridDepth: integer;
  New_GridWidth,New_GridHeight,New_GridDepth: integer;
  screen: smallint;
  StartOption,New_StartOption: smallint;
  New_DisplayField,DisplayField : smallint;
  Time,TimeStep : extended;
  RescaleFactor, New_ReScale : double;
  Z_Plane, New_ZPlane : smallint;
  FreezeTime, New_FreezeTime, GridTransformed: Boolean;
  TileZ, New_TileZ: Boolean;
  ShowColour, New_ShowColour: Boolean;
  IsGrey,DoUpdate : Boolean;
  E_Energy_Tot,B_Energy_Tot,MaxVal : extended;
  Amplification : extended;
  X_RGB,Y_RGB,Z_RGB : Tcolor;
  New_AutoScale,AutoScale: smallint;
  ZplanePos,OriginX,OriginY : smallint;
  FirstPass : boolean;
  ArrowScale: extended;
  ScrScaleX,ScrScaleY,HalfX,HalfY : extended;
  BitmapX,BitmapY: smallint;
  TileX,TileY,TileXcount,TileYcount : smallint;
  Aspect: extended;
  NullVect: Vector;
  NullPoint: Point;
  NullVectGrp: VectorGrp;
  NullScalarGrp: ScalarGrp;
  NullPointGrp: PointGrp;
  MyMouse: TMouse;
  AxisColours,New_AxisColours: string;
  Display_Level,New_DisplayLevel: integer;
  Rate_Of_Time,New_RateOfTime: integer;
  Arrows, New_Arrows: boolean;
```

```
  UpdateColours,ReDraw: boolean;
  CurrentBitmap: TBitmap;
  New_Render,Render: smallint;
  New_Rendered,Rendered: boolean;
  New_VectSpacing,VectSpacing: integer;
  TileScrScaleX,TileScrScaleY: double;
  TileHalfX,TileHalfY: extended;
  MaintainAspect,New_MaintainAspect,VectorChange: boolean;
  ActualWidth,ActualHeight,ActualDepth: extended;
  PointArea,PointVolume: extended;
  ScreenAspect: extended;
  dx,dy,dz: extended;
  ArrowsUnitsChange,AutoWarnState: boolean;
  TileRect: TRect;
  Quit: boolean;
  Timestep_count: smallint;
  FrameCount: integer;
  save_frames,save_3D,New_Flip_YZ,Flip_YZ: boolean;
  arrow_step: extended;
  New_ArrowScaleFactor: integer;
  ArrowScaleFactor: extended;
  Restart: boolean;
  ViewTop: boolean;
  AllFields: boolean;


implementation

{$R *.DFM}
{$G-} {Disable Data Importation from Units - Improves Memory access
efficiency}

procedure TForm1.FormCreate(Sender: TObject);
begin
  Initialise(true);
  Propagate();
end;

procedure TForm1.Initialise(first:boolean);
var
  scr,i,j,k : smallint;
begin
  if first then begin
    Restart:=true;
    FreezeTime:=false;
    New_FreezeTime:=false;
    GridTransformed:=false;
    StartOption:=0;                {No Starting conditions untill
selected}
    New_StartOption:=1;           {default to start config 10}
    MyMouse:=TMouse.Create;        {Create a mouse variable}
    BitmapX:=Image1.Width;         {get width of picture control on
the form}
    BitmapY:=Image1.Height;        {get height of picture control on
the form}
```

```
    ScreenAspect:=BitmapY/BitmapX; {Calc Screen's aspect ratio}
    SetLength(YLinePtrs,BitmapY);  {determine size for array of bitmap
line pointers}
    New_DisplayField:=VECTOR_POTENTIAL_FIELD;     {set default to
displaying the Electric field}
    NewBitmap(@Bitmap1);     {create a new blank 24bit bitmap}
    NewBitmap(@Bitmap2);     {create a new blank 24bit bitmap}
    BitmapRed := TBitmap.Create;     {create a bitmap for the red
square}
    BitmapGreen := TBitmap.Create;   {create a bitmap for the green
square}
    BitmapBlue := TBitmap.Create;    {create a bitmap for the blue
square}
    BitmapBlack := TBitmap.Create;    {create a bitmap for the black
square}
    BitmapRed.LoadFromFile('red.bmp');      {load the red square from
disk}
    BitmapGreen.LoadFromFile('green.bmp');  {load the green square
from disk}
    BitmapBlue.LoadFromFile('blue.bmp');    {load the blue square from
disk}
    BitmapBlack.LoadFromFile('black.bmp');  {load the black square
from disk}

    New_ShowColour:=true;            {default to displaying a colour
image}
    AutoScale:=CONTINUAL;             {default autoscale option to at
Continual}
    New_TileZ:=false;                {default Z plane Tiling option}
    New_Arrows:=true;                {default to Vector Arrows On}
    New_Render:=Blend;               {default to Colour Blended
Rendering}
    New_Rendered:=true;              {default to Rendered Display On}
    New_DisplayLevel:=1995;          {default Level slider to (2000 -
n)*1000%}
    New_RateOfTime:=RateOfTime.Position; {default Rate of Time slider}
    New_MaintainAspect:=true;        {Default Aspect Ratio Control
setting}
    New_AutoScale:=CONTINUAL;        {Default AutoScale option to
Continual}
    GridWidth:=0;                    {Ensure a full update occurs}
    GridHeight:=0;
    GridDepth:=0;
    New_GridWidth:=Default_GridWidth;     {set default Grid
Dimensions}
    New_GridHeight:=Default_GridHeight;
    New_GridDepth:=Default_GridDepth;
    VectorChange:=false;              {initialise flag}
    FrameCount:=1;
    save_frames:=false;
    save_3D:=false;
    New_Flip_YZ:=false;
    Flip_YZ:=false;
    New_ArrowScaleFactor := ArrowScaleScroll.Position;
    ViewTop := ViewFromTop.Checked;
```

```
    dx:=PointSize;
    dy:=PointSize;
    dz:=PointSize;
  end;
  with NullVect do begin
    x:=0;
    y:=0;
    z:=0;
  end;
  with NullPoint do begin
    Electric:=NullVect;
    Magnetic:=NullVect;
  end;
  with NullVectGrp do begin
    v0:=NullVect;
    v1:=NullVect;
    v2:=NullVect;
    v3:=NullVect;
    v4:=NullVect;
    v5:=NullVect;
    v6:=NullVect;
  end;
  with NullScalarGrp do begin
    s0:=0;
    s1:=0;
    s2:=0;
    s3:=0;
    s4:=0;
    s5:=0;
    s6:=0;
  end;
  with NullPointGrp do begin
    p0:=NullPoint;
    p1:=NullPoint;
    p2:=NullPoint;
    p3:=NullPoint;
    p4:=NullPoint;
    p5:=NullPoint;
    p6:=NullPoint;
  end;
  Form1.visible:=true;        {show the Form (user interface)}
  Image1.visible:=true;       {ensure the Image is visible}
  Screen:=0;                  {start by displaying bitmap1}
  DisplayScreen(Screen);      {make it visible}
  Time:=0;                    {set time to zero}
  Timestep_count:=0;
  MaxVal:=0;                  {ensure MaxVal is zero}
  E_Energy_Tot:=0;            {ensure Electric field energy total=0}
  B_Energy_Tot:=0;            {ensure Magnetic field energy total=0}
  New_AxisColours:='ALL';     {set axis colour display boxes to match
selections}
  UpdateDetails;              {Update all changed values & displayed
text}
  DoUpdate:=false;            {Initial conditions set up via FirstPass}
  FirstPass:=true;
```

```
  AllFields:=false;

  if not first then          {if first initialisation then Grid
zero'ed already}
    for scr:=0 to 1 do             {scan both copies of the grid's
points}
      for i:=0 to GridWidth-1 do     {and set all values to zero}
        for j:=0 to GridHeight-1 do
          for k:=0 to GridDepth-1 do
            points[scr,i,j,k]:=NullPoint;
  ReDraw:=true;               {ensure redraw is enabled}
end;

procedure TForm1.RecalcFields(scr:smallint);
var
  Current_Ex,Current_Ey,Current_Ez: extended;
  Current_Bx,Current_By,Current_Bz : extended;
  r,x,y,z,unit_x,unit_y,unit_z : extended;
  normal_x,normal_y,normal_z,dir_x,dir_y,dir_z : extended;
  Scalar_amp,Vector_amp,SpinConstant,E_amp : extended;
  NewScreen : smallint;
  xpos,ypos,zpos,midx,midy,midz:smallint;
  ThisGroup,NewGroup: PointGrp;
  vect,CurlVect,DivVect: vector;
  Scalar_Group: ScalarGrp;
  VectGrp: VectorGrp;
begin

  if scr=0 then NewScreen:=1 else NewScreen:=0;  {determine which data
to update}
  if not Flip_YZ then begin

  midx:=Trunc(GridWidth/2);
  midy:=Trunc(GridHeight/2);
  midz:=Trunc(GridDepth/2);

  /////////////////////////////////////


SpinConstant:=(Hhat/(2*ElectronMass));//*(ElectronCharge/(4*Pi*Permitt
ivity));

  /////////////////////////////////////

    for xpos:=0 to GridWidth-1 do begin            {scan grid's x
coords}
      for ypos:=0 to GridHeight-1 do begin         {scan grid's y
coords}
        for zpos:=0 to GridDepth-1 do begin        {scan grid's z
coords}
          ThisGroup:=PointGroup(scr, xpos, ypos, zpos);

          x:= xpos - midx;
          y:= ypos - midy;
          z:= zpos - midz;
```

```
          r:=sqrt( sqr(x) + sqr(y) + sqr(z) );
          if ( r < 0.00000000001 ) then r:=0.00000000001;

          unit_x:= x/r;
          unit_y:= y/r;
          unit_z:= z/r;

          r:=r*(ActualWidth/GridWidth);   // get actual distance in
metres

          /////////////////////////////////////
          ///  WAVE FUNCTION TO TEST
          ///
          ///  Note: i is the unit vector in the direction of r
          ///

          case StartOption of

          1: begin
              Scalar_amp := -
(ElectronCharge/(8*Pi*Permittivity))*ln(r);
              Vector_amp := Scalar_amp;
             end;

          end;

          ///
          ///
          /////////////////////////////////////

          // Hertzian Scalar Field is the Scalar part of the Wave
Equation Field
          with points[NewScreen,xpos,ypos,zpos] do begin
            Phi:=Scalar_amp;
          end;

          // Hertzian Field is the Vector part of the Wave Equation
          with points[NewScreen,xpos,ypos,zpos].Hertzian do begin
            x:=unit_x*Vector_amp;
            y:=unit_y*Vector_amp;
            z:=unit_z*Vector_amp;
          end;
        end;
      end;
    end;

    for xpos:=0 to GridWidth-1 do begin            {scan grid's x
coords}
      for ypos:=0 to GridHeight-1 do begin         {scan grid's y
coords}
        for zpos:=0 to GridDepth-1 do begin        {scan grid's z
coords}
          NewGroup:=PointGroup(scr, xpos, ypos, zpos);
```

```
          // ElectricPotential = grad of Hertzian Scalar Field
          VectGrp:=VectorGroup(NewGroup, HERTZIAN_FIELD);

points[NewScreen,xpos,ypos,zpos].ElectricPotential:=VectDiv(VectGrp);
        end;
      end;
    end;

  end;

  for xpos:=0 to GridWidth-1 do begin            {scan grid's x
coords}
    for ypos:=0 to GridHeight-1 do begin         {scan grid's y
coords}
      for zpos:=0 to GridDepth-1 do begin        {scan grid's z
coords}

        ThisGroup:=PointGroup(scr, xpos, ypos, zpos);
        NewGroup:=PointGroup(NewScreen, xpos, ypos, zpos);

        { ThisGroup's points are assigned as follows:    P3
P5
                                                      P1 P0 P2
                                                         P4        P6

                        Where P5 & P6 are in the Z plane (P5 at the
back and P6 at the front) }

        x:= xpos - midx;
        y:= ypos - midy;
        z:= zpos - midz;

        r:=sqrt( sqr(x) + sqr(y) + sqr(z) );
        if ( r < 0.00000000001 ) then r:=0.00000000001;

        unit_x:= x/r;
        unit_y:= y/r;
        unit_z:= z/r;

        r:=r*(ActualWidth/GridWidth);   // get actual distance in
metres

        if ( ViewTop ) then begin
          normal_x:=unit_y;
          normal_y:=-unit_x;
          normal_z:=unit_z;
        end
        else begin
          normal_x:=unit_z;
          normal_y:=unit_y;
          normal_z:=-unit_x;
        end;

        // Electric Field is: -div of ElectricPotential Field - d/dt
of Vector Potential field
```

```
        Scalar_Group:=ScalarGroup(NewGroup, ELECTRIC_POTENTIAL_FIELD);

points[NewScreen,xpos,ypos,zpos].Electric:=ScalarGrad(Scalar_Group);

        with points[NewScreen,xpos,ypos,zpos].Electric do begin
          // get amplitude of Static Electric field component
          E_amp:=sqrt( sqr(x) + sqr(y) + sqr(z) );
        end;

        // From Schrodinger's wave equation:
        //
        // VectorPotential = -i*SpinConstant*div(V)
        // SpinConstant = (Hhat/(2*ElectronMass))
        //
        // A is orthogonal to div(V) and also proportional to to
div(V)
        // note: div(V) = Laplacian(Z)
        with points[NewScreen,xpos,ypos,zpos].VectorPotential do begin
          x := -normal_x*SpinConstant*E_amp;
          y := -normal_y*SpinConstant*E_amp;
          z := -normal_z*SpinConstant*E_amp;
        end;

        with points[NewScreen,xpos,ypos,zpos].Electric do begin
          // E = -div(V) - (1/c)*dA/dt     { dA/dt = (A^2/r)   i.e.
acceleration of angular velocity }
          x := -x -
unit_x*(abs(sqr(points[NewScreen,xpos,ypos,zpos].VectorPotential.x)/r)
/SpeedOfLight);
          y := -y -
unit_y*(abs(sqr(points[NewScreen,xpos,ypos,zpos].VectorPotential.y)/r)
/SpeedOfLight);
          z := -z -
unit_z*(abs(sqr(points[NewScreen,xpos,ypos,zpos].VectorPotential.z)/r)
/SpeedOfLight);
        end;

        // Magnetic Field is Curl of Vector Potential Field
        VectGrp:=VectorGroup(NewGroup, VECTOR_POTENTIAL_FIELD);
        CurlVect:=VectCurl(VectGrp);

        with points[NewScreen,xpos,ypos,zpos].Magnetic do begin
          x:=Permeability*CurlVect.x;
          y:=Permeability*CurlVect.y;
          z:=Permeability*CurlVect.z;
        end;
      end;
    end;
  end;

end;

procedure TForm1.Propagate;
begin
  repeat
```

```
  Application.ProcessMessages;

  if (New_StartOption<>StartOption) or Restart then begin  {Restart
with new start option}
    Restart:=false;
    Time:=0;
    StartOption:=New_StartOption;
    Initialise(False);
    UpdateDetails;           {ensure the screen is displaying up-to-
date info}
    Auto_Scale(Screen); {determine the scaling factor required for
current data}
    UpdateDetails;           {ensure the screen is displaying up-to-
date info}
    UpdateBitmap(Screen); {Redraw the new data on the current
screen(bitmap)}
    DisplayScreen(Screen);   {Display the current screen}
  end;

  if DoUpdate then begin
    ReDraw:=false;           {Prevent re-draw unless specifically
asked to}
    UpdateDetails;           {ensure the screen is displaying up-to-
date info}

    if (not FreezeTime) or Flip_YZ then begin   {if time is running}

      if not GridTransformed then begin
        {Analyse the current grid & recalculate the new}
        {values for the E & H fields for the TimeStep}
        RecalcFields(Screen);
      end;

      if Screen=0 then Screen:=1 else Screen:=0;  {initiate screen
swap}
      ReDraw:=true;
    end;

    Auto_Scale(Screen);     {determine the scaling factor required for
current data}
    UpdateDetails;           {ensure the screen is displaying up-to-
date info}
    if ( Form1.Timesteps.Value <> 0 ) then begin
      if Redraw and (Timestep_count mod Form1.Timesteps.Value = 0)
then begin
        UpdateBitmap(Screen); {Redraw the new data on the current
screen(bitmap)}
        DisplayScreen(Screen);   {Display the current screen}
        Timestep_count:=0;
      end;
    end;

    DoUpdate:=false;
  end;
```

```
  if ((not FreezeTime) and (not GridTransformed)) and (StartOption<>0)
and (not Flip_YZ) and ( not AllFields or ( AllFields and (DisplayField
>= POWER_FLOW_FIELD))) then begin
     Time:=Time+TimeStep;  {if time is running, increase it by
TimeStep}
     Inc(Timestep_count);
     Inc(FrameCount);
     DoUpdate:=true;
     ReDraw:=true;
  end;

  Flip_YZ:=false;

  if ( AllFields ) then begin
    New_DisplayField := DisplayField + 1;
    if( New_DisplayField > POWER_FLOW_FIELD ) then New_DisplayField :=
1;

    DoUpdate:=true;
  end;


  until Quit;
  Application.Terminate;
end;

procedure TForm1.UpdateBitmap(scr:smallint);
{Starting with a new, black, 24bit Colour bitmap, this routine takes
all the }
{relevent selected display options into account and produces a
complete bitmap}
{image ready to be displayed to screen. The main display options it
considers }
{are: (a)which field to display. (b)Colour or Greyscale. (c)Vector
arrows on/off.}
{(d)Z Plane tiling on/off. (e)Colours for each axis.}
var
  colour : longint;
  i,j,k,count :smallint;
  xyzcol: byte;
  value : double;
  VectorType : boolean;
  ArrowsX,ArrowsY,OffsetX,OffsetY: smallint;
  ArrowsOn,FirstY: boolean;
  ThisBitmap: Tbitmap;
  ActualX,ActualY: smallint;
  pointX,pointY,Xfrac,Yfrac: extended;
  Xleft,Ytop,imax,jmax,linescan: smallint;
  v1,v2,v3,v4,ArrowVect,ColVect,vect: vector;
  p1,p2,p3,p4: point;
  Xstep,Ystep,step_int: smallint;
  GridX,GridY: extended;
  JpegImage: TJpegImage;
  OutStream: TFileStream;
```

```
  path,fname,zstr: string;
  Point1,Point2,CopyPoint1,CopyPoint2: Point;
  xpos,ypos,zpos : smallint;

begin

  if scr=0 then begin      {get a new bitmap}
    NewBitmap(@Bitmap1);
    ThisBitmap:=Bitmap1;
  end
  else begin
    NewBitmap(@Bitmap2);
    ThisBitmap:=Bitmap2;
  end;

  if TileZ then begin
    Xstep:=1;{round2(1/(TileScaleX*ScrScaleX))-1;   {Don't bother with
points which map to}
    Ystep:=1; {round2(1/(TileScaleY*ScrScaleY))-1;   {the same screen
pixel}
  end        {needs to be one less than theory to prevent beat freq}
  else begin
    Xstep:=1;                      {Full size picture so use every
point}
    Ystep:=1;
  end;

    if (TileZ or (save_frames and save_3D)) then k:=0 else k:=Z_Plane;
{set Z axis position. If Z Plane tiling,}
repeat

  Application.ProcessMessages;

  if ( save_frames and save_3D ) then begin
    if scr=0 then begin      {get a new bitmap}
      if (k = Z_Plane) then begin
        NewBitmap(@Bitmap1);
        ThisBitmap:=Bitmap1;
      end
      else begin
        NewBitmap(@Bitmap2);
        ThisBitmap:=Bitmap2;
      end;
    end
    else begin
      if (k = Z_Plane) then begin
        NewBitmap(@Bitmap2);
        ThisBitmap:=Bitmap2;
      end
      else begin
        NewBitmap(@Bitmap1);
        ThisBitmap:=Bitmap1;
      end;
    end;
  end;
```

```
  repeat                                {all Z values are drawn, one
at a time }
    VectorType:=false;    {initialize working variables}
    vect:=NullVect;
    ColVect:=NullVect;
    value:=0;
    OffsetX:=0;
    OffsetY:=0;
    colour:=0;

    for j:=0 to BitmapY-1 do   {build array of bitmap line pointers}
      YLinePtrs[j]:=ThisBitmap.ScanLine[j];
    IsGrey:=not ShowColour;    {set local boolean values}
    ArrowsOn:=Arrows and (VectorX.Checked or VectorY.checked or
VectorZ.checked);
    if (not TileZ) and ArrowsOn then begin {if vector arrows are
required, do some calculations}
      arrow_step:=VectSpacing;    {using spacing value from control on
Form}
      GridX:=(GridWidth+1)*ScrScaleX;     {width of active region in
pixels}
      GridY:=(GridHeight+1)*ScrScaleY;    {height of active region in
pixels}

      if Spacing_Metres.Checked then      {if spacing defined as 'n'
per meter}
        arrow_step:=(GridX/(GridWidth/PPM))/arrow_step; {rescale the
step value}

      ArrowsX:=Trunc(GridX/arrow_step);  {determine number of arrows
across screen}
      ArrowsY:=Trunc(GridY/arrow_step); {determine number of arrows
up/down screen}
      OffsetX:=round2(Frac(GridX/arrow_step)*arrow_step/2);  {calc
offsets for 1st arrow}
      OffsetY:=round2(Frac(GridY/arrow_step)*arrow_step/2);
      step_int:=Round2(arrow_step);
      if step_int=0 then Inc(step_int);
    end
    else step_int:=1; {prevent compiler giving 'uninitialised' warning
message}

   i:=0;
    repeat    {scan through each (x,y) in Z plane of grid}
      j:=0;
      with points[scr,i,j,k] do repeat
       vect:=VectorProperty(DisplayField,points[scr,i,j,k]);

       case DisplayField of   {depending on which field is required to
display}
          ELECTRIC_FIELD,
          MAGNETIC_FIELD,
          POWER_FLOW_FIELD,
          HERTZIAN_FIELD,
```

```
          VECTOR_POTENTIAL_FIELD: begin {Show Electric, Magnetic,
Power flow, Div Phi, Hertzian, Vector Potential or Electric Potential
Fields}
                   VectorType:=true;
                 end;
          E_ELECTRIC_FIELD: value:=E_Energy(VectSize(vect));   {Show
energy in Electric Field}
          E_MAGNETIC_FIELD: value:=B_Energy(VectSize(vect));   {Show
energy in Magnetic Field}
          HERTZIAN_SCALAR_FIELD: value:=points[scr,i,j,k].Phi;
          ELECTRIC_POTENTIAL_FIELD:
value:=points[scr,i,j,k].ElectricPotential;
        end;

        if VectorType then begin {convert component values to colour
values}
          ColVect:=VectToColours(vect,Amplification);
          with ColVect do begin   {Combine colours}

colour:=(Round2(abs(x))*X_RGB)+(Round2(abs(y))*Y_RGB)+(Round2(abs(z))*
Z_RGB);
            SignArray[i,j,0]:=Sign(x);
            SignArray[i,j,1]:=Sign(y);
            SignArray[i,j,2]:=Sign(z);
          end;
        end;
        if (not VectorType) or IsGrey then begin {if greyscale display
required}
          if VectorType then value:=VectSize(vect); {if vector use
vector's size}
          xyzcol:=ColourRange(value,Amplification);   {set equal RGB
colour values}
          colour:=xyzcol*RED+xyzcol*GREEN+xyzcol*BLUE;    {Combine
colours}
          SignArray[i,j,0]:=Sign(xyzcol);
          SignArray[i,j,1]:=Sign(xyzcol);
          SignArray[i,j,2]:=Sign(xyzcol);
        end;
        if Rendered then {if colour or grey rendered display req'd}
          ColourArray[i,j]:=colour;   {save point's colour in array}
        Inc(j,Ystep);
      until j>GridHeight-1;
      Inc(i,Xstep);
    until i>GridWidth-1;

      ZplanePos:=k; {set global variable for current Z plane used by
PlotPoint}

      i:=0;
      if Rendered then    {if colour or greyscale Rendered display
req'd}
        repeat
          j:=0;
          repeat
            PlotPoint(ThisBitmap,i,j);  {plot each point onto bitmap}
```

```
          Inc(j,Ystep);
        until j>GridHeight-1;
        Inc(i,Xstep);
      until i>Gridwidth-1;

    if TileZ then                  {if Z Plane tiling required}
      TileCursor(ThisBitmap,ZplanePos,clGray);

    if VectorType and ArrowsOn and (not TileZ) then begin  {if
Vector arrows required}

      if Spacing_gridpoints.Checked then begin

        if ( ScrScaleX > ScrScaleY ) then
arrow_step:=VectSpacing*ScrScaleY;
        if ( ScrScaleY >= ScrScaleX ) then
arrow_step:=VectSpacing*ScrScaleX;

        ArrowScale:=ArrowScaleFactor*arrow_step/$FF;    {set scaling
factor for arrow size}

        i:=0;
        repeat
          j:=0;
          repeat
            ActualX:=round2(GetRealX((i+0.5)));  {the actual screen
x location (on the image control)}
            ActualY:=round2(GetRealY((j+0.5)));  {the actual screen
y location (on the image control)}
            p1:=points[scr,i,j,k];
            ArrowVect:=VectorProperty(DisplayField,p1);
            with ArrowVect do begin
              x:=x*ReScaleFactor*ArrowScale;          {re-scale raw
values}
              y:=y*ReScaleFactor*ArrowScale;
              z:=z*ReScaleFactor*ArrowScale;
            end;
            DrawArrow(ThisBitmap,ActualX,ActualY,ArrowVect);
{draw the vector arrow}
            Inc(j,VectSpacing);
          until j>GridHeight-1;
          Inc(i,VectSpacing);
        until i>Gridwidth-1;

      end
      else begin

        ArrowScale:=ArrowScaleFactor*arrow_step/$FF;    {set scaling
factor for arrow size}

        i:=OriginX+OffsetX;      {calc screen x coord of arrow
start}
        imax:=round2(OriginX+GridWidth*ScrScaleX);   {calculate
screen coords}
```

```
          jmax:=round2(OriginY+GridHeight*ScrScaleY);  {of edge of
active region}
          repeat
            pointX:=(i-OriginX)/ScrScaleX;  {calc grid x coord for
screen (i,j) }
            Xleft:=Trunc(pointX);    {find x coord of top left point}
            Xfrac:=pointX-Xleft;     {calc fractional distance to next
point}
            j:=OriginY+OffsetY;    {calc screen y coord of arrow
start}
            repeat
              pointY:=(j-OriginY)/ScrScaleY; {calc grid y coord for
screen (i,j) }
              Ytop:=Trunc(pointY);   {find y coord of top left point}
              Yfrac:=pointY-Ytop;    {calc fractional distance to next
point}
              p1:=points[scr,Xleft,Ytop,k];
              if Xleft<(GridWidth-1) then      {get relevant point
vectors}
                p2:=points[scr,Xleft+1,Ytop,k] else p2:=NullPoint;
              if YTop<(GridHeight-1) then
                p3:=points[scr,Xleft,Ytop+1,k] else p3:=NullPoint;
              if (Xleft<(GridWidth-1)) and (YTop<(GridHeight-1)) then
                p4:=points[scr,Xleft+1,Ytop+1,k] else p4:=NullPoint;
              if not PointNull(p1) then
                v1:=VectorProperty(DisplayField,p1) else v1:=NullVect;
              if not PointNull(p2) then
                v2:=VectorProperty(DisplayField,p2) else v2:=NullVect;
              if not PointNull(p3) then
                v3:=VectorProperty(DisplayField,p3) else v3:=NullVect;
              if not PointNull(p4) then
                v4:=VectorProperty(DisplayField,p4) else v4:=NullVect;
              ArrowVect:=VectorInterpolate(v1,v2,v3,v4,Xfrac,Yfrac);
              with ArrowVect do begin
                x:=x*ReScaleFactor*ArrowScale;          {re-scale raw
values}
                y:=y*ReScaleFactor*ArrowScale;
                z:=z*ReScaleFactor*ArrowScale;
              end;
              DrawArrow(ThisBitmap,i,j,ArrowVect);    {draw the vector
arrow}
              Inc(j,step_int);
            until j>=jmax;     {do next arrow till done}
            Inc(i,step_int);
          until i>=imax;
        end;
      end;
      Inc(k);                {do next Z Plane}
  until (not TileZ) or (k>GridDepth-1); {only repeat if tiling req'd
and Z plane in range}

  if TileZ then   {draw a red rectangle around the currently selected
Z plane}
    TileCursor(ThisBitmap,Z_Plane,clRed);
```

```
  if MaintainAspect and (not TileZ) then with ThisBitmap.Canvas do
begin
    Pen.Width:=1;                        {set pen to thin white line}
    Pen.Style:=psSolid;
    Pen.Color:=clGray;
    if OriginX<>0 then begin                    {Draw boundary lines
for Grid within the }
      MoveTo(OriginX-2,0);          {Screen space if the Aspect ratio
is being}
      LineTo(OriginX-2,BitmapY);    {preserved and onlypart of the
screen space}
      MoveTo(BitmapX-OriginX,0);  {is active.}
      LineTo(BitmapX-OriginX,BitmapY);
    end;
    if OriginY<>0 then begin
      MoveTo(0,OriginY-2);
      LineTo(BitmapX,OriginY-2);
      MoveTo(0,BitmapY-OriginY);
      LineTo(BitmapX,BitmapY-OriginY);
    end;
  end;

  if ( save_frames ) then begin
    // convert to jpeg and save
    try
      JpegImage := TJpegImage.Create;
      JpegImage.Assign(ThisBitmap);

      path := '.\Frames\';

      case DisplayField of   {depending on which field is required to
display}
        HERTZIAN_SCALAR_FIELD:
          fname := 'HertzianScalar';
        ELECTRIC_POTENTIAL_FIELD:
          fname := 'ElecPotential';
        HERTZIAN_FIELD:
          fname := 'HertzianVect';
        VECTOR_POTENTIAL_FIELD:
          fname := 'VectPotential';
        ELECTRIC_FIELD:
          fname := 'Electric';
        MAGNETIC_FIELD:
          fname := 'Magnetic';
        E_ELECTRIC_FIELD:
          fname := 'E_Energy';
        E_MAGNETIC_FIELD:
          fname := 'B_Energy';
        POWER_FLOW_FIELD:
          fname := 'Power';
      end;

      if (FrameCount < 10) then fname:=fname+'00000'
      else if (FrameCount < 100) then fname:=fname+'0000'
      else if (FrameCount < 1000) then fname:=fname+'000'
```

```
      else if (FrameCount < 10000) then fname:=fname+'00'
      else if (FrameCount < 100000) then fname:=fname+'0';

      if (k < 10) then zstr:='00000'
      else if (k < 100) then zstr:='0000'
      else if (k < 1000) then zstr:='000'
      else if (k < 10000) then zstr:='00'
      else zstr:='0';

      if not DirectoryExists(path) then CreateDir(path);

      if ( save_3D ) then begin
        OutStream :=
TFileStream.Create(path+fname+IntToStr(FrameCount)+'_z'+zstr+IntToStr(
k)+'.jpg',fmOpenWrite or fmCreate);
      end
      else begin
        OutStream :=
TFileStream.Create(path+fname+IntToStr(FrameCount)+'.jpg',fmOpenWrite
or fmCreate);
      end;

      JpegImage.SaveToStream(OutStream);
    finally
      JpegImage.Free;
      OutStream.Free;
    end;
  end;

until (not (save_frames and save_3D)) or (k>GridDepth-1); {only repeat
if tiling req'd and Z plane in range}

end;

procedure TForm1.DisplayScreen(scr:smallint);  {display the current
bitmap}
begin
  if scr=0 then
    CurrentBitmap:= Bitmap1  {set the picture control to display
bitmap1}
  else
    CurrentBitmap:= Bitmap2;  {set the picture control to display
bitmap2}
  Image1.Picture.Graphic:=CurrentBitmap
end;

procedure TForm1.Start1Click(Sender: TObject);
{The 1st Start option has been selected}
begin
  New_StartOption:=1;
  DoUpdate:=true;
end;

procedure TForm1.Start2Click(Sender: TObject);
{The 2nd Start option has been selected}
```

```
begin
  New_StartOption:=2;
  DoUpdate:=true;
end;

procedure TForm1.start3Click(Sender: TObject);
{The 3rd Start option has been selected}
begin
  New_StartOption:=3;
  DoUpdate:=true;
end;

procedure TForm1.Start4Click(Sender: TObject);
{The 4th Start option has been selected}
begin
  New_StartOption:=4;
  DoUpdate:=true;
end;

procedure TForm1.Start5Click(Sender: TObject);
{The 5th Start option has been selected}
begin
  New_StartOption:=5;
  DoUpdate:=true;
end;

procedure TForm1.Start6Click(Sender: TObject);
{The 6th Start option has been selected}
begin
  New_StartOption:=6;
  DoUpdate:=true;
end;

procedure TForm1.Start7Click(Sender: TObject);
{The 7th Start option has been selected}
begin
  New_StartOption:=7;
  DoUpdate:=true;
end;

procedure TForm1.Start8Click(Sender: TObject);
{The 8th Start option has been selected}
begin
  New_StartOption:=8;
  DoUpdate:=true;
end;

procedure TForm1.Start9Click(Sender: TObject);
{The 9th Start option has been selected}
begin
  New_StartOption:=9;
  DoUpdate:=true;
end;

procedure TForm1.Start10Click(Sender: TObject);
```

```
{The 10th Start option has been selected}
begin
  New_StartOption:=10;
  DoUpdate:=true;
end;

function TForm1.sign(number: extended): shortint;
{This function takes a real number and returns either -1 or +1 }
{If: number<0, -1 is returned }
{    number>=0, +1 is returned }
var
  signval: shortint;
begin
  if number>=0 then signval:=1
  else signval:=-1;
  Result:=signval;
end;

function TForm1.Gradient(val1, val2: extended): extended;
{Determines the gradient from val1 to val2 assuming a horizontal
separation}
{between the points of one unit}
begin
  result:=Val2-Val1;
end;

function TForm1.Gradient_3point(val1, val2, val3: extended): extended;
{Determines the gradient from val1 to val2 assuming a horizontal
separation}
{between the points of one unit}
begin
  result:= (Gradient(val1, val2) + Gradient(val2, val3))/2;
end;

procedure TForm1.Field1Click(Sender: TObject);
{The Electric field has been selected to be displayed}
begin
  New_DisplayField:=ELECTRIC_FIELD;
  DoUpdate:=true;
end;

procedure TForm1.Field2Click(Sender: TObject);
{The Magnetic field has been selected to be displayed}
begin
  New_DisplayField:=MAGNETIC_FIELD;
  DoUpdate:=true;
end;

procedure TForm1.Field6Click(Sender: TObject);
{The Magnetic field has been selected to be displayed}
begin
//  New_DisplayField:=PHI_FIELD;
  New_DisplayField:=HERTZIAN_SCALAR_FIELD;
  DoUpdate:=true;
end;
```

```
procedure TForm1.Field7Click(Sender: TObject);
{The Magnetic field has been selected to be displayed}
begin
  New_DisplayField:=HERTZIAN_FIELD;
  DoUpdate:=true;
end;

procedure TForm1.Field8Click(Sender: TObject);
{The Magnetic field has been selected to be displayed}
begin
  New_DisplayField:=VECTOR_POTENTIAL_FIELD;
  DoUpdate:=true;
end;

procedure TForm1.Field9Click(Sender: TObject);
{The Magnetic field has been selected to be displayed}
begin
  New_DisplayField:=ELECTRIC_POTENTIAL_FIELD;
  DoUpdate:=true;
end;

procedure TForm1.Field3Click(Sender: TObject);
{The Electric field Energy has been selected to be displayed}
begin
  New_DisplayField:=E_ELECTRIC_FIELD;
  DoUpdate:=true;
end;

procedure TForm1.Field4Click(Sender: TObject);
{The Magnetic field Energy has been selected to be displayed}
begin
  New_DisplayField:=E_MAGNETIC_FIELD;
  DoUpdate:=true;
end;

procedure TForm1.Field5Click(Sender: TObject);
{The Power flow field has been selected to be displayed}
begin
  New_DisplayField:=POWER_FLOW_FIELD;
  DoUpdate:=true;
end;

procedure TForm1.TimeFreezeClick(Sender: TObject);
{The Freeze Time control has been changed, so toggle button's text}
begin
  if FreezeTime then New_FreezeTime:=false else New_FreezeTime:=true;
  DoUpdate:=true;
end;

function TForm1.Gauss(x: extended): extended;
{return a value (between 0 and -1) corresponding to a gaussian
distribution}
{based of the x value supplied. The whole curve (essentially) is
defined }
```

```
{between x=-2 to x=2}
begin
  result:=Exp(-sqr(x));
end;

procedure TForm1.UpdateDetails();
var
  ReCalcTileSize,ReCalcAspectRatio: boolean;
begin
  ReCalcTileSize:=false;          {Default re-calc to Off}
  ReCalcAspectRatio:=false;       {Default re-calc to Off}

  Flip_YZ:=New_Flip_YZ;
  New_Flip_YZ:=false;

  if ViewTop<>ViewFromTop.Checked then begin
    ViewTop := ViewFromTop.Checked;
    ReDraw:=true;                          {Trigger a screen re-draw}
  end;

  // If flipping Y & Z planes, need to make sure they have the same
dimensions first.
  if Flip_YZ then begin
    if GridHeight > GridDepth then New_GridHeight := GridDepth;
    if GridDepth > GridHeight then New_GridDepth := GridHeight;
  end;

  {Reset the Grid Dimensions if required (retains existing data)}
  if ((New_GridWidth<>GridWidth) or (New_GridHeight<>GridHeight) or
      (New_GridDepth<>GridDepth)) or FirstPass then begin
    ReAllocGridMemory;                     {Re-allocate memory for new
Grid size}
    GridWidth:=New_GridWidth;              {Adopt new Dimensions}
    GridHeight:=New_GridHeight;
    GridDepth:=New_GridDepth;

    if ( GridHeight <> GridDepth ) then ViewFromTop.Enabled := false
else ViewFromTop.Enabled := true;

    GridX.Text:=IntToStr(GridWidth);   {Ensure Controls reflect
Current}
    GridY.Text:=IntToStr(GridHeight);  {program settings.}
    GridZ.Text:=IntToStr(GridDepth);
    SetGridGlobals;                        {Re-Calc Globals for new Grid
size}
    ReCalcAspectRatio:=true;               {Trigger an Aspect Ratio
recalculation}
    ReCalcTileSize:=true;                  {Trigger a Tile Size
recalculation}
    ReDraw:=true;                          {Trigger a screen re-draw}
  end;

  {Update Time Related Displays}
  TimeDisplay.Text:='Time (Secs) : '+FloatToStr(Time); {update time
display}
```

```
  if (FreezeTime<>New_FreezeTime) or FirstPass then begin
    FreezeTime:=New_FreezeTime;
    with TimeFreeze do
      if FreezeTime then
        Caption:='Unfreeze Time'
      else
        Caption:='Freeze Time';
  end;

  {Update the Auto-Scale selection}
  if (New_AutoScale<>AutoScale) or FirstPass then begin
    if AutoScale=CONTINUAL then begin {Re-Enable control incase it was
out of}
      ReScale.Enabled:=true;          {range whilst autoscaling.}
      New_ReScale:=ReScaleFactor;     {Prevent the next update, as the
control}
      New_DisplayLevel:=Display_Level;{was turned off.}
    end;
    AutoScale:=New_AutoScale;         {Adopt the new control state.}
    case AutoScale of
      START:     begin
                   if Time=0 then begin
                     AutoWarn.Top:=Auto1.Top+2; {Show Auto-scale
warning indicator}
                     AutoWarn.Visible:=true;    {next to the relevant
button.}
                     AutoWarnTimer.Enabled:=true;
                   end
                   else begin
                     AutoWarn.Visible:=false;
                     AutoWarnTimer.Enabled:=false;
                   end;
                 end;
      CONTINUAL: begin
                   AutoWarn.Top:=Auto2.Top+2; {Show Auto-scale warning
indicator}
                   AutoWarn.Visible:=true;    {next to the relevant
button.}
                   AutoWarnTimer.Enabled:=true;
                 end;
      NEVER:     begin
                   AutoWarn.Visible:=false;
                   AutoWarnTimer.Enabled:=false;
                 end;
    end;
  end;

  {Turn off Auto Scale indicator after time started if req'd}
  if (Time<>0) and (AutoScale=START) then begin
    AutoWarn.Visible:=false;
    AutoWarnTimer.Enabled:=false;
  end;

  {Update the display mode Colour/GreyScale}
  if New_ShowColour<>ShowColour then begin
```

```
    ShowColour:=New_ShowColour;
    ReDraw:=true;                     {Trigger a screen re-draw}
  end;

  {Update Axis Colour Allocation}
  if UpdateColours or FirstPass then begin
    UpdateAxisColours(New_AxisColours);
    UpdateColours:=false;
    ReDraw:=true;                     {Trigger a screen re-draw}
  end;

  if (New_Rendered<>Rendered) or FirstPass then begin
    Rendered:=New_Rendered;
    if Rendered then begin
      RendGroup.Enabled:=true;
      RenderOption1.Enabled:=true;
      RenderOption2.Enabled:=true;
      RenderOption3.Enabled:=true;
      ColourButton.Enabled:=true;
      GreyScaleButton.Enabled:=true;
    end
    else begin
      RendGroup.Enabled:=false;
      RenderOption1.Enabled:=false;
      RenderOption2.Enabled:=false;
      RenderOption3.Enabled:=false;
      ColourButton.Enabled:=false;
      GreyScaleButton.Enabled:=false;
    end;
    ReDraw:=true;                     {Trigger a screen re-draw}
  end;

  {Update the Maintain Aspect Ratio Checkbox control}
  if (New_MaintainAspect<>MaintainAspect) or FirstPass then begin
    MaintainAspect:=New_MaintainAspect;
    ReCalcAspectRatio:=true;          {Trigger an Aspect Ratio
recalculation}
    ReCalcTileSize:=true;             {Trigger a Tile Size
recalculation}
    ReDraw:=true;                     {Trigger a screen re-draw}
  end;

  {Update the screen rendering Option}
  if (New_Render<>Render) or FirstPass then begin
    Render:=New_Render;
    ReCalcTileSize:=true;     {Reset Tiling size for new Rendering
selection}
    ReDraw:=true;                     {Trigger a screen re-draw}
  end;

  {Update Z Tiling Option Variable and button display}
  if (TileZ<>New_TileZ) or FirstPass then begin
    TileZ:=New_TileZ;
    ReDraw:=true;                     {Trigger a screen re-draw}
    with Z_Tiling do
```

```
      if TileZ then
        Caption:='Single Z Plane'
      else
        Caption:='Tile Z Planes';
  end;

  {Update Currently displayed Z plane, the control's value and its
display}
  if (Z_Plane<>New_ZPlane) or FirstPass then begin
    {Shift Tile Cursor if new tile selected}
    if TileZ then begin
      TileCursor(CurrentBitmap,Z_Plane,clGray);   {return old Tile's
border to Grey}
      TileCursor(CurrentBitmap,New_ZPlane,clRed); {Outline new Tile's
border in Red}
    end
    else ReDraw:=true;                     {Trigger a screen re-draw}
    Z_Plane:=New_ZPlane;
    ZPlane.Position:=Z_Plane+1;
    Z_Plane_Number.Text:='Z Plane :  '+IntToStr(Z_Plane+1)+'  (of
'+IntToStr(GridDepth)+')';
  end;

  if ReCalcAspectRatio then SetAspectRatio;    {Re-Calc if flagged}
  if ReCalcTileSize then SetTileSize;

  {Update the Re-scaling factor value and ensure control's value
agrees}
  if (ReScaleFactor<>New_ReScale) or FirstPass then begin
    ReScaleFactor:=New_ReScale;
    ReScale.Enabled:=true;
    ReDraw:=true;                         {Trigger a screen re-draw}
  end;

  {Update the DisplayLevel control's position}
  if (Display_Level<>New_DisplayLevel) or FirstPass then begin
    Display_Level:=New_DisplayLevel;
    if ((AutoScale=CONTINUAL) or ((AutoScale=START) and (Time=0)))
then
      DisplayLevel.Position:=Display_Level;
    ReDraw:=true;                         {Trigger a screen re-draw}
  end;

  if (Rate_Of_Time<>New_RateOfTime) or FirstPass then begin
    Rate_Of_Time:=New_RateOfTime;
    TimeStep:=(Rate_Of_Time/1000)*PointSize/SpeedOfLight; {Increment
of Time per iteration - Secs }
  end;

  {Calc the Amplification factor & update its display}
//  CentrePos:=Round2(DisplayLevel.Max/2);
  Amplification:=1000*(DisplayLevel.Max - Display_Level);
  AmpDisplay.Text:='Amplification :
'+IntToStr(Trunc(Amplification))+'%';
  Amplification:=ReScaleFactor*Amplification/100;
```

```
  {Update which Field is being shown}
  if DisplayField<>New_DisplayField then begin
    DisplayField:=New_DisplayField;
    ReDraw:=true;                      {Trigger a screen re-draw}
  end;

  {Update all the Field's statistical values}
  UpdateE_Energy(Screen);
  UpdateB_Energy(Screen);
  Energy1.Text:=FloatToStr(E_Energy_Tot); {display the Electric field
energy total}
  Energy2.Text:=FloatToStr(B_Energy_Tot); {display the Magnetic field
energy total}
  Energy3.Text:=FloatToStr(E_Energy_Tot+B_Energy_Tot); {display total
field energy}

  ActualGridWidth.Text:=FloatToStr(ActualWidth); {display actual size
in metres that grid represents}

  try
    New_VectSpacing:=StrToInt(VectorSpacing.Text);
  except                       {catch any invalid integer conditions}
    VectorSpacing.Text:='11';
    New_VectSpacing:=11;
  end;

  {Update the Vector Arrows' spacing value}
  if (New_VectSpacing<>VectSpacing) or FirstPass then begin
    VectSpacing:=New_VectSpacing;
    ReDraw:=true;                      {Trigger a screen re-draw}
  end;

  {The X,Y or Z vector arrows have been selected/de-selected, so re-
draw req'd}
  if VectorChange then begin
    VectorChange:=false;
    ReDraw:=true;
  end;

  {if the Arrows separation units have been changed, trigger a re-
draw}
  if ArrowsUnitsChange then begin
    ArrowsUnitsChange:=false;
    ReDraw:=true;
  end;

  {Update Arrows variable & status of Vector Arrows controls}
  if (Arrows<>New_Arrows) or FirstPass then begin
    Arrows:=New_Arrows;
    if Arrows then begin
      VectorX.Enabled:=true;
      VectorY.Enabled:=true;
      VectorZ.Enabled:=true;
      Spacing_Text.Enabled:=true;
```

```
      VectorSpacing.Enabled:=true;
      Spacing_pixels.Enabled:=true;
      Spacing_metres.Enabled:=true;
      Spacing_gridpoints.Enabled:=true;
    end
    else begin
      VectorX.Enabled:=false;
      VectorY.Enabled:=false;
      VectorZ.Enabled:=false;
      Spacing_Text.Enabled:=false;
      VectorSpacing.Enabled:=false;
      Spacing_pixels.Enabled:=false;
      Spacing_metres.Enabled:=false;
      Spacing_gridpoints.Enabled:=false;
    end;
    ReDraw:=true;                      {Trigger a screen re-draw}
  end;

  if (ArrowScaleFactor <> New_ArrowScaleFactor) or FirstPass then
begin
    ArrowScaleFactor := New_ArrowScaleFactor;
    ReDraw:=true;                      {Trigger a screen re-draw}
  end;

  FirstPass:=false;
end;

procedure TForm1.ZPlaneChange(Sender: TObject);
{The Z plane control has been changed, so re-validate display values}
begin
  New_ZPlane:=ZPlane.Position-1;
  DoUpdate:=true;
end;

procedure TForm1.DisplayLevelChange(Sender: TObject);
{The displaylevel slider control has been changed, so re-validate
display values}
begin
  New_DisplayLevel:=DisplayLevel.Position;
  DoUpdate:=true;
end;

procedure TForm1.ReScaleChange(Sender: TObject);
{The rescale factor control has been changed, so re-validate display
values}
begin
  try
    New_ReScale:=ReScale.value;
  except                       {catch any invalid integer conditions}
    New_ReScale:=ReScaleFactor;
  end;
  DoUpdate:=true;
end;

procedure TForm1.X_RedClick(Sender: TObject);
```

```
{The x axis display colour has been changed, so set it to new colour}
begin
  New_AxisColours:='X';
  UpdateColours:=true;
  DoUpdate:=true;
end;

procedure TForm1.X_GreenClick(Sender: TObject);
{The x axis display colour has been changed, so set it to new colour}
begin
  New_AxisColours:='X';
  UpdateColours:=true;
  DoUpdate:=true;
end;

procedure TForm1.X_BlueClick(Sender: TObject);
{The x axis display colour has been changed, so set it to new colour}
begin
  New_AxisColours:='X';
  UpdateColours:=true;
  DoUpdate:=true;
end;

procedure TForm1.X_noneClick(Sender: TObject);
begin
  New_AxisColours:='X';
  UpdateColours:=true;
  DoUpdate:=true;
end;

procedure TForm1.Y_RedClick(Sender: TObject);
{The y axis display colour has been changed, so set it to new colour}
begin
  New_AxisColours:='Y';
  UpdateColours:=true;
  DoUpdate:=true;
end;

procedure TForm1.Y_GreenClick(Sender: TObject);
{The y axis display colour has been changed, so set it to new colour}
begin
  New_AxisColours:='Y';
  UpdateColours:=true;
  DoUpdate:=true;
end;

procedure TForm1.Y_BlueClick(Sender: TObject);
{The y axis display colour has been changed, so set it to new colour}
begin
  New_AxisColours:='Y';
  UpdateColours:=true;
  DoUpdate:=true;
end;

procedure TForm1.Y_noneClick(Sender: TObject);
```

```
begin
  New_AxisColours:='Y';
  UpdateColours:=true;
  DoUpdate:=true;
end;

procedure TForm1.Z_RedClick(Sender: TObject);
{The z axis display colour has been changed, so set it to new colour}
begin
  New_AxisColours:='Z';
  UpdateColours:=true;
  DoUpdate:=true;
end;

procedure TForm1.Z_GreenClick(Sender: TObject);
{The z axis display colour has been changed, so set it to new colour}
begin
  New_AxisColours:='Z';
  UpdateColours:=true;
  DoUpdate:=true;
end;

procedure TForm1.Z_BlueClick(Sender: TObject);
{The z axis display colour has been changed, so set it to new colour}
begin
  New_AxisColours:='Z';
  UpdateColours:=true;
  DoUpdate:=true;
end;

procedure TForm1.Z_noneClick(Sender: TObject);
begin
  New_AxisColours:='Z';
  UpdateColours:=true;
  DoUpdate:=true;
end;

procedure TForm1.GreyscaleButtonClick(Sender: TObject);
{The greyscale display button has been selected, so set global
variable}
begin
  if GreyscaleButton.Checked then
    New_ShowColour:=false else New_ShowColour:=true;
  DoUpdate:=true;
end;

procedure TForm1.ColourButtonClick(Sender: TObject);
{The colour gradient display button has been selected, so set global
variable}
begin
  if ColourButton.Checked then
    New_ShowColour:=true else New_ShowColour:=false;
  DoUpdate:=true;
end;
```

```
procedure TForm1.UpdateAxisColours(which: string);
{Ensures that the colour rectangles showing the primary colour
currently}
{being used to depict that axis, are correct. If ALL is passed as a
parameter}
{all the colour bitmaps are updated. Likewise if X,Y or Z are passed
then only}
{that axis colour is updated.}
begin
  if (which='ALL') or (which='X') then begin
    if X_Red.Checked then X_RGB:=RED;      {only one of these can be
true}
    if X_Green.Checked then X_RGB:=GREEN;
    if X_Blue.Checked then X_RGB:=BLUE;
    if X_none.Checked then X_RGB:=BLACK;
    with X_Colour.Picture do
      case X_RGB of                        {point to correct bitmap}
        RED:   Graphic:=BitmapRed;
        GREEN: Graphic:=BitmapGreen;
        BLUE:  Graphic:=BitmapBlue;
        BLACK: Graphic:=BitmapBlack;
      end;
  end;
  if (which='ALL') or (which='Y') then begin
    if Y_Red.Checked then Y_RGB:=RED;      {only one of these can be
true}
    if Y_Green.Checked then Y_RGB:=GREEN;
    if Y_Blue.Checked then Y_RGB:=BLUE;
    if Y_none.Checked then Y_RGB:=BLACK;
    with Y_Colour.Picture do
      case Y_RGB of                        {point to correct bitmap}
        RED:   Graphic:=BitmapRed;
        GREEN: Graphic:=BitmapGreen;
        BLUE:  Graphic:=BitmapBlue;
        BLACK: Graphic:=BitmapBlack;
      end;
  end;
  if (which='ALL') or (which='Z') then begin
    if Z_Red.Checked then Z_RGB:=RED;      {only one of these can be
true}
    if Z_Green.Checked then Z_RGB:=GREEN;
    if Z_Blue.Checked then Z_RGB:=BLUE;
    if Z_none.Checked then Z_RGB:=BLACK;
    with Z_Colour.Picture do
      case Z_RGB of                        {point to correct bitmap}
        RED:   Graphic:=BitmapRed;
        GREEN: Graphic:=BitmapGreen;
        BLUE:  Graphic:=BitmapBlue;
        BLACK: Graphic:=BitmapBlack;
      end;
  end;
end;

procedure TForm1.Auto1Click(Sender: TObject);
{The AutoScale at Start Only button has been selected}
```

```
begin
  New_AutoScale:=START;
  DoUpdate:=true;
end;

procedure TForm1.Auto2Click(Sender: TObject);
{The AutoScale Continual button has been selected}
begin
  New_AutoScale:=CONTINUAL;
  DoUpdate:=true;
end;

procedure TForm1.Auto3Click(Sender: TObject);
{The AutoScale Never button has been selected}
begin
  New_AutoScale:=NEVER;
  DoUpdate:=true;
end;

procedure TForm1.Auto_Scale(scr: smallint);  {Calculate scaling factor
for data display}
{If AutoScale is not set to NEVER, then the colours (RGB values) are
scaled}
{such that the grid point of maximum absolute value for the quantity
currently}
{being displayed is given the maximum colour value (255 or $FF). The
other }
{points are given colours determined by a linear transition from zero
to that}
{maximum value. The absolute values are used for determining the
colour, so}
{that for example: -100 and +100 will be displayed as the same colour}
const
  PixMax=$FF;
begin
  if AutoScale<> NEVER then begin     {only autoscale if option
selected}
    if (AutoScale=CONTINUAL) or ((AutoScale=START) and (Time=0)) then
begin
      FindMaxVal(scr,DisplayField);  {Find the Maximum value of the
field concerned}

      if MaxVal<>0 then begin
        New_ReScale:=PixMax/MaxVal;
      end
      else begin
        New_ReScale:=0;            {if all screen points have zero
value,}
      end;                         {reset default scaling control
positions}

      DoUpdate:=true;
    end;
  end;
end;
```

```
procedure TForm1.MaxCheck(element: PointPtr);
begin
  with element.Electric do          {for Electric field of current
grid point}
    if Max_E<>0 then begin          {if a maximum allowable value is
defined}
      if x>Max_E then x:=Max_E;     {restrict the point's values to
within this}
      if y>Max_E then y:=Max_E;     {range of values}
      if z>Max_E then z:=Max_E;
      if x<-Max_E then x:=-Max_E;
      if y<-Max_E then y:=-Max_E;
      if z<-Max_E then z:=-Max_E;
    end;
  with element.Magnetic do          {for Magnetic field of current
grid point}
    if Max_B<>0 then begin          {if a maximum allowable value is
defined}
      if x>Max_B then x:=Max_B;     {restrict the point's values to
within this}
      if y>Max_B then y:=Max_B;     {range of values}
      if z>Max_B then z:=Max_B;
      if x<-Max_B then x:=-Max_B;
      if y<-Max_B then y:=-Max_B;
      if z<-Max_B then z:=-Max_B;
    end;
end;

function TForm1.VectSize(vect: Vector): extended;
{Calculate the total vector size by combining the three component
vectors}
begin
  with vect do
    Result:=sqrt(sqr(x)+sqr(y)+sqr(z));
end;

function TForm1.E_Energy(E_amp: extended): extended;
{calculate the energy density in the electric field at the point with
amplitude E_amp }
begin
  Result:=0.5*Permittivity*sqr(E_amp);
end;

function TForm1.B_Energy(B_amp: extended): extended;
{calculate the energy density in the magnetic field at the point with
amplitude B_amp }
begin
  Result:=0.5*sqr(B_amp)/Permeability;
end;

procedure TForm1.Vect_ArrowsClick(Sender: TObject);
{The Vector Arrows button has been changed, so either enable or
disable}
{all the vector option controls}
```

```
begin
  if Vect_Arrows.Checked then
    New_Arrows:=true
  else
    New_Arrows:=false;
  DoUpdate:=true;
end;

procedure TForm1.DrawArrow(ThisBitmap: Tbitmap;x,y: smallint;arrow:
Vector);
var      {draws a vector corresponding to the relevant x,y & z values}
         {x & y vectors are depicted as lines, z as crosses or
circles}
  xpos,ypos,zsize: smallint;
begin
  with ThisBitmap.Canvas do begin    {using the appropriate bitmap}
    with Pen do begin
      Width:=1;                          {set pen to thin white line}
      Color:=clWhite;
      Style:=psSolid;

      if VectorX.Checked then begin
        if (arrow.x > arrow_step) then begin
          arrow.x := arrow_step;
          Style:=psDot;
        end;

        if (arrow.x < -arrow_step) then begin
          arrow.x := -arrow_step;
          Style:=psDot;
        end;
      end;

      if VectorY.Checked then begin
        if (arrow.y > arrow_step) then begin
          arrow.y := arrow_step;
          Style:=psDot;
        end;

        if (arrow.y < -arrow_step) then begin
          arrow.y := -arrow_step;
          Style:=psDot;
        end;
      end;

      if VectorZ.Checked then begin
        if (arrow.z > arrow_step) then begin
          arrow.z := arrow_step;
          Style:=psDot;
        end;

        if (arrow.z < -arrow_step) then begin
          arrow.z := -arrow_step;
          Style:=psDot;
        end;
```

```
      end;
    end;

   MoveTo(x,y);                       {start the vector from the point
(x,y) }
    if VectorX.Checked then            {if x axis vectors required calc
xpos}
      xpos:=round2(x+(arrow.x))
    else xpos:=x;                      {else use x value of starting
point}
    if VectorY.Checked then            {if y axis vectors required calc
ypos}
      ypos:=round2(y+(arrow.y))
    else ypos:=y;                      {else use y value of starting
point}
    LineTo(xpos,Ypos);                 {draw the vector arrow for the x
and/or y displacement}
    if VectorZ.Checked then begin      {if z axis vectors required,}
      zsize:=round2(arrow.z);          {calc vector size}
      if arrow.z>0 then begin          {if z>0 (coming out of the page)
draw circle of appropriate size}
        MoveTo(x,y);
        brush.Style:=bsClear;          {ensure circle is not filled}
        Ellipse(x-zsize,y-zsize,x+zsize,y+zsize);
      end;
      if arrow.z<0 then begin          {if z<0 (going into the page)
draw cross of appropriate size}
        MoveTo(x-zsize,y-zsize);
        LineTo(x+zsize-1,y+zsize-1); {slight correction added to fix
arrow's}
        MoveTo(x-zsize,y+zsize);     {look - make both sides look
even}
        LineTo(x+zsize-1,y-zsize+1);
      end;
    end;
  end;
end;

procedure TForm1.NewBitmap(BmapPtr: BitmapPtr);
{given a pointer to a bitmap, erase the bitmap it points to (if any) }
{and return the pointer pointing to a new blank 24bit bitmap with }
{width and height the same as the Image control}
begin
  if BmapPtr<>nil then BmapPtr.free;        {free memory of current
bitmap if req'd}
  BmapPtr^:=TBitmap.Create;                 {create a new bitmap}
  if BmapPtr<> nil then with BmapPtr^ do begin                 {using
the new bitmap's pointer}
    Height:=Image1.Height;                  {set bitmap height}
    Width:=Image1.Width;                    {set bitmap width}
    Canvas.Brush.Color:=clBlack;
    Canvas.FloodFill(1,1,clBlack,fsBorder); {ensure black filled}
    PixelFormat:=pf24bit;                   {set 24bit colour}
  end;
end;
```

```
procedure TForm1.PlotPoint(Bmap: TBitmap; x, y: smallint);
{Using the colour array built by UpdateBitmap, display the colour
information}
{of the point (x,y) either as: (a) a extended pixel. (b) a circle of
uniform }
{colour with diametre equal to the smallest of ScrScaleX & ScrScaleY.
}
{ (c) a rectangle of width ScrScaleX and height ScrScaleY with the
original }
{colour value in the centre, and a graduated change to match the other
grid }
{points around it - the colour graduation is provided by
VectorInterpolate }
{and uses a linear interpolation in the X and Y directions for each of
the }
{three colour planes (Red Green Blue).}
var
  ScreenX,ScreenY: extended;
  Edge,PointNum: byte;
  DoQuad1,DoQuad2,DoQuad3,DoQuad4: boolean;
  TileOfsX,TileOfsY: smallint;
  newx,newy: smallint;
  Signs: array[1..9,1..3] of shortint;
  Colour: TColor;
  XCol,Yrow: Longint;
  Xtop,Ytop,Xbot,Ybot: extended;
  PixelRender: smallint;
  DrawPoint: boolean;
begin
  TileOfsX:=0;
  TileOfsY:=0;
  DrawPoint:=true;
  PixelRender:=Render;
  Colour:=ColourArray[x,y];
  begin
    if TileZ then begin      {if the Z planes are required to be
tiled}
      Yrow:=ZplanePos div TileXcount;
      Xcol:=ZplanePos mod TileXcount;
      TileRect:=TileGrid.CellRect(Xcol,Yrow);
      with TileRect do begin
        TileOfsX:=Left+EdgeSize;  {add a bit bypass tile's border}
        TileOfsY:=Top+EdgeSize;
      end;
      TileHalfX:=TileScrScaleX/2;        {calc midway points}
      TileHalfY:=TileScrScaleY/2;
      ScreenX:=TileOfsX+(x*TileScrScaleX)+TileHalfX; {calc x coord for
pixel}
      ScreenY:=TileOfsY+(y*TileScrScaleY)+TileHalfY; {calc y coord for
pixel}
      if ((TileScrScaleX<=1) and (TileScrScaleY<=1)) then
        PixelRender:=OneToOne; {prevent uneccesary work}
     { if (GridDepth>50) and (PixelRender=Blend) then
PixelRender:=Chunky;  }
```

```
        {Blend takes too long if too many Z Planes}
      if not ((ScreenX<TileRect.Right) and (ScreenY<TileRect.Bottom))
then
        DrawPoint:=false;    {Don't draw pixel if outside tile's
space}
    end
    else begin
      ScreenX:=GetRealX((x+0.5));  {the actual screen x location (on
the image control)}
      ScreenY:=GetRealY((y+0.5));  {the actual screen y location (on
the image control)}
      if ((ScrScaleX<=1) and (ScrScaleY<=1)) then
        PixelRender:=OneToOne; {prevent uneccesary work}
    end;
  if DrawPoint then
    if (PixelRender=OneToOne) then  {Display 1 point as one pixel}
      PlotPixel(Round2(ScreenX),Round2(ScreenY),Colour)
    else if PixelRender=Chunky then with Bmap.Canvas do begin  {if
'chunky' pixels of uniform colour req'd}
      if TileZ then begin
        Xtop:=ScreenX-TileHalfX;
        Ytop:=ScreenY-TileHalfY;
        Xbot:=Xtop+TileScrScaleX+1;
        Ybot:=Ytop+TileScrScaleY+1;
        if Xtop<TileOfsX then Xtop:=TileOfsX;
        if Xbot>=TileRect.Right then Xbot:=TileRect.Right-1;
        if Ytop<TileOfsY then Ytop:=TileOfsY;
        if Ybot<=TileRect.Bottom then Ybot:=TileRect.Bottom-1;
      end
      else begin
        Ytop:=ScreenY-halfY-1;
        Xtop:=ScreenX-halfX-1;
        Xbot:=Xtop+ScrScaleX+1;
        Ybot:=Ytop+ScrScaleY+1;
      end;
      Brush.Color:=Colour;

FillRect(rect(Round2(Xtop),Round2(Ytop),Round2(Xbot),Round2(Ybot)));
    end
    else if PixelRender=Blend then begin

{The colour graduation for the rectangle depicting the grid point in
question}
{is determined by examining the 8 grid points surrounding it and
noting the }
{Red, Green, and Blue values for each (and the point itself). A linear
}
{interpolation is then carried out in each quadrant around the central
point}
{The points are numbered as:     1   2   3
}
{                                4   5   6
}
{                                7   8   9
}
```

```
{Fill ColArray with combined RGB value for each point}
      Edge:=EdgeCase(x,y,0,0,GridWidth-1,GridHeight-1);  {determine
point's location}
      if EdgeArray[Edge,1,1]=1 then begin              {if not at top
left}
        ColArray[1]:=ColourArray[x-1,y-1];
        Signs[1,1]:=SignArray[x-1,y-1,0];
        Signs[1,2]:=SignArray[x-1,y-1,1];
        Signs[1,3]:=SignArray[x-1,y-1,2];
      end
      else ColArray[1]:=0;

      if EdgeArray[Edge,1,2]=1 then begin              {if not at top}
        ColArray[2]:=ColourArray[x,y-1];
        Signs[2,1]:=SignArray[x,y-1,0];
        Signs[2,2]:=SignArray[x,y-1,1];
        Signs[2,3]:=SignArray[x,y-1,2];
      end
      else ColArray[2]:=0;

      if EdgeArray[Edge,1,3]=1 then begin              {if not at top
right}
        ColArray[3]:=ColourArray[x+1,y-1];
        Signs[3,1]:=SignArray[x+1,y-1,0];
        Signs[3,2]:=SignArray[x+1,y-1,1];
        Signs[3,3]:=SignArray[x+1,y-1,2];
      end
      else ColArray[3]:=0;

      if EdgeArray[Edge,2,1]=1 then begin              {if not at left}
        ColArray[4]:=ColourArray[x-1,y];
        Signs[4,1]:=SignArray[x-1,y,0];
        Signs[4,2]:=SignArray[x-1,y,1];
        Signs[4,3]:=SignArray[x-1,y,2];
      end
      else ColArray[4]:=0;

      ColArray[5]:=ColourArray[x,y];                  {refers to point
itself}
      Signs[5,1]:=SignArray[x,y,0];
      Signs[5,2]:=SignArray[x,y,1];
      Signs[5,3]:=SignArray[x,y,2];

      if EdgeArray[Edge,2,3]=1 then begin              {if not at
right}
        ColArray[6]:=ColourArray[x+1,y];
        Signs[6,1]:=SignArray[x+1,y,0];
        Signs[6,2]:=SignArray[x+1,y,1];
        Signs[6,3]:=SignArray[x+1,y,2];
      end
      else ColArray[6]:=0;

      if EdgeArray[Edge,3,1]=1 then begin              {if not at
bottom left}
```

```
        ColArray[7]:=ColourArray[x-1,y+1];
        Signs[7,1]:=SignArray[x-1,y+1,0];
        Signs[7,2]:=SignArray[x-1,y+1,1];
        Signs[7,3]:=SignArray[x-1,y+1,2];
      end
      else ColArray[7]:=0;

      if EdgeArray[Edge,3,2]=1 then begin             {if not at
bottom}
        ColArray[8]:=ColourArray[x,y+1];
        Signs[8,1]:=SignArray[x,y+1,0];
        Signs[8,2]:=SignArray[x,y+1,1];
        Signs[8,3]:=SignArray[x,y+1,2];
      end
      else ColArray[8]:=0;

      if EdgeArray[Edge,3,3]=1 then begin             {if not at
bottom right}
        ColArray[9]:=ColourArray[x+1,y+1];
        Signs[9,1]:=SignArray[x+1,y+1,0];
        Signs[9,2]:=SignArray[x+1,y+1,1];
        Signs[9,3]:=SignArray[x+1,y+1,2];
      end
      else ColArray[9]:=0;


{Determine if it is worth doing calculation for each quadrant. If all
four points}
{which define the quadrant have a zero value, nothing need be done.}

DoQuad1:=not ((ColArray[1]=0) and (ColArray[2]=0) and (ColArray[4]=0)
and (ColArray[5]=0));
DoQuad2:=not ((ColArray[2]=0) and (ColArray[3]=0) and (ColArray[5]=0)
and (ColArray[6]=0));
DoQuad3:=not ((ColArray[4]=0) and (ColArray[5]=0) and (ColArray[7]=0)
and (ColArray[8]=0));
DoQuad4:=not ((ColArray[5]=0) and (ColArray[6]=0) and (ColArray[8]=0)
and (ColArray[9]=0));


   {   if TileZ then begin          {prevent Tile over-run}
   {     if EdgeArray[Edge,1,1]=0 then DoQuad1:=false;
        if EdgeArray[Edge,1,3]=0 then DoQuad2:=false;
        if EdgeArray[Edge,3,1]=0 then DoQuad3:=false;
        if EdgeArray[Edge,3,3]=0 then DoQuad4:=false;
      end;
{Process ColArray to provide signed R,G & B values for each point}
{each R,G or B value lies in the range -255 to +255 due to the
signing. }
{Once the interpolation has been done, the results can then be turned
into}
{an absolute value to give all positive values once more. The reason
for }
{this process is to get the correct gradient between points of equal
absolute}
```

```
{value but of different signs.}

      if (DoQuad1 or DoQuad2 or DoQuad3 or DoQuad4) then begin
        for PointNum:=1 to 9 do   {determine RGB values for each of
the 9 points}
          with PntArray[PointNum] do begin
            x:=abs(Signs[PointNum,1])*RGB_Val(ColArray[PointNum],RED);
{set x to point's Red value}

y:=abs(Signs[PointNum,2])*RGB_Val(ColArray[PointNum],GREEN);  {set y
to point's Green value}

z:=abs(Signs[PointNum,3])*RGB_Val(ColArray[PointNum],BLUE);   {set z
to point's Blue value}
          end;


{Call the routine which calculates and draws to screen the graduated
quadrants}
        if DoQuad1 then PlotQuadrant(Bmap,1,ScreenX,ScreenY);
        if DoQuad2 then PlotQuadrant(Bmap,2,ScreenX,ScreenY);
        if DoQuad3 then PlotQuadrant(Bmap,3,ScreenX,ScreenY);
        if DoQuad4 then PlotQuadrant(Bmap,4,ScreenX,ScreenY);
      end;
    end;
  end;
end;

function TForm1.RGB_Val(colour: Tcolor; primary: integer): byte;
{Given a combined RGB value (3 bytes of type TColor) and a parameter}
{determining which colour is required, this function returns the }
{byte value of that colour component.}
var
  Mask: integer;
begin
  Mask:=0;
  case primary of           {select appropriate mask value}
    RED:    Mask:=RedMask;
    GREEN:  Mask:=GreenMask;
    BLUE:   Mask:=BlueMask;
  end;
  if primary<>0 then
    Result:=round((colour and Mask)/primary)  {mask out unwanted
colours}
  else
    Result:=0;
end;

function TForm1.EdgeCase(x, y, Xmin, Ymin, Xmax, Ymax: smallint):
byte;
     {determines if the point (x,y) lies at the edge of the rectangle
}
     {defined by (Xmin,Ymin) and (Xmax,Ymax), and if so, what sort of
}
```

```
     {edge is it on (ie. left, right, bottom left corner etc...).
This}
     {information is required during calculations which require
comparisons}
     {between adjacent grid points, where such points may lie out of
the }
     {grid's bounds. The case values correspond to the following
locations:}
     {       (top left)  1   2   3
}
     {                   4   5   6
}
     {                   7   8   9  (bottom right)
}
var
  Xcase,Ecase: byte;
begin
   Ecase:=0;
   if x=Xmin then Xcase:=1        {calc X coord boundary state}
   else if x=Xmax then  Xcase:=3  {1=left boundary, 3=right boundary}
   else Xcase:=2;

   if y=Ymin then         {calc overall boundary state using y and
xcase}
     case Xcase of        {if y is at top boundary}
       1: Ecase:=1;         {x at left boundary}
       2: Ecase:=2;         {x not at a boundary}
       3: Ecase:=3;         {x at right boundary}
     end
   else if y=Ymax then    {if y is at bottom boundary}
     case Xcase of
       1: Ecase:=7;         {x at left boundary}
       2: Ecase:=8;         {x not at a boundary}
       3: Ecase:=9;         {x at right boundary}
     end
   else
     case Xcase of        {if y is not at a boundary}
       1: Ecase:=4;         {x at left boundary}
       2: Ecase:=5;         {x not at a boundary}
       3: Ecase:=6;         {x at right boundary}
     end;
  Result:=Ecase;          {report finding}
end;

function TForm1.VectorInterpolate(v1, v2, v3, v4: Vector; Xfrac,
  Yfrac: extended): Vector;
var
  NewVect: Vector;
begin
  NewVect.x:=Interpolate(v1.x,v2.x,v3.x,v4.x,Xfrac,Yfrac);
  NewVect.y:=Interpolate(v1.y,v2.y,v3.y,v4.y,Xfrac,Yfrac);
  NewVect.z:=Interpolate(v1.z,v2.z,v3.z,v4.z,Xfrac,Yfrac);
  Result:=NewVect;
end;
```

```
function TForm1.Interpolate(val1, val2, val3, val4, Xfrac,
  Yfrac: extended): extended;
{provide a value for a point somewhere with four other points which
define}
{the corners of a rectangle. The position of the point in question is
given as}
{a fraction along the x axis and a fraction along the y axis}
{The value is calculated by linear interpolation along each axis
direction}
var
  Xgrad,Xgrad1,Xgrad2,Ygrad2,newval: extended;
begin
  Xgrad1:=val2-val1;   {gradient along top of rectangle}
  Xgrad2:=val4-val3;   {gradient along bottom of rectangle}
  Ygrad2:=val3-val1;   {gradient along left of rectangle}
  Xgrad:=Xgrad1+Yfrac*(Xgrad2-Xgrad1);   {how x axis gradient varies
with y}
  newval:=val1+(Yfrac*Ygrad2)+(Xfrac*Xgrad); {top-left corner value
plus }
  Result:=newval;      {contributions from travel down y axis and
across x axis}
end;

procedure TForm1.PlotQuadrant(Bmap: TBitmap; Quadrant: byte; RealX,
RealY: extended);
{Use the PntArray configured by PlotPoint and the supplied variables
such as }
{Quadrant number (1,2,3 or 4) and actual (x,y) location for grid
point. The }
{four points defining the quadrant are sent to VectorInterpolate to
determine}
{the new RGB value for each pixel in the quadrant which is to be
plotted.}
{Note: only a quarter of the area defined by the quadrant is actually
}
{calculated and drawn, as adjacent points will do the other three
quarters}
{when they are calculated.}
var
  xstart,ystart,xend,yend,i,j: smallint;
  v1,v2,v3,v4,pix: Vector;
  colour: Tcolor;
  Xofs,Yofs: extended;
  Rval,Gval,Bval: byte;
  Xpos,Ypos: smallint;
  Width,Height,HalfWidth,HalfHeight: extended;
begin
  Xofs:=0;
  Yofs:=0;
  xstart:=0;
  ystart:=0;
  xend:=0;
  yend:=0;
  if TileZ then begin
    HalfWidth:=TileHalfX;
```

```
    HalfHeight:=TileHalfY;
    Width:=TileScrScaleX;
    Height:=TileScrScaleX;
  end
  else begin
    HalfWidth:=halfX;
    HalfHeight:=halfY;
    Width:=ScrScaleX;
    Height:=ScrScaleY;
  end;
{Determine boundary conditions for relevant Quadrant. The origin of
the four}
{Quadrants is condidered to be point (0,0) for the purposes of the
calculation}
  case Quadrant of                 {Quadrant 1}
    1: begin                       {x<=0, y<=0}
      xstart:=-Trunc(HalfWidth)-1;
      ystart:=-Trunc(HalfHeight)-1;
      xend:=0;
      yend:=0;
      Xofs:=Width;
      Yofs:=Height;
    end;                           {Quadrant 2}
    2: begin                       {x>0, y<=0}
      xstart:=0;
      ystart:=-Trunc(HalfHeight)-1;
      xend:=Trunc(Width-HalfWidth)+1;
      yend:=0;
      Xofs:=0;
      Yofs:=Height;
    end;                           {Quadrant 3}
    3: begin                       {x<=0, y>0}
      xstart:=-Trunc(HalfWidth)-1;
      ystart:=0;
      xend:=0;
      yend:=Trunc(Height-HalfHeight)+1;
      Xofs:=Width;
      Yofs:=0;
    end;                           {Quadrant 4}
    4: begin                       {x>0, y>0}
      xstart:=0;
      ystart:=0;
      xend:=Trunc(Width-HalfWidth)+1;
      yend:=Trunc(Height-HalfHeight)+1;
      Xofs:=0;
      Yofs:=0;
    end;
  end;

  if TileZ then with TileRect do begin   {Trim edges, so it fits
exactly}
    if (RealX+xstart)<=(Left+EdgeSize) then
xstart:=Round2(Left+EdgeSize-RealX);
    if (RealX+xend)>=(Right-1) then xend:=Round2(Right-1-RealX);
```

```
    if (RealY+ystart)<=(Top+EdgeSize) then
ystart:=Round2(Top+EdgeSize-RealY);
    if (RealY+yend)>=(Bottom-1) then yend:=Round2(Bottom-1-RealY);
  end;

  if Quadrant>2 then Inc(Quadrant); {Algorithm for selecting the
appropriate points}
  v1:=PntArray[Quadrant];
  v2:=PntArray[Quadrant+1];
  v3:=PntArray[Quadrant+3];
  v4:=PntArray[Quadrant+4];

  for j:=ystart to yend do begin {scan each horizontal line in the
Quadrant}
    Ypos:=Round2(RealY+j);       {Determine actual y coord for point
in Quadrant}
    for i:=xstart to xend do begin {scan along horizontal line in
Quadrant}
      {do the linear interpolation for each colour of point in
Quadrant}

pix:=VectorInterpolate(v1,v2,v3,v4,((i+Xofs)/Width),((j+Yofs)/Height))
;
      with pix do begin
        Rval:=byte(Round2(abs(x)));    {convert real values to
integers}
        Gval:=byte(Round2(abs(y)));    {round values to nearest
integer (down if 0.5)}
        Bval:=byte(Round2(abs(z)));
      end;
      {if greyscale display is required, the R,G & B values are all
equalized}
      {prior to building the ColourArray and calling the PlotPoint and
}
      {PlotQuadrant routines, so the resulting colour is guaranteed to
also}
      {be a greyscale.}

      Colour:=Tcolor((Rval*RED) + (Gval*GREEN) + (Bval*BLUE));
{combine values to give one RGB value}
      Xpos:=Round2(RealX+i);   {calc x coord of the point on the Image
control}
      PlotPixel(Xpos,Ypos,Colour);
    end;
  end;
end;

function TForm1.GetActualX(x: smallint): smallint;
{return the actual screen x coordinate (in the picture control) of the
x }
{coord of a point in the grid (assumes bitmap to be displayed at full
size)}
begin
  Result:=round2(GetRealX(x)); {ScrScaleX=number of pixels b/w points}
end;
```

```
function TForm1.GetActualY(y: smallint): smallint;
{return the actual screen y coordinate (in the picture control) of the
y }
{coord of a point in the grid (assumes bitmap to be displayed at full
size)}
begin
  Result:=round2(GetRealY(y)); {ScrScaleY=number of pixels b/w points}
end;

procedure TForm1.VectorSpacingChange(Sender: TObject);
{Redraw screen with different arrow spacing, unless Z plane tiling is
}
{on in which case arrows are not shown anyhow.}
var
  step: integer;
begin
  step:=VectSpacing;

  try
    step:=StrToInt(VectorSpacing.Text);
  except                        {catch any invalid integer conditions}
    VectorSpacing.Text:='';
  end;

  New_VectSpacing:=step;

  if not TileZ then DoUpdate:=true;
end;

procedure TForm1.SetTileSize;
{If the Z Plane tiling option is selected, each of the Z coordinate
X/Y planes}
{is to be displayed simultaneously on the Image control (screen). So,
as the }
{dimensions of the grid (including the number of Z planes) can vary,
the }
{optimum tile size must be calculated so that all planes can be
displayed }
{with their correct aspect ratios and at the largest possible tile
size}
{Note: A tile is a small version of one of the X/Y planes which are
normally}
{      displayed as a full screen image.}
var
  nx,ny: smallint;
  height: integer;
  TileAspect: extended;
begin
    if MaintainAspect then
      TileAspect:=Aspect
    else
      TileAspect:=ScreenAspect;
```

```
    TileX:=(BitmapX-EdgeSize*2);  {start with one tile the width of
the screen}
    repeat                        {TileX will be width of tile}
      nx:=(BitmapX-EdgeSize*2) div TileX;      {nx is the number of
tiles across the screen}
      TileY:=Trunc(TileAspect*TileX); {TileY will be height of tile}
      ny:=GridDepth div nx;       {ny is number of rows of tiles down
the screen}
      if (GridDepth mod nx)<>0 then Inc(ny);
      height:=ny*TileY;           {total height used by tiles}
      Dec(TileX);                 {Try smaller tile size}
{height must fit screen. Don't allow vanishingly small. Don't exceed
max Z plane}
    until (height<=(BitmapY-EdgeSize*2)) or (TileX<10) or
(nx>GridDepth);
    Inc(TileX);       {reverse the last decrement}
    if Render=OneToOne then begin    {If one to one representation
required}
      if TileX>GridWidth then begin  {restrict max tile size so that}
        TileX:=GridWidth;            {one grid point becomes one
pixel}
        TileY:=Trunc(TileAspect*TileX);
      end;
      if TileY>GridHeight then begin
        TileY:=GridHeight;
        TileX:=Trunc(TileY/TileAspect);
      end;
    end;
    TileXcount:=(BitmapX-EdgeSize*2) div TileX;  {determine number
across...}
    TileYcount:=GridDepth div TileXcount; {and down screen.}
    if GridDepth mod TileXcount<>0 then Inc(TileYcount); {add one for
an incmplete row}
    if MaintainAspect then begin
      TileScrScaleX:=(TileX-EdgeSize)/GridWidth;  {calc scaling
factors for reducing full}
      TileScrScaleY:=(TileY-EdgeSize)/GridHeight; {screen to tile size
(allow for border)}
    end
    else begin
      TileScrScaleX:=((TileX-EdgeSize)/BitmapX)*ScrScaleX;  {calc
scaling factors for reducing full}
      TileScrScaleY:=((TileY-EdgeSize)/BitmapY)*ScrScaleY;  {screen to
tile size (allow for border)}
    end;
    with TileGrid do begin
      ColCount:=TileXcount;
      RowCount:=TileYcount;
      DefaultColWidth:=TileX;
      DefaultRowHeight:=TileY;
    end;
end;

function TForm1.Round2(realval: extended): int64;
{This function provided an equal, consistent rounding of real numbers}
```

```
{such that they are always rounded to the nearest integer, and when }
{exactly 0.5 between integers, they are rounded towards zero.}
{The standard Round function provided rounds to the EVEN integer!!}
var
  intval: int64;
  fraction: extended;
begin
  intval:=Trunc(realval);           {first round real value towards
zero}
  fraction:=(realval-intval);       {subtract this result from the
real}
  if fraction>0.5 then Inc(intval)  {if the fraction>0.5, add one to
intval}
  else if fraction<-0.5 then Dec(intval); {cater for negative
fractions too}
  Result:=intval;                   {return the integer value
obtained}
end;

function TForm1.GetRealX(x: extended): extended;
{performs same function as GetActualX but returns a real value rather
than }
{an integer. This is so culuative rounding errors don't occur in some
calcs}
begin
  Result:=OriginX+(x*ScrScaleX);
end;

function TForm1.GetRealY(y: extended): extended;
{performs same function as GetActualY but returns a real value rather
than }
{an integer. This is so culuative rounding errors don't occur in some
calcs}
begin
  Result:=OriginY+(y*ScrScaleY);
end;

function TForm1.ColourRange(value: extended; ScaleFactor: extended):
byte;
{Convert a real value to a colour value (between 0 and 255)}
var
  ColourVal: smallint;
begin
  ColourVal:=abs(ByteLimit(value*ScaleFactor));
  Result:=byte(ColourVal);
end;

function TForm1.VectToColours(vect: vector; ScaleFactor: extended):
vector;
{Takes a vector of real numbers and converts it to a vector of colour}
{values (one for each component: x,y & z). Negative values become }
{positive colours.}
var
  ColourVect: Vector;
begin
```

```
  with ColourVect do begin
    x:=ColourRange(vect.x,ScaleFactor); {convert component values to
colour values}
    y:=ColourRange(vect.y,ScaleFactor);
    z:=ColourRange(vect.z,ScaleFactor);
  end;
  Result:=ColourVect;       {return a vector of colour values}
end;


function TForm1.ByteLimit(value: extended): smallint;
{Take a real value and return it as a smallint value between the
limits}
{-255 and +255}
var
  newval: smallint;
begin
  if abs(value)>$FF then        {if out of range, clip it at the
limit}
    newval:=Sign(value)*$FF
  else newval:=round2(value);
  Result:=newval;               {return as a smallint}
end;

function TForm1.VectByteLimit(vect: Vector): Vector;
{Process each component vector in vect such that the resulting
vector's}
{component vectors are all in the range -255 to 255}
var
  ResultVect: Vector;
begin
  with ResultVect do begin
    x:=ByteLimit(vect.x);
    y:=ByteLimit(vect.y);
    z:=ByteLimit(vect.z);
  end;
  Result:=ResultVect;
end;

function TForm1.VectorCross(v1, v2: Vector): Vector;
var
  ResultVect: Vector;
begin
  with ResultVect do begin
    x:=((v1.y)*(v2.z) - (v1.z)*(v2.y));
    y:=((v1.z)*(v2.x) - (v1.x)*(v2.z));
    z:=((v1.x)*(v2.y) - (v1.y)*(v2.x));
  end;
  Result:=ResultVect;
end;

function TForm1.PowerFlow(Apoint: point): vector;
{Calculate the Power flow vector at a point in the grid by taking }
{the vector cross product of the Electric & Magnetic fields, and  }
{re-scaling the result by the area of a grid point.}
```

```
var
  vect: Vector;
begin
  with Apoint do
    vect:=VectorCross(Electric,Magnetic);
  with vect do begin
    x:=x*PointArea;         {Rescale values by area of grid point}
    y:=y*PointArea;
    z:=z*PointArea;
  end;
  Result:=vect;
end;

function TForm1.VectorProperty(field: byte; Apoint: point): Vector;
{Return a vector quantity of the field which is required to be
displayed}
{using the Electric and Magnetic field vectors at a grid point.}
var
  Vect: Vector;
begin
  Vect:=NullVect;
  with Apoint do
    case Field of   {depending on which field is required to display}
      ELECTRIC_FIELD,E_ELECTRIC_FIELD: Vect:=Electric;       {Show
Electric Field}
      MAGNETIC_FIELD,E_MAGNETIC_FIELD: Vect:=Magnetic;       {Show
Magnetic Field}
      POWER_FLOW_FIELD: Vect:=PowerFlow(Apoint);             {Show
Power flow}
      HERTZIAN_FIELD: Vect:=Hertzian;
      VECTOR_POTENTIAL_FIELD: Vect:=VectorPotential;
    end;
  Result:=Vect;
end;

function TForm1.PointNull(Apoint: point): boolean;
{See if the point's information is all zeros}
begin
  with Apoint do
    Result:=(VectorNull(Electric) and VectorNull(Magnetic));
end;

function TForm1.VectorNull(vect: Vector): boolean;
{Look at each component vector to see if the vector is overall a}
{Null vector.}
begin
  with vect do
    if (x=0) and (y=0) and (z=0) then
      Result:=true else Result:=false;
end;

function TForm1.ReverseTColor(input: TColor): Tcolor;
{Function reverses the byte order of a TColor type. This is required}
{prior to storing in bitmap's memory using the pointers provided by}
{the ScanLine function.}
```

```
var
  Byte1,Byte2,Byte3: byte;
begin
  if input<>0 then begin
      Byte1:=PByteArray(@input)[0];
      Byte2:=PByteArray(@input)[1];
      Byte3:=PByteArray(@input)[2];
      Result:=Byte3*RED + Byte2*Green + Byte1*Blue;
    end
    else Result:=0;
end;

procedure TForm1.Z_TilingClick(Sender: TObject);
begin
   if TileZ then New_TileZ:=false else New_TileZ:=true;
   DoUpdate:=true;
end;

function TForm1.MouseZplane: smallint;
{If Z plane tiling is being used, return the tile number which}
{the mouse cursor is over. Zero returned if not over a tile.}
var
  x,y: longint;
  Coord: TPoint;
  Coord2: TGridCoord;
  Tile: smallint;
begin
  if TileZ then  begin
    with Form1 do begin
      x:=left+MainGroup.left+TileGrid.left+3;
      y:=top+(Height-ClientHeight)+MainGroup.top+TileGrid.top-3;
    end;
    Coord:=MyMouse.CursorPos;
    Coord2:=TileGrid.MouseCoord(Coord.x-x,Coord.y-y);
    Tile:=Coord2.x+Coord2.y*TileXcount;
    if (Tile>GridDepth-1) or (Tile<0) then Tile:=-1;       {ensure in
range}
    Result:=Tile;
  end
  else
    Result:=-1;
end;

procedure TForm1.Image1Click(Sender: TObject);
var
  Tile: smallint;
begin
  Tile:=MouseZplane;
  if Tile>=0 then New_ZPlane:=Tile;
  DoUpdate:=true;
end;

procedure TForm1.Image1DblClick(Sender: TObject);
var
  Tile: smallint;
```

```
begin
  Tile:=MouseZplane;
  if Tile>=0 then New_ZPlane:=Tile;
  New_TileZ:=false;
  DoUpdate:=true;
end;

procedure TForm1.VectorXClick(Sender: TObject);
begin
  VectorChange:=true;
  DoUpdate:=true;
end;

procedure TForm1.VectorYClick(Sender: TObject);
begin
  VectorChange:=true;
  DoUpdate:=true;
end;

procedure TForm1.VectorZClick(Sender: TObject);
begin
  VectorChange:=true;
  DoUpdate:=true;
end;

procedure TForm1.TileCursor(Bmap: TBitmap; Tile: smallint; colour:
TColor);
{Draw a coloured rectangle around the selected Z Plane Tile}
var
  Xcol,Yrow: Longint;
begin
  Yrow:=Tile div TileXcount;
  Xcol:=Tile mod TileXcount;
  with Bmap.Canvas do begin
    brush.Color:=colour;
    FrameRect(TileGrid.CellRect(Xcol,Yrow));
  end;
end;

procedure TForm1.Spacing_pixelsClick(Sender: TObject);
{Redraw screen with different arrow spacing, unless Z plane tiling is
}
{on in which case arrows are not shown anyhow.}
begin
  if not TileZ then begin
    ArrowsUnitsChange:=true;
    DoUpdate:=true;
  end;
end;

procedure TForm1.Spacing_metresClick(Sender: TObject);
{Redraw screen with different arrow spacing, unless Z plane tiling is
}
{on in which case arrows are not shown anyhow.}
begin
```

```
  if not TileZ then begin
    ArrowsUnitsChange:=true;
    DoUpdate:=true;
  end;
end;

procedure TForm1.Spacing_gridpointsClick(Sender: TObject);
{Redraw screen with different arrow spacing, unless Z plane tiling is
}
{on in which case arrows are not shown anyhow.}
begin
  if not TileZ then begin
    ArrowsUnitsChange:=true;
    DoUpdate:=true;
  end;
end;

procedure TForm1.PlotPixel(x, y: smallint; Colour: TColor);
{Using the current array of bitmap's horizontal line pointers, plot}
{a point of a certain 24bitr colour value into memory. The colour
must}
{by reversed prior to storage as the values are stored in byte
reversed}
{order.}
var
  YLine: PByteArray;
begin
  if (x>0) and (y>0) and (x<=BitmapX) and (y<=BitmapY) then begin
    Yline:=YLinePtrs[y-1];    {find array of colour values for line y
in bitmap}
    TColorPtr(@Yline[3*(x-1)])^:=ReverseTColor(Colour);    {get
pointer to 24bit RGB value for point (x,y) }
  end;
end;

procedure TForm1.RenderOption1Click(Sender: TObject);
begin
  if RenderOption1.Checked then begin
    New_Render:=OneToOne;
    DoUpdate:=true;
  end;
end;

procedure TForm1.RenderOption2Click(Sender: TObject);
begin
  if RenderOption2.Checked then begin
    New_Render:=Chunky;
    DoUpdate:=true;
  end;
end;

procedure TForm1.RenderOption3Click(Sender: TObject);
begin
  if RenderOption3.Checked then begin
    New_Render:=Blend;
```

```
    DoUpdate:=true;
  end;
end;

procedure TForm1.RendDisplayClick(Sender: TObject);
begin
  if RendDisplay.Checked then
    New_Rendered:=true
  else
    New_Rendered:=false;
  DoUpdate:=true;
end;

procedure TForm1.UpdateE_Energy(scr: smallint);
var
  i,j: smallint;
begin
  E_Energy_Tot:=0;
  for i:=0 to GridWidth-1 do
    for j:=0 to GridHeight-1 do with points[scr,i,j,Z_Plane] do
      E_Energy_Tot:=E_energy_Tot +
E_Energy(VectSize(Electric))*PointVolume; {adjust for unit volume};
end;

procedure TForm1.UpdateB_Energy(scr: smallint);
var
  i,j: smallint;
begin
  B_Energy_Tot:=0;
  for i:=0 to GridWidth-1 do
    for j:=0 to GridHeight-1 do with points[scr,i,j,Z_Plane] do
      B_Energy_Tot := B_Energy_Tot +
B_Energy(VectSize(Magnetic)*PointVolume); {adjust for unit volume}
end;

procedure TForm1.AspectControlClick(Sender: TObject);
begin
  if AspectControl.Checked then New_MaintainAspect:=true
  else New_MaintainAspect:=false;
  DoUpdate:=true;
end;

procedure TForm1.SetAspectRatio;
begin
  Aspect:=GridHeight/GridWidth;   {Aspect ratio of X/Y plane in grid}
  ScrScaleX:=BitmapX/GridWidth;   {how many times does grid width fit
screen?}
  ScrScaleY:=BitmapY/GridHeight;  {how many times does grid height fit
screen?}
  if MaintainAspect then          {if grid aspect ratio is to be
preserved,}
    if ScreenAspect>Aspect then   {then adjust scaling values to
allow}
      ScrScaleY:=ScrScaleX*Aspect {the largest image with the same
aspect ratio}
```

```
    else
      ScrScaleX:=ScrScaleY/Aspect;
  halfX:=ScrScaleX/2; {determine the number of pixels from one point}
  halfY:=ScrScaleY/2; {to half way to the next point (in x & y
directions)}
  {Determine bitmap coord's of the Origin (top left) of the }
  {active area of the screen (picture control) where the image will
start}
  OriginX:=Round2((BitmapX-(GridWidth*ScrScaleX))/2);
  OriginY:=Round2((BitmapY-(GridHeight*ScrScaleY))/2);
end;

procedure TForm1.GridXChange(Sender: TObject);
begin
  try
    StrToInt(GridX.Text);
  except                       {catch any invalid integer conditions}
    GridX.Text:='';
  end;
end;

procedure TForm1.GridYChange(Sender: TObject);
begin
  try
    StrToInt(GridY.Text);
  except                       {catch any invalid integer conditions}
    GridY.Text:='';
  end;
end;

procedure TForm1.GridZChange(Sender: TObject);
begin
  try
    StrToInt(GridZ.Text);
  except                       {catch any invalid integer conditions}
    GridZ.Text:='';
  end;
end;

procedure TForm1.AcceptGridSizeClick(Sender: TObject);
{Read the new Grid dimensions and change them to their new values}
{if they are sensible.}
begin
  try
    New_GridWidth:=StrToInt(GridX.Text);
    if New_GridWidth<3 then begin     {Keep it in range}
      GridX.Text:='3';
      New_GridWidth:=3;
    end;
  except                      {catch any invalid integer conditions}
    New_GridWidth:=GridWidth;
  end;
  try
    New_GridHeight:=StrToInt(GridY.Text);
    if New_GridHeight<3 then begin     {Keep it in range}
```

```
      GridY.Text:='3';
      New_GridHeight:=3;
    end;
  except                    {catch any invalid integer conditions}
    New_GridHeight:=GridHeight;
  end;
  try
    New_GridDepth:=StrToInt(GridZ.Text);
    if New_GridDepth<3 then begin     {Keep it in range}
      GridZ.Text:='3';
      New_GridDepth:=3;
    end;
  except                    {catch any invalid integer conditions}
    New_GridDepth:=GridDepth;
  end;
  DoUpdate:=true;
end;

procedure TForm1.SetGridGlobals;
{Set the values of various global variables which are dependent on the
Grid}
{size chosen.}
begin
  ActualWidth:=GridWidth/PPM;   {Actual width the screen models - in
metres}
  ActualHeight:=GridHeight/PPM;  {Actual Height the screen models - in
metres}
  ActualDepth:=GridDepth/PPM;    {Actual Depth the screen models - in
metres}
  PointArea:=(ActualWidth*ActualHeight)/(GridWidth*GridHeight); {Area
each pixel represents}
  PointVolume:=PointArea*ActualDepth/GridDepth; {volume each pixel
represents}
  ZPlane.Max:=GridDepth;          {set the ZPlane control's max
position}
  New_ZPlane:=round(GridDepth/2);  {default start point is mid point
in Z axis}
  LastZ.Caption:=IntToStr(GridDepth);  {show max Z plane value under
ZPlane control}
end;

function GetLargestFreeMemRegion(var AAddressOfLargest: pointer):
LongWord;
var
  Si: TSystemInfo;
  P, dwRet: LongWord;
  Mbi: TMemoryBasicInformation;
begin
  Result := 0;
  AAddressOfLargest := nil;
  GetSystemInfo(Si);
  P := 0;
  while P < LongWord(Si.lpMaximumApplicationAddress) do begin
    dwRet := VirtualQuery(pointer(P), Mbi, SizeOf(Mbi));
    if (dwRet > 0) and (Mbi.State and MEM_FREE <> 0) then begin
```

```
      if Result < Mbi.RegionSize then begin
        Result := Mbi.RegionSize;
        AAddressOfLargest := Mbi.BaseAddress;
      end;
      Inc(P, Mbi.RegionSize);
    end else
      Inc(P, Si.dwPageSize);
  end;
end;

procedure TForm1.ReAllocGridMemory;
{Try and allocate memory for the Grid of data points. If successful}
{Adopt the new Dimensions. Only newly allocated memory is initialised}
var
  NewSizeOK: boolean;
  scr,i,j,k: integer;
  BaseAddr: pointer;
  MemSize: LongWord;
begin
  MemSize := GetLargestFreeMemRegion(BaseAddr);
  NewSizeOK:=true;
  try
    SetLength(Points,2,New_GridWidth,New_GridHeight,New_GridDepth);
{set size of Grid}
  except
    NewSizeOK:=false;
  end;
  try
    SetLength(ColourArray,New_GridWidth,New_GridHeight);
  except
    NewSizeOK:=false;
  end;
  try
    SetLength(SignArray,New_GridWidth,New_GridHeight,3);
  except
    NewSizeOK:=false;
  end;
  if NewSizeOK then begin
    for scr:=0 to 1 do              {Ensure new space is initialised}
      for i:=0 to New_GridWidth-1 do
        for j:=0 to New_GridHeight-1 do
          for k:=0 to New_GridDepth-1 do
            if (i>GridWidth-1) or (j>GridHeight-1) or
            (k>GridDepth-1) then
              Points[scr,i,j,k]:=NullPoint;
  end
  else begin
    New_GridWidth:=GridWidth;     {Memory allocation failed, so }
    New_GridHeight:=GridHeight;   {return to previous Dimensions}
    New_GridDepth:=GridDepth;
  end;
end;

procedure TForm1.FormDestroy(Sender: TObject);
{When the Form closes, De-allocate all the dynamic array memory}
```

```
{allocated during the program's execution.}
begin
  Points:=nil;          {Free all memory allocated}
  ColourArray:=nil;
  SignArray:=nil;
end;

function TForm1.VectDiv(VectGroup: VectorGrp): extended;
{This function is the same as Del . Vect (Vector dot product) }
//
//        { VectGroup's points are assigned as follows:    P3
P5
//                                                      P1 P0 P2
//                                                         P4
P6
//
//                        Where P5 & P6 are in the Z plane  (P5 at the
back and P6 at the front) }
//
// Note: the Grid (0,0,0) point is at the Top, Left & Back point of
the Grid
//       So X increments to the right, Y increments downwards, & Z
increments out of the screen

{Perform the Vector Div function on the vector group passed to it.}
{The value returned (as a vector) is for the central point of the
group}
var
  dVx_dx,dVy_dy,dVz_dz: extended;
begin
  with VectGroup do begin
    dVx_dx:=((v2.x - v0.x)/dx + (v0.x - v1.x)/dx)/2;
    dVy_dy:=((v4.y - v0.y)/dy + (v0.y - v3.y)/dy)/2;
    dVz_dz:=((v6.z - v0.z)/dz + (v0.z - v5.z)/dz)/2;
  end;

  Result:=dVx_dx + dVy_dy + dVz_dz;
end;

function TForm1.VectCurl(VectGroup: VectorGrp): Vector;
// {This function is the same as Del x Vect (Vector cross product) }
//
// {Perform the Vector Curl function on the vector group passed to
it.}
// {The value returned (as a vector) is for the central point of the
group}
//
//        { VectGroup's points are assigned as follows:    P3
P5
//                                                      P1 P0 P2
//                                                         P4
P6
//
//                        Where P5 & P6 are in the Z plane  (P5 at the
back and P6 at the front) }
```

```
//
// Note: the Grid (0,0,0) point is at the Top, Left & Back point of
the Grid
//       So X increments to the right, Y increments downwards, & Z
increments out of the screen
//
var
  dVz_dy,dVy_dz,dVx_dz,dVz_dx,dVy_dx,dVx_dy: extended;
  CurlVect: Vector;
begin
  with VectGroup do begin
    dVz_dy:=((v4.z - v0.z)/dy + (v0.z - v3.z)/dy)/2;
    dVy_dz:=((v6.y - v0.y)/dz + (v0.y - v5.y)/dz)/2;
    dVx_dz:=((v6.x - v0.x)/dz + (v0.x - v5.x)/dz)/2;
    dVz_dx:=((v2.z - v0.z)/dx + (v0.z - v1.z)/dx)/2;
    dVy_dx:=((v2.y - v0.y)/dx + (v0.y - v1.y)/dx)/2;
    dVx_dy:=((v4.x - v0.x)/dy + (v0.x - v3.x)/dy)/2;
  end;

  with CurlVect do begin
    x:=dVz_dy - dVy_dz;
    y:=dVx_dz - dVz_dx;
    z:=dVy_dx - dVx_dy;
  end;

  Result:=CurlVect;
end;

function TForm1.PointGroup(scr, x, y, z: smallint): PointGrp;
{Return a group of point values for the point in question and all}
{the points immediately adjacent to it. If they are out of the }
{Grid, assign them zero values.}
{Assume the starting point (x,y,z) is valid}
var
  TheGroup: PointGrp;
  xless,yless,zless: boolean;
  xmore,ymore,zmore: boolean;
begin
  if x<=0 then xless:=true else xless:=false;
  if y<=0 then yless:=true else yless:=false;
  if z<=0 then zless:=true else zless:=false;

  if x>=(GridWidth-1) then xmore:=true else xmore:=false;
  if y>=(GridHeight-1) then ymore:=true else ymore:=false;
  if z>=(GridDepth-1) then zmore:=true else zmore:=false;

  { TheGroup's points are assigned as follows:    P3             P5
                                               P1 P0 P2
                                                  P4        P6

                   Where P5 & P6 are in the Z plane (P5 at the back
and P6 at the front) }

  with TheGroup do begin
    P0:=points[scr,x,y,z];
```

```
//    if xless then P1:=NullPoint else P1:=points[scr,x-1,y,z];
//    if xmore then P2:=NullPoint else P2:=points[scr,x+1,y,z];
//    if yless then P3:=NullPoint else P3:=points[scr,x,y-1,z];
//    if ymore then P4:=NullPoint else P4:=points[scr,x,y+1,z];
//    if zless then P5:=NullPoint else P5:=points[scr,x,y,z-1];
//    if zmore then P6:=NullPoint else P6:=points[scr,x,y,z+1];

    if xless then P1:=points[scr,x+1,y,z] else P1:=points[scr,x-
1,y,z];
    if xmore then P2:=points[scr,x-1,y,z] else
P2:=points[scr,x+1,y,z];
    if yless then P3:=points[scr,x,y+1,z] else P3:=points[scr,x,y-
1,z];
    if ymore then P4:=points[scr,x,y-1,z] else
P4:=points[scr,x,y+1,z];
    if zless then P5:=points[scr,x,y,z+1] else P5:=points[scr,x,y,z-
1];
    if zmore then P6:=points[scr,x,y,z-1] else
P6:=points[scr,x,y,z+1];
  end;
  Result:=TheGroup;
end;

function TForm1.VectorGroup(PntGroup: PointGrp; Field: smallint):
VectorGrp;
{Return a group of vectors for a particular field at, and around the }
{point in question.}
var
  VectGroup: VectorGrp;
begin
  with PntGroup do
    case Field of

      ELECTRIC_FIELD: begin
           VectGroup.v0:=p0.Electric;
           VectGroup.v1:=p1.Electric;
           VectGroup.v2:=p2.Electric;
           VectGroup.v3:=p3.Electric;
           VectGroup.v4:=p4.Electric;
           VectGroup.v5:=p5.Electric;
           VectGroup.v6:=p6.Electric;
         end;

      MAGNETIC_FIELD: begin
           VectGroup.v0:=p0.Magnetic;
           VectGroup.v1:=p1.Magnetic;
           VectGroup.v2:=p2.Magnetic;
           VectGroup.v3:=p3.Magnetic;
           VectGroup.v4:=p4.Magnetic;
           VectGroup.v5:=p5.Magnetic;
           VectGroup.v6:=p6.Magnetic;
         end;

      HERTZIAN_FIELD: begin
           VectGroup.v0:=p0.Hertzian;
```

```
          VectGroup.v1:=p1.Hertzian;
          VectGroup.v2:=p2.Hertzian;
          VectGroup.v3:=p3.Hertzian;
          VectGroup.v4:=p4.Hertzian;
          VectGroup.v5:=p5.Hertzian;
          VectGroup.v6:=p6.Hertzian;
        end;

      VECTOR_POTENTIAL_FIELD: begin
          VectGroup.v0:=p0.VectorPotential;
          VectGroup.v1:=p1.VectorPotential;
          VectGroup.v2:=p2.VectorPotential;
          VectGroup.v3:=p3.VectorPotential;
          VectGroup.v4:=p4.VectorPotential;
          VectGroup.v5:=p5.VectorPotential;
          VectGroup.v6:=p6.VectorPotential;
        end;

      else VectGroup:=NullVectGrp;
    end;
  Result:=VectGroup;
end;

function TForm1.ScalarGroup(PntGroup: PointGrp; Field: smallint):
ScalarGrp;
{Return a group of vectors for a particular field at, and around the }
{point in question.}
var
  ScalarGroup: ScalarGrp;
begin
  with PntGroup do
    case Field of

//      PHI_FIELD: begin
      HERTZIAN_SCALAR_FIELD: begin
          ScalarGroup.s0:=p0.Phi;
          ScalarGroup.s1:=p1.Phi;
          ScalarGroup.s2:=p2.Phi;
          ScalarGroup.s3:=p3.Phi;
          ScalarGroup.s4:=p4.Phi;
          ScalarGroup.s5:=p5.Phi;
          ScalarGroup.s6:=p6.Phi;
        end;

      ELECTRIC_POTENTIAL_FIELD: begin
          ScalarGroup.s0:=p0.ElectricPotential;
          ScalarGroup.s1:=p1.ElectricPotential;
          ScalarGroup.s2:=p2.ElectricPotential;
          ScalarGroup.s3:=p3.ElectricPotential;
          ScalarGroup.s4:=p4.ElectricPotential;
          ScalarGroup.s5:=p5.ElectricPotential;
          ScalarGroup.s6:=p6.ElectricPotential;
        end;

      else ScalarGroup:=NullScalarGrp;
```

```
    end;
  Result:=ScalarGroup;
end;

function TForm1.IntegrateScalarGrp(ScalarGroup: ScalarGrp): extended;
var
  Average: extended;
begin
  Average:=(ScalarGroup.s0 + ScalarGroup.s1 + ScalarGroup.s2 +
            ScalarGroup.s3 + ScalarGroup.s4 + ScalarGroup.s5 +
ScalarGroup.s6)/7;

  Result:=Average*(dx*dy*dz);
end;

procedure TForm1.FindMaxVal(scr, Field: smallint);
{Find the maximum absolute value of the quantity being displayed, so}
{that the colour levels can be adjusted to give maximum brightness
for}
{that value.}
var
  i,j,k: smallint;
  vect: Vector;
  VectorType: boolean;
  value: extended;
  Zstart,Zend: smallint;
begin
  VectorType:=false;
  value:=0;
  MaxVal:=0;                {Set it to zero first}
  if TileZ or scale_3D.Checked then begin    {If Z Planes are tiled
find Max of whole volume}
    Zstart:=0;
    Zend:=GridDepth-1;
  end
  else begin
    Zstart:=Z_Plane;     {If not, find Max of current plane only}
    Zend:=Z_Plane;
  end;

  for i:=0 to GridWidth-1 do
    for j:=0 to GridHeight-1 do
      for k:=Zstart to Zend do begin
        case Field of   {depending on which field is required to use}
          ELECTRIC_FIELD,
          MAGNETIC_FIELD,
          POWER_FLOW_FIELD,
          HERTZIAN_FIELD,
          VECTOR_POTENTIAL_FIELD: begin {calc Electric, Magnetic,
Power flow, Div Phi, Hertzian, Vector Potential or Electric Potential
Fields}
                   Vect:=VectorProperty(Field,points[scr,i,j,k]);
                   VectorType:=true;
                 end;
```

```
          E_ELECTRIC_FIELD:
value:=E_Energy(VectSize(points[scr,i,j,k].Electric));   {calc energy
in Electric Field}
          E_MAGNETIC_FIELD:
value:=B_Energy(VectSize(points[scr,i,j,k].Magnetic));   {calc energy
in Magnetic Field}
          HERTZIAN_SCALAR_FIELD: value:=points[scr,i,j,k].Phi;
          ELECTRIC_POTENTIAL_FIELD:
value:=points[scr,i,j,k].ElectricPotential;
        end;

        if VectorType then with vect do begin
          MaxVal:=Max(MaxVal,abs(x));
          MaxVal:=Max(MaxVal,abs(y));
          MaxVal:=Max(MaxVal,abs(z));
        end
        else
          MaxVal:=Max(MaxVal,abs(value));
      end;
end;

function TForm1.VectorDot(v1, v2: Vector): extended;
var
  DotProduct: extended;
begin
  DotProduct:=(v1.x*v2.x) + (v1.y*v2.y) + (v1.z*v2.z);
  Result:=DotProduct;
end;

function TForm1.ScalarGrad(ScalarGroup: ScalarGrp): Vector;
{This function is the same as delS }
{It gives the gradient vector of a Scalar field}
//
//        { VectGroup's points are assigned as follows:    P3
P5
//                                                      P1 P0 P2
//                                                         P4
P6
//
//                        Where P5 & P6 are in the Z plane  (P5 at the
back and P6 at the front) }
//
// Note: the Grid (0,0,0) point is at the Top, Left & Back point of
the Grid
//       So X increments to the right, Y increments downwards, & Z
increments out of the screen
var
  GradVect: Vector;
begin
  with ScalarGroup do begin
    GradVect.x:=-((s2-s0)/dx + (s0-s1)/dx)/2;
    GradVect.y:=-((s4-s0)/dy + (s0-s3)/dy)/2;
    GradVect.z:=-((s6-s0)/dz + (s0-s5)/dz)/2;
  end;
  Result:=GradVect;
```

```
end;

procedure TForm1.AutoWarnTimerTimer(Sender: TObject);
{If the auto-scale warning indicator's timer is active, toggle the}
{state of the warning indicator at each timer tick.}
begin
  AutoWarnState:=not AutoWarnState;
  if AutoWarnState then
    AutoWarn.Picture.Graphic:=BitmapRed
  else
    AutoWarn.Picture.Graphic:=BitmapBlack;
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Quit:=true;
end;

procedure TForm1.RateOfTimeChange(Sender: TObject);
begin
  New_RateOfTime:=RateOfTime.Position;
  DoUpdate:=true;
end;

procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  if CheckBox1.Checked then begin
    CheckBox2.Enabled:=true;
    save_frames:=true;
    FrameCount:=1;
  end
  else begin
    CheckBox2.Enabled:=false;
    save_frames:=false;
  end;
end;

procedure TForm1.CheckBox2Click(Sender: TObject);
begin
  if CheckBox2.Checked then begin
    save_3D:=true;
  end
  else begin
    save_3D:=false;
  end;
end;

procedure TForm1.CheckBox3Click(Sender: TObject);
begin
  if CheckBox3.Checked then begin
    AllFields:=true;
  end
  else begin
    AllFields:=false;
  end;
```

```
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  New_Flip_YZ:=true;
  DoUpdate:=true;
end;

procedure TForm1.ArrowScaleScrollChange(Sender: TObject);
begin
  New_ArrowScaleFactor := ArrowScaleScroll.Position;
  DoUpdate:=true;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Restart:=true;
  DoUpdate:=true;
end;

procedure TForm1.ViewFromTopClick(Sender: TObject);
begin
  DoUpdate:=true;
end;

procedure TForm1.Scale_3DClick(Sender: TObject);
begin
  DoUpdate:=true;
end;

End.
```