

DEVELOPMENT OF VISUAL EMU, A GRAPHICAL USER INTERFACE FOR THE
PERIDYNAMIC EMU CODE

by

JUSTIN BIRKEY

B.S., Kansas State University, 2006

A THESIS

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Mechanical and Nuclear Engineering
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2007

Approved by:

Major Professor
Dr. Daniel Swenson

Abstract

This thesis provides a description of Visual EMU, a graphical user interface for the peridynamic EMU code. The peridynamic model is a fundamental method for computational mechanical analysis that makes no assumption of continuous or small deformation behavior and has no requirement for the concepts of stress and strain. The model does not require spatial derivatives and instead uses integral equations. A force density function, called the pairwise force function, is postulated to act between each pair of infinitesimally small particles if the particles are closer together than some finite distance. A spatial integration process is employed to determine the total force acting upon each particle and a time integration process is employed to track the positions of the particles due to the applied body forces and applied displacements. EMU is a computer code developed by Sandia National Laboratories that implements the peridynamic model. Visual EMU is a pre-processor for the EMU code that allows any user to enter all parameters and visualize the resulting material regions, peridynamic grid, and a preview of resulting nodes. Visual EMU can be used before starting a lengthy solution with potential errors. The language, visual layout, and code design of Visual EMU are described along with two examples and their results.

Table of Contents

List of Figures	vi
Acknowledgements	ix
CHAPTER 1 - Introduction	1
Peridynamics	1
EMU	1
Visual EMU	2
Examples	2
CHAPTER 2 - Peridynamics	3
CHAPTER 3 - EMU	8
CHAPTER 4 - Visual EMU	11
Language	11
Visual layout	13
Main Window	13
Menu Bar	14
Tabbed Pane	15
Grid	16
Material	16
Geometry	18
Penetrator	19
Penetrator Data	20
Boundary Condition	20
General	21
Code design	25
Action	25
Command	27
Dialog	29
DisplayHelper	30
Images	31

Panel.....	31
Shapes	31
toFile().....	36
draw(String)	36
getType().....	36
isIncluded(double, double, double).....	37
getMaterial()	37
State.....	37
VE	38
Camera	39
Initialize	40
InterShape	41
MaterialData	42
ReadInFile.....	43
ShapeData	44
VisualEMUView.....	45
VisualEMUWindow	46
CHAPTER 5 - Examples	47
Sphere into Glass Plate	47
Read an EMU infile	47
Write an EMU infile	48
Results.....	48
Reset Visual EMU	48
Define the internal grid	49
Create a material	49
Create a material region	51
Manipulate the view.....	52
Create a penetrator	52
Set EMU solution parameters	53
Write an EMU infile	53
Run an EMU solution	54

EMU results	55
Small Pipe into Glass Plate	56
Results	56
Define the internal grid	57
Create materials	57
Create a material region	58
Manipulate the view	59
Add a grid file	59
Set EMU solution parameters	60
Write an EMU infile	61
Results	61
EMU Results	61
CHAPTER 6 - Conclusions	63
Summary	63
Future work	63
References	65
Appendix A - Infile results	66
Sphere into glass plate	66
Original Infile	66
Read/Write Infile	67
User Visual EMU infile	68
Small pipe into glass plate	69
Original infile	69
Read/Write Infile	69
User Visual EMU infile	70

List of Figures

Figure 2.1 Position definitions	4
Figure 2.2 Alternate force models (Silling, 2002)	7
Figure 3.1 Rectangle material region created internally (a) and externally (b)	9
Figure 4.1 Java shape hierarchy	12
Figure 4.2 Visual EMU main window layout	13
Figure 4.3 File submenu	15
Figure 4.4 Edit submenu	15
Figure 4.5 Help submenu	15
Figure 4.6 Tabbed pane showing grid	16
Figure 4.7 More dialog from the grid panel	16
Figure 4.8 Tabbed pane showing material	17
Figure 4.9 Default - Material dialog	17
Figure 4.10 More - Material dialog showing both tabs	18
Figure 4.11 Initial Conditions – Material dialog showing both tabs	18
Figure 4.12 Tabbed pane showing geometry	19
Figure 4.13 Tabbed pane showing penetrator	19
Figure 4.14 Tabbed pane showing penetrator data	20
Figure 4.15 Tabbed pane showing boundary condition	20
Figure 4.16 Velocity boundary condition dialog	21
Figure 4.17 Displacement boundary condition dialog	21
Figure 4.18 Tabbed Pane showing general	21
Figure 4.19 More – General dialog showing the Restart tab	22
Figure 4.20 More – General dialog showing the Output tab	23
Figure 4.21 More – General dialog showing the Interface tab	24
Figure 4.22 More – General dialog showing the Misc tab	24
Figure 4.23 NewAction UML	26

Figure 4.24 The undo and redo lists with three commands (a), undo action (b), and new command (c)	27
Figure 4.25 CommandManager UML	28
Figure 4.26 Penetrator types	33
Figure 4.27 View panel showing the grid boundary (black)	33
Figure 4.28 View panel with a slit plane (green).....	34
Figure 4.29 View panel with a precrack (green).....	34
Figure 4.30 Cylinder void shown in solid (left), wire (middle), and grid (right) frame views.....	35
Figure 4.31 Partial Shape UML	35
Figure 4.32 <i>toFile()</i> method from the Cylinder class	36
Figure 4.33 State UML	38
Figure 4.34 <i>rotateX(double)</i> method from the camera class.....	39
Figure 4.35 Camera UML.....	40
Figure 4.36 InterShape UML.....	41
Figure 4.37 <i>isEqual(String, String)</i> method from the InterShape class	42
Figure 4.38 Partial MaterialData UML.....	43
Figure 4.39 ReadInFile UML	44
Figure 4.40 Partial ShapeData UML	44
Figure 4.41 <i>drawShapes()</i> method in the ShapeData class.....	45
Figure 5.1 EMUGR plot of damage at time 0.....	47
Figure 5.2 Defining the internal grid	49
Figure 5.3 Defining the grid margin	49
Figure 5.4 Creating a new material.....	50
Figure 5.5 Define material properties 1	50
Figure 5.6 Define material properties 2	51
Figure 5.7 Creating a material region	51
Figure 5.8 Creating a penetrator	52
Figure 5.9 Changing the penetrator properties	53
Figure 5.10 Changing additional settings	53
Figure 5.11 Running an EMU solution.....	54
Figure 5.12 EMUGR code added to infile	55

Figure 5.13 EMUGR plot of damage at time $5.286e-5$	55
Figure 5.14 EMUGR plot of damage at time $2.646e-4$	56
Figure 5.15 EMUGR plot of damage at time 0.....	56
Figure 5.16 Setting the grid	57
Figure 5.17 Creating the first material.....	57
Figure 5.18 Creating the second material	58
Figure 5.19 Specifying an initial velocity.....	58
Figure 5.20 Creating a material region	59
Figure 5.21 Adding a grid file.....	59
Figure 5.22 Assigning Mat2 to material region 2.....	60
Figure 5.23 Changing additional settings	60
Figure 5.24 Disconnecting the material regions	61
Figure 5.25 EMUGR plot of damage at time $8.154e-5$	62
Figure 5.26 EMUGR plot of damage at time $1.816e-4$	62

Acknowledgements

I would first like to thank my major professor, Dr. Daniel Swenson, for his support, encouragement, and enthusiasm during my years at Kansas State University. I also thank Dr. Kevin Lease and Dr. Dunja Peric for serving on my committee. I thank my family for their love and support over the years. Most of all I thank my fiancée Amy for listening, encouraging, supporting, and praying for me always.

CHAPTER 1 - Introduction

This thesis provides a description of Visual EMU, a graphical user interface for the peridynamic EMU code. The interface is a pre-processor motivated by the desire to ease and spread the use of EMU. The following sections give an introduction to each chapter of this thesis in the order they will appear.

Peridynamics

The peridynamic model is a fundamental method for computational mechanical analysis that makes no assumption of continuous or small deformation behavior and has no requirement for the concepts of stress and strain. The model does not require spatial derivatives to be evaluated within the body and instead uses integral equations. Beginning with Newton's second law, a force density function, called the pairwise force function, is postulated to act between each pair of infinitesimally small particles if the particles are closer together than some finite distance, called the material horizon. The pairwise force function may be assumed to be a function of the relative position and the relative displacement between the two particles. A spatial integration process is employed to determine the total force acting upon each particle, and a time integration process is employed to track the positions of the particles due to the applied body forces and applied displacements.

EMU

EMU is a computer code developed by Sandia National Laboratories that implements the bond based theory of peridynamics by applying the peridynamic equations to a set of nodes. The nodes and solution parameters are entered through a keyword text file called an infile. After initializing the grid, the EMU code evaluates the peridynamic equations along with prescribed displacements and velocities between time steps to find the resulting displacement and velocity of each node. There is no feedback on EMU infile creation without attempting an EMU solution which may crash, quit during initialization, or complete the solution with unexpected results.

Visual EMU

Visual EMU is a pre-processor for the EMU code that allows a user to enter all keyword parameters and visualize the resulting material regions, peridynamic grid, and a preview of resulting nodes. No pre-processor currently exists. Additional features include: materials that are defined once and applied to any number of material regions, 3D visualization allowing the user to rotate, translate, and zoom, infiles can be read into Visual EMU to continue working or visualize the current setup before running an EMU solution, and the ability to run EMU from within Visual EMU.

Examples

Two examples are provided which show Visual EMU accurately reads and writes EMU infiles. The first example is a sphere impacting a cylindrical plate of glass at an angle. The second example is an externally generated material region in the shape of a small pipe impacting a rectangular glass plate normal to the surface. The user can also visualize the material regions and their placement relative to the peridynamic grid before performing the EMU solution. The example results from EMUGR show the accuracy of Visual EMU and the complex fracture possible with the use of the peridynamic EMU.

CHAPTER 2 - Peridynamics

Numerical prediction of crack growth is a longstanding problem in computational mechanics with difficulty arising from the basic incompatibility of cracks with the partial differential equations used in the classical theory of solid mechanics (Silling and Askari, 2004). A fundamental method for computational mechanical analysis has recently been introduced, called the peridynamic model (Silling, 1998; Silling, 2002; Macek and Silling, 2006; Silling et al., 2006). The model does not require spatial derivatives to be evaluated within the body and instead uses integral equations.

The following description of peridynamics comes from a research proposal for the Army. The peridynamic model makes no assumption of continuous or small deformation behavior. It has no requirement for the concepts of stress and strain. The peridynamic model starts with the assumption that Newton's second law holds true on every infinitesimally small freebody (or particle) within the domain of analysis. A force density function, called the pairwise force function, is postulated to act between each pair of infinitesimally small particles if the particles are closer together than some finite distance, called the material horizon. The pairwise force function may be assumed to be a function of the relative position and the relative displacement between the two particles. A spatial integration process is employed to determine the total force acting upon each particle, and a time integration process is employed to track the positions of the particles due to the applied body forces and applied displacements. One of the advantages of the peridynamic approach is that no finite element meshes are required. It is truly a meshless method.

As described by Silling (1998) and Macek and Silling (2006), the acceleration of any particle at \mathbf{x} in the reference configuration at time t is found from

$$\rho \ddot{\mathbf{u}}(\mathbf{x}, t) = \int_{H_x} \mathbf{f}(\mathbf{u}(\mathbf{x}', t) - \mathbf{u}(\mathbf{x}, t), \mathbf{x}' - \mathbf{x}) dV_{x'} + \mathbf{b}(\mathbf{x}, t), \quad (1)$$

where H_x is a neighborhood of \mathbf{x} , \mathbf{u} is the displacement vector field, \mathbf{b} is a prescribed body force density field, ρ is mass density, and \mathbf{f} is a pairwise force function whose value is the force vector (per unit volume squared) that the particle \mathbf{x}' exerts on particle \mathbf{x} . In the following

discussion, we denote the relative position of these two particles in the reference configuration by ξ :

$$\xi = \mathbf{x}' - \mathbf{x} \quad (2)$$

and their relative displacement by η :

$$\eta = \mathbf{u}(\mathbf{x}', t) - \mathbf{u}(\mathbf{x}, t) \quad (3)$$

Note that $\xi + \eta$ represents the current relative position vector connecting the particles, Figure 2.1.

The direct physical interaction (which occurs through unspecified means) between the particles at \mathbf{x} and \mathbf{x}' is called a *bond*, or in the special case of an elastic interaction to be defined, a *spring*. The concept of a bond that extends over a finite distance is a fundamental difference between the peridynamic theory and the classical molecular and discrete element theories (Potyondy and Cundall, 2004), which are based on the idea of contact forces that arise from interactions between particles that are in direct contact with each other.

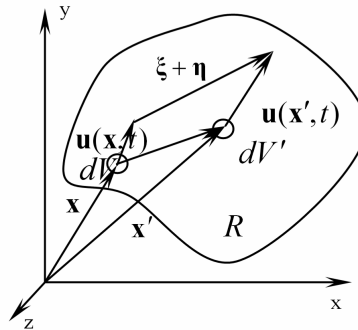


Figure 2.1 Position definitions

It is convenient to assume that for a given material that there is a *horizon*, δ , beyond which particles do not interact, or

$$|\xi| > \delta \Rightarrow \mathbf{f}(\eta, \xi) = 0 \quad \forall \eta \quad (4)$$

In this discussion, H_x will denote the spherical neighborhood of \mathbf{x} in R with radius δ .

The pairwise force function \mathbf{f} is required to have the following properties:

$$\mathbf{f}(-\eta, -\xi) = \mathbf{f}(\eta, \xi) \quad \forall \eta, \xi \quad (5)$$

which ensures conservation of linear momentum, and

$$(\eta + \xi) \times \mathbf{f}(\eta, \xi) = 0 \quad \forall \eta, \xi \quad (6)$$

which ensures conservation of angular momentum. The latter equation means that the force vector between any two particles is parallel to the particles' current relative position vector.

A material is said to be *microelastic* if the pairwise force function is derivable from a scalar *micropotential* w :

$$\mathbf{f}(\boldsymbol{\eta}, \boldsymbol{\xi}) = \frac{\partial w}{\partial \boldsymbol{\eta}}(\boldsymbol{\eta}, \boldsymbol{\xi}) \quad \forall \boldsymbol{\eta}, \boldsymbol{\xi} \quad (7)$$

The micropotential is the energy in a single bond and has dimensions of energy per unit volume squared. The energy per unit volume in the body at a given point (i.e., the local strain energy density) is therefore found from

$$W = \frac{1}{2} \int_{H_x} w(\boldsymbol{\eta}, \boldsymbol{\xi}) dV_{\boldsymbol{\xi}} \quad (8)$$

The factor of 1/2 appears because each endpoint of a bond “owns” only half the energy in the bond.

If a body is composed of a microelastic material, work done on it by external forces is stored in recoverable form in much the same way as in the classical theory of elasticity. Furthermore, it can be shown that the micropotential depends on the relative displacement vector $\boldsymbol{\eta}$ only through the scalar distance between the deformed points. Thus, there is a scalar-valued function \hat{w} such that

$$\hat{w}(y, \boldsymbol{\xi}) = w(\boldsymbol{\eta}, \boldsymbol{\xi}) \quad \forall \boldsymbol{\eta}, \boldsymbol{\xi}, \quad y = |\boldsymbol{\eta} + \boldsymbol{\xi}| \quad (9)$$

Therefore, the interaction between any two points in a microelastic material may be thought of as an elastic (and possibly nonlinear) spring. The spring properties may depend on the separation vector $\boldsymbol{\xi}$ in the reference configuration.

Combining Eqs. (7) and (9) and differentiating the latter with respect to the components of $\boldsymbol{\eta}$ leads to

$$\mathbf{f}(\boldsymbol{\eta}, \boldsymbol{\xi}) = \frac{\boldsymbol{\eta} + \boldsymbol{\xi}}{|\boldsymbol{\eta} + \boldsymbol{\xi}|} f(|\boldsymbol{\eta} + \boldsymbol{\xi}|, \boldsymbol{\xi}) \quad \forall \boldsymbol{\eta}, \boldsymbol{\xi} \quad (10)$$

where f is the scalar-valued function defined by

$$f(y, \boldsymbol{\xi}) = \frac{\partial \hat{w}}{\partial y}(y, \boldsymbol{\xi}) \quad \forall y, \boldsymbol{\xi}. \quad (11)$$

This satisfies the requirements of Eqs. (5) and (6), provided

$$\hat{w}(y, -\xi) = \hat{w}(y, \xi) \quad \forall y, \xi. \quad (12)$$

The relation shown in Eq. (11), together with the equation of motion, Eq. (1), contain the totality of the peridynamic model for a nonlinear microelastic material. In particular, note that the issue of how to treat rigid rotation does not arise in this formulation because y is invariant under rotation of the body. Similarly, objectivity of a constitutive model is not an issue in this approach.

The simplest material model is the proportional *microelastic* material, in which the bond force f varies linearly with *bond stretch* s ,

$$f(s, |\xi|) = \begin{cases} cs & \text{if } |\xi| < \delta \\ 0 & \text{otherwise} \end{cases} \quad (13)$$

where c is called the *spring constant* and $s = |\xi + \eta|/|\eta| - 1$.

To determine c from a given bulk modulus k , consider a large homogeneous body under isotropic extension, i.e., s is constant for all ξ , and $\eta = s\xi$. Defining $\xi = |\xi|$ and $\eta = |\eta|$, we have $\eta = s\xi$. Using the definition of the micropotential shown in Eq. (7), since $f = cs = c\eta/\xi$, it follows that $w = c\eta^2/2\xi = cs^2\xi/2$. Then, applying Eq. (8) leads to

$$W = \frac{1}{2} \int_H w dV_\xi = \frac{1}{2} \int_0^\delta \left(\frac{cs^2\xi}{2} \right) 4\pi\xi^2 d\xi = \frac{\pi cs^2 \delta^4}{4}. \quad (14)$$

This is required to equal the strain energy density in the classical theory of elasticity for the same material and the same deformation, $W = 9ks^2/2$. Combining this requirement with Eq. (14) leads to the spring constant in the proportional microelastic material model,

$$c = \frac{18k}{\pi\delta^4} \quad (15)$$

More complex behavior can be obtained using the microplastic or damage models shown in Figure 2.2. The microplastic model uses a 1D elastic-plastic behavior for each link. In the damage model, the links break after a specified amount of stretch. The appropriate failure stretch can be obtained by considering the fracture energy of a given material.

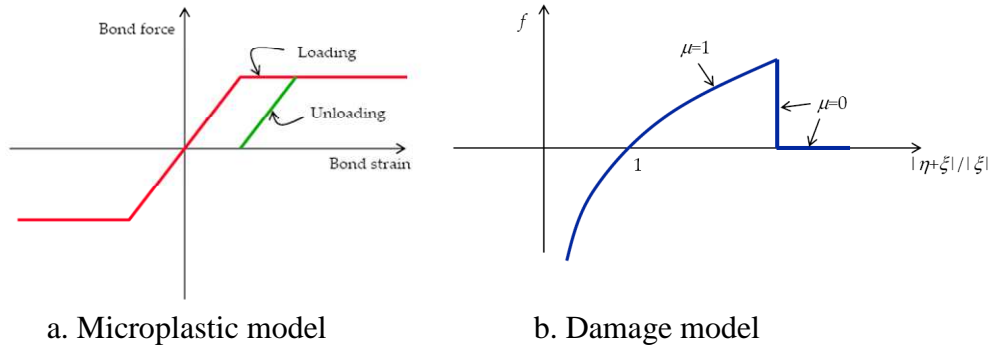


Figure 2.2 Alternate force models (Silling, 2002)

CHAPTER 3 - EMU

EMU, developed by Sandia National Laboratories, is a computer code based on peridynamics (Silling et al., 2006). EMU implements the bond based theory of peridynamics by applying the peridynamic equations to a set of nodes. The solution time is dependent on the number of nodes in the solution and the number of time steps. Multiprocessing can be used to reduce the solution time.

To begin a solution, the EMU code reads a keyword file, called an infile, and formulates the nodes defined through material regions and the number dropped, or deleted, through void regions. At each time step, the code takes each node in turn and finds the pairwise force functions described previously and also applies any prescribed displacements or velocities and short range forces necessary. After the resulting force for each node is found, new displacement and velocity values are calculated for the time step. The process continues until a stop condition is met.

EMU solutions are based on the nodes defined by material regions, which can be created in EMU or generated externally and read as part of the problem input. Internal generation is restricted to the volume defined by the peridynamic grid. Once the peridynamic grid is established, these are the only nodes available for any internally defined material region. Any material region defined outside of this grid has no nodes within its bounds and therefore no effect on the solution. A material region created across the boundary of the grid uses only the nodes within the grid as part of the solution. This can be confusing for the EMU user, as the grid is not defined by dimensions, but by keywords that specify the center point, the number of nodes in each direction, and the distance between nodes. The user must calculate the dimensions of the resulting grid to determine the boundary of possible nodes. Though not a complex calculation, a slight change to any of the three keywords that define the grid can accidentally place a material region outside the boundary. Externally generated nodes are independent of the peridynamic grid defined by EMU. These nodes are entered through a separate file called a grid file that defines the location and material of each node. The nodes are placed at the exact location specified in the grid file without regard to the peridynamic grid.

Nodes that are created too close to each other through external and internal generation are dropped before the solution begins. The distance that determines if a node should be dropped can be specified by the user. Nodes that get close to each other during a solution are not dropped but invoke short range forces that prevent them from occupying the same space. Short range forces apply to all nodes that come within the minimum distance and act to repel each other until the distance is greater than the minimum. Coefficients that help determine the force and minimum distance can be specified by the user.

For impact problems, EMU allows the user to create a special object called a penetrator. The penetrator is limited to one per solution and is not defined by nodes. The penetrator is a true solid object that interacts through contact with the surface. There are no peridynamic bonds related to the penetrator but forces still apply to the penetrator and the nodes that it comes into contact with. The penetrator is used primarily to impact material regions and the user can specify mass, angle of impact, angle of attack, impact velocity, tip location, friction and choose from five different shapes.

Some keywords in EMU can define a volume internally or externally. The two types specify whether the desired volume is inside or outside the boundary described by the keyword. Internal keywords specify the desired volume within the boundary. Figure 3.1 (a) shows a material region in the shape of a rectangle created internally. External keywords specify the desired volume outside the boundary yet inside the peridynamic grid. Figure 3.1 (b) shows the same material region boundary as (a) but created externally. The black lines in both figures represent the boundary of the peridynamic grid.

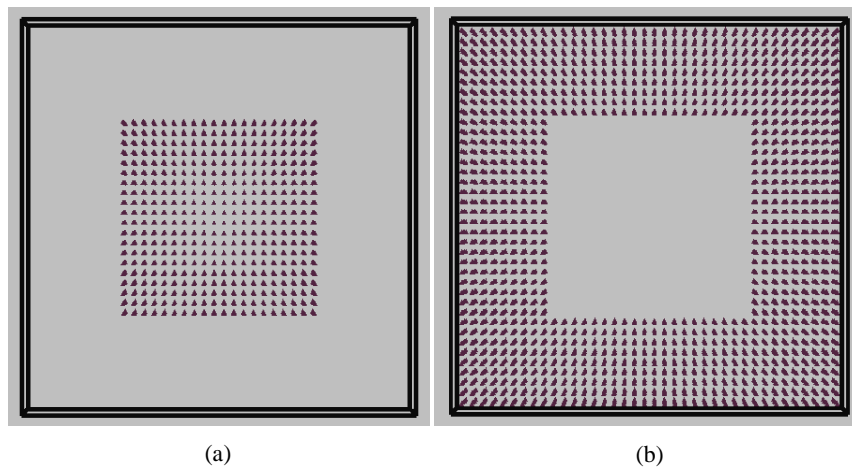


Figure 3.1 Rectangle material region created internally (a) and externally (b)

Execution of EMU requires the infile to contain keywords about the peridynamic grid, material regions, boundary conditions, and all other parameters that control the solution. Each keyword has a specific format and most are followed by a set of numbers. If the keyword signifies a single numerical value such as the maximum number of time steps, there is only one number that follows the keyword. If the keyword represents something more complex, such as a material region, the keyword itself contains an identifying number and is followed by a set of numbers that tell which geometry is being specified and providing the data needed to define that geometry.

To ensure that EMU runs properly, each keyword must follow the correct format. Some formatting errors, such as omitting a numerical value after a keyword, end EMU during initialization while others, such as switching the order of variables after a keyword, alter the desired results without warning. Errors that alter the results without ending EMU are hard to notice until the solution finishes, which can cost many hours of computing time. A pre-processor is clearly needed for the entry of these keywords and to allow visualization of material regions and the resulting nodes before running a lengthy EMU solution.

CHAPTER 4 - Visual EMU

Visual EMU is a graphical user interface for the peridynamic EMU, developed by Sandia National Laboratories, which is a computer code based on peridynamics (Silling et al., 2006). Visual EMU allows the user to input data, see the 3D model, and execute EMU from one interface. The following sections describe the language, visual layout, and code design of Visual EMU.

Language

Visual EMU is written entirely in Java using Eclipse as the development environment. Java is an object oriented language designed to divide programs into separate modules, called objects, which encapsulate the program's data and operations (Morelli and Walde, 2006). The objects are organized in a hierarchy from general to specific and can be broken down into more specific groups infinitely. Each class in the hierarchy inherits, or obtains the characteristics, from the class above it. A class is a template for an object and encapsulates the attributes and actions that characterize a certain type of object (Morelli and Walde, 2006). Java has a built in class called Object that all other classes inherit from, making it the most general object.

The following discussion uses the Visual EMU Shape class as an example. As you can see in Figure 4.1, Shape is a subclass of Object and therefore inherits all the characteristics of Object. Shape can be called an Object because it is a specific type of Object. Cylinder, Rectangle, and Sphere are all subclasses of Shape and therefore inherit the qualities of Shape as well as Object. Cylinder can be called an Object and it can be called a Shape. Cylinder can not be called a Rectangle however, as it does not inherit from Rectangle. Cylinder and Rectangle are both Shapes though, and specific types of Shapes. All classes that inherit from Object have a method, or collection of programming instructions that describe how to carry out a particular task (Horstmann, 2006), called *toString()*. Through inheritance, Shape, Cylinder, Rectangle, and Sphere all have a *toString()* method defined for them.

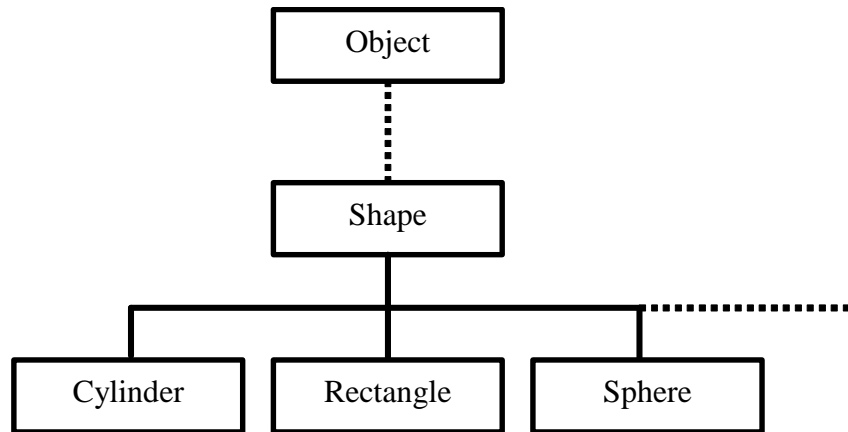


Figure 4.1 Java shape hierarchy

The *toString()* method can also be modified for each specific type, which is called polymorphism. Polymorphism denotes the principle that behavior can vary depending on the actual type of an object (Horstmann, 2006). Shape, Cylinder, Rectangle, and Sphere are all different types of Object. They can all override and change the *toString()* method so the method acts in a unique way for each class.

The power of object oriented programming is that all classes that inherit from Shape can be handled together as Shapes without the need to know which specific type is being used. For example, all classes that inherit from Shape have a method called *draw(String)*. This method displays the Shape in Visual EMU. The Cylinder, Rectangle, and Sphere are placed in a holder that only knows each is a Shape and nothing else. Thanks to inheritance, using the *draw(String)* method from each Shape displays the correct geometry without the need to find the specific type of Shape being drawn.

In Java, the command *extends* makes the class a subclass of the one specified by the command. The class is not only a subclass of the one specified, but a subclass of all super classes of the one specified. Using the example above, Object is a super class of Shape, Cylinder, Rectangle, and Sphere. A class inherits from all super classes. Cylinder extends Shape and is a subclass of any class above Shape up to the Object class. This creates a tree structure with the Object class as the root. The branches are all the subclasses of Object and the number of subclasses is unlimited. Each subclass of Object can also have an unlimited number of subclasses and this pattern continues indefinitely. Each class can only exist in one spot on the tree however, as each class can extend only one other class.

Classes can inherit from a class that is not a super class using the *implements* command. A class is not a subclass of what it implements (Morelli and Walde, 2006) and the location in the tree does not change. The interface, or implemented class, has methods that it requires each class that implements it to have. In this way, classes inherit methods from other classes they extend and implement.

Visual layout

Visual EMU follows the layout of most professional applications. The main window of Visual EMU has five sections, shown in Figure 4.2. Visual EMU is controlled through this window and the user can view different options by manipulating the main window. The options and their locations are described in this section.

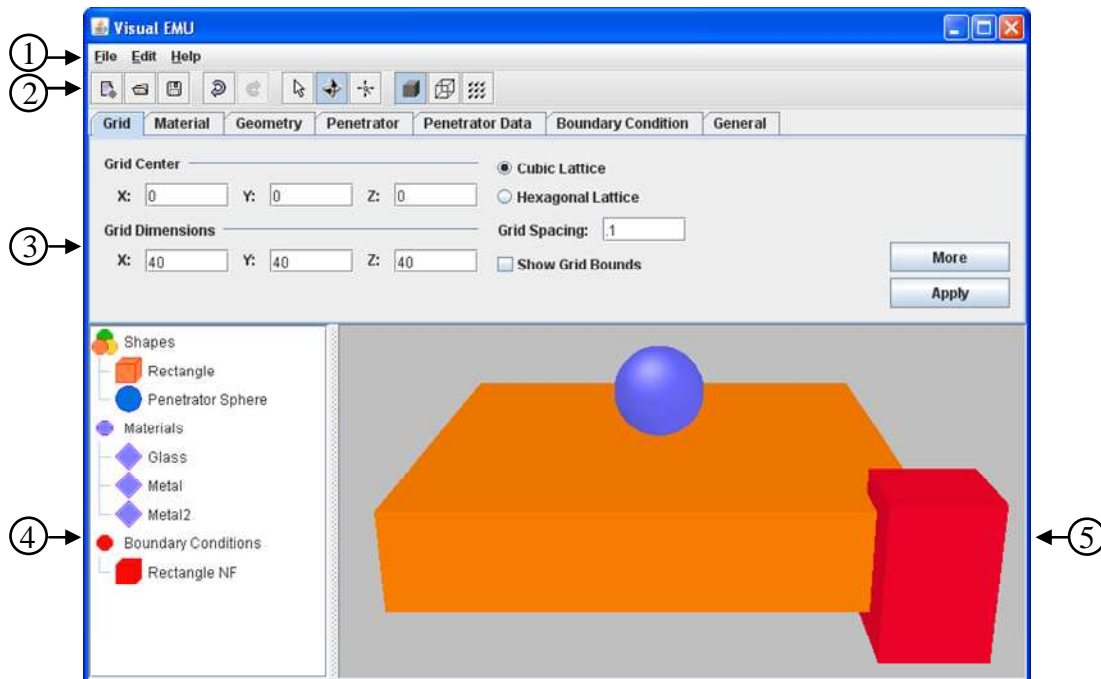


Figure 4.2 Visual EMU main window layout

Main Window

Located at the top of Visual EMU in section 1 is a menu bar with options such as File, Edit, and Help. Clicking these opens a submenu with more options. More on the menu bar is explained in the Menu Bar section. Below the menu bar in section 2 is a toolbar. Each button has a picture that represents the function and a short description appears when the mouse is

hovered, or held momentarily without clicking, over the button. Many of these buttons are the same options found in the menu bar above and provide quick access to the most popular choices. In addition, the toolbar contains buttons to control the 3D view located in section 5.

Section 3 of Figure 4.2, located below the toolbar, contains a tabbed pane. A tabbed pane shows different content depending on which tab is selected. In this way, the seven options in section 3 share the same space though only one is visible at a time. More on the tabbed pane is explained in the Tabbed Pane section. Below the tabbed pane, the rest of the main window is split into two parts. Section 4 on the left is a tree view of all Shapes, Materials, and Boundary Conditions. Through the rest of the discussion on Visual EMU, this section is referred to as the tree. Section 5 to the right of the tree is the 3D view. Anything the user can benefit from seeing in 3D is shown here, such as Shapes, Boundary Conditions, and the peridynamic grid boundary. Through the rest of the discussion on Visual EMU, section 5 is referred to as the view panel.

Menu Bar

The menu bar contains the controls for Visual EMU. Most options are held in the File submenu shown in Figure 4.3. The New option clears all of the current data in Visual EMU and allows the user to start with default settings as if the program has just been opened. The Open option allows the user to continue from a previously saved Visual EMU file. All of the settings from the file are applied to the current Visual EMU. The Save option creates a file with a “.vem” extension. The file holds the current Visual EMU information to allow the user to return to the current settings later. The Read Infile option allows the user to import a previously created EMU infile. The infile settings can then be viewed and manipulated as desired. The Write Infile option then allows the user to create an EMU infile from the current Visual EMU settings. The infile can be used to run an EMU solution or to save the current settings. The Run EMU option allows the user to begin an EMU solution from within Visual EMU. The solution can use the current Visual EMU settings or settings from an EMU input file. The Exit option closes Visual EMU.

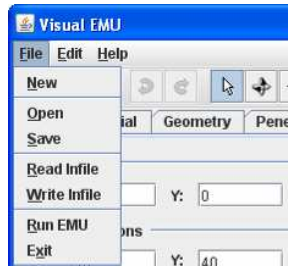


Figure 4.3 File submenu

The Edit submenu, shown in Figure 4.4, holds only two options: undo and redo. These options are also found on the toolbar represented by the blue arrows. They are explained in more detail in the Command section of Code Design.



Figure 4.4 Edit submenu

The Help submenu, shown in Figure 4.5, holds information about Visual EMU. Clicking on About opens a dialog that displays the version and creator of Visual EMU.



Figure 4.5 Help submenu

Tabbed Pane

The use of a tabbed pane in the main window allows the user to see more options without using additional windows. By selecting the title of a specific tab in section 3, that panel becomes visible and the previous is hidden. The user can navigate between tabs at any point in time without opening an additional window and covering the main window. The tabs are set in the order they are most likely to be used from left to right.

The panels held in the tabbed pane control almost all of the EMU keywords. Each panel controls a group of similar keywords. The most common keywords in the group are placed directly on the panel and the remainder are entered through modal dialog boxes that appear after pressing the appropriate button on the panel. All data from the dialog is saved with the

information on the panel it came from. This keeps all keyword information grouped together and allows easy access in one location. Each panel is explained in more detail in the following sections.

Grid

The grid panel, shown in Figure 4.6, contains all the keywords related to the nodes of the peridynamic grid mentioned previously. All internally generated material regions are dependent on this information and Visual EMU uses it to preview the configuration of nodes in EMU. Default values for the grid are initially set so the user can skip ahead if desired. The user can return to this panel at any time and change the information to better represent the chosen material regions.

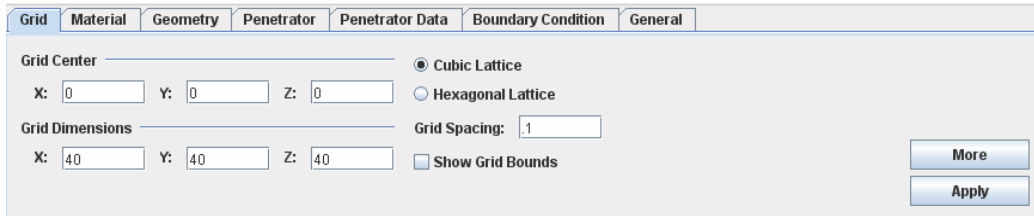


Figure 4.6 Tabbed pane showing grid

The More button on the right of the grid panel opens a modal dialog (Figure 4.7) for the user to enter data defining the grid margin. The grid margin is the area around the peridynamic grid that nodes can move into during the solution. Nodes that move outside the grid margin are dropped from calculation.

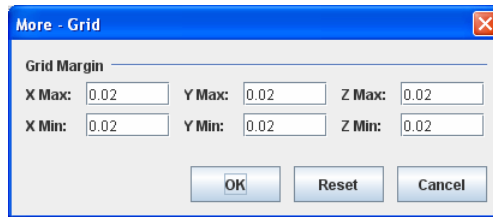


Figure 4.7 More dialog from the grid panel

Material

The material panel (Figure 4.8) contains all of the keywords related to material properties. Each material region requires a material be assigned to it. Materials are created or

edited in the material panel. There are three buttons on the lower right of the panel that open dialogs: Set Defaults, More, and Initial Conditions.

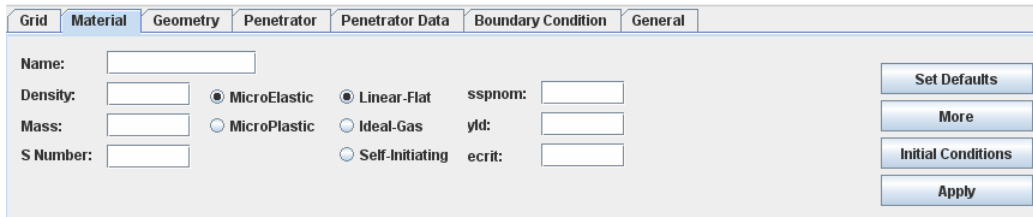


Figure 4.8 Tabbed pane showing material

The Set Defaults button opens the dialog shown in Figure 4.9. The information entered here applies to all materials and is the value used by EMU unless specified otherwise for a certain material.

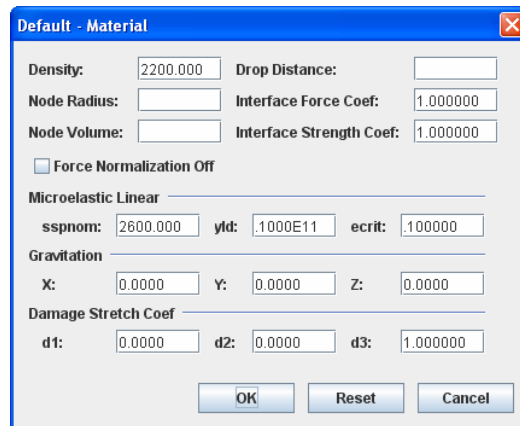


Figure 4.9 Default - Material dialog

The More button opens a dialog for the entry of additional material information. The dialog contains a tabbed pane just like the main window that allows the information to be viewed in two parts, keeping the dialog reasonably sized and organized. The dialog with each tab selected is shown in Figure 4.10.

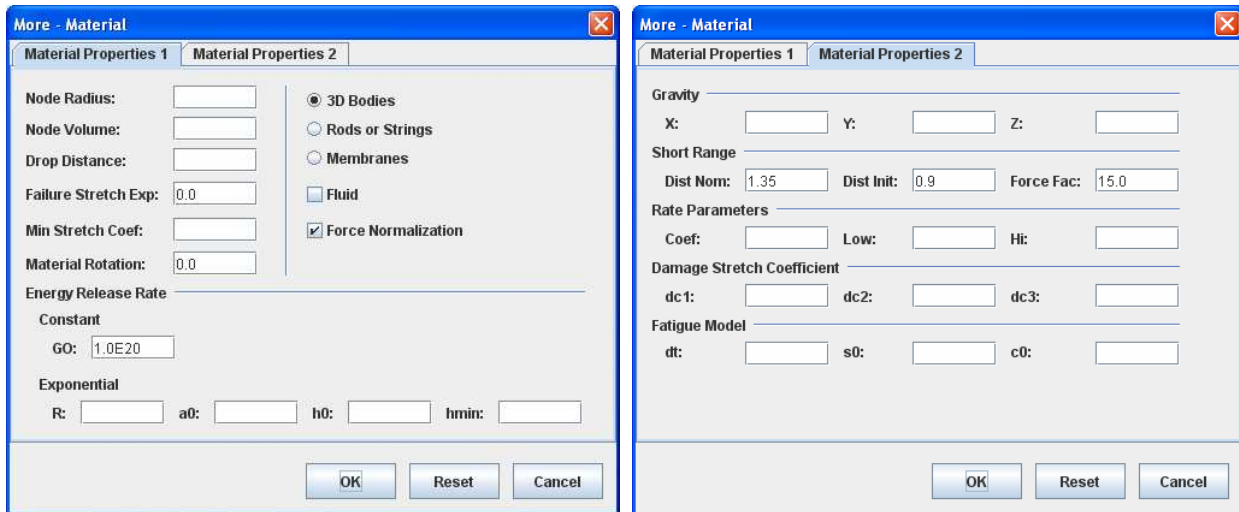


Figure 4.10 More - Material dialog showing both tabs

The Initial Conditions button opens the dialog shown in Figure 4.11. The data entered here specifies the initial displacement and velocity as well as the displacement and velocity gradients for the material. The dialog contains a tabbed pane that allows the information to be split into two parts, a displacement panel and a velocity panel. The dialog with each tab selected is shown in Figure 4.11. The information entered in the More and Initial Conditions dialog is held and saved with the material when created.

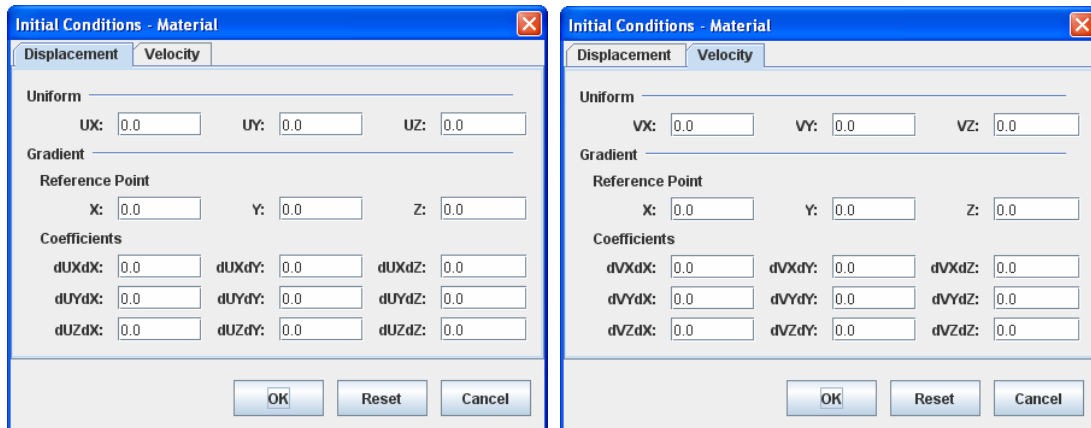



Figure 4.11 Initial Conditions – Material dialog showing both tabs

Geometry

The geometry panel (Figure 4.12) allows the user to create all of the material regions, voids, slits, precracks, and enter grid files. Each of the different geometries is represented by a toggle button with a picture. Toggle buttons restrict selection to one button at a time. When a

button is selected the previous button is deselected keeping the selection to one at a time. Hovering the mouse over the toggle button displays information about the button. Selecting the button changes the area below to enter the data necessary to create the chosen shape. The first button, the rectangle button (), displays the panel as shown in Figure 4.12. There are six parameters needed to define the rectangle geometry and three different types of rectangle: internal, external, and void. If the shape is a material region (internal and external types for a rectangle) then a material must be chosen from the drop down menu above the Apply button. If the shape is not a material region (void type for a rectangle), there is no need to specify a material and the drop down menu is disabled. After entering the appropriate data, the shape is created by clicking the Apply button.

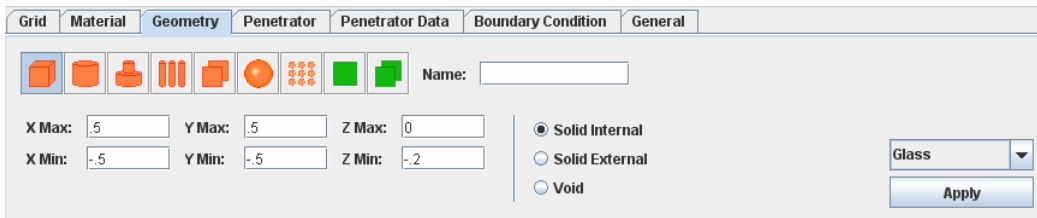


Figure 4.12 Tabbed pane showing geometry

Penetrator

The penetrator panel (Figure 4.13) controls the creation of the penetrator shape. As mentioned previously, there can only be one penetrator and there are five different types to choose from. The user can select each option and see a picture of the desired type by pressing the radio buttons on the left of Figure 4.13. The data needed to define the chosen type is displayed to the right of the picture. After entering the necessary information, the penetrator is created by clicking the Apply button. If a penetrator has already been created, pressing the Apply button replaces the current penetrator with a new one of the chosen type. The delete button is enabled when a penetrator exists and removes the penetrator.

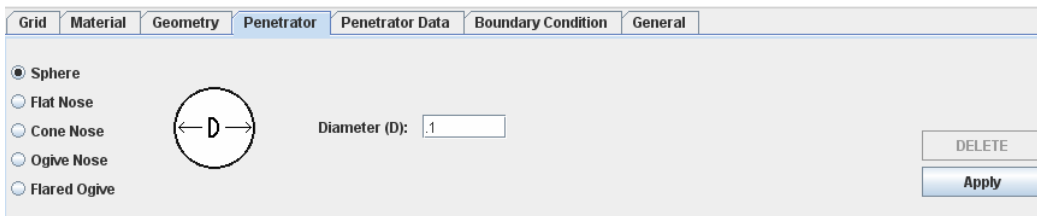


Figure 4.13 Tabbed pane showing penetrator

Penetrator Data

The penetrator data panel (Figure 4.14) allows the user to enter information about the penetrator. After entering the desired data, the Apply button saves the information with the current penetrator. If there is no penetrator, the Apply button is disabled.

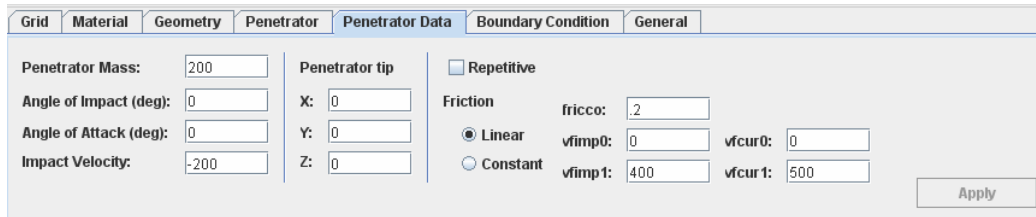


Figure 4.14 Tabbed pane showing penetrator data

Boundary Condition

The boundary condition panel (Figure 4.15) allows the user to create three different types of boundary conditions. The types are “no fail”, displacement, and velocity. As with the geometry panel, the rectangle and cylinder geometries are represented by a toggle button with a picture. Clicking the toggle button displays the data fields necessary to define the geometry. The user can also choose interior or exterior types of boundary conditions. Clicking the Apply button creates the specified boundary condition.

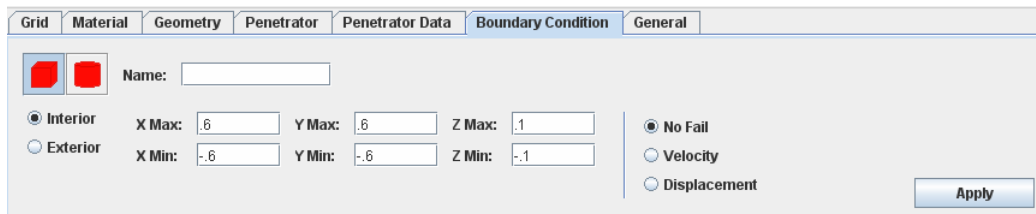


Figure 4.15 Tabbed pane showing boundary condition

When choosing between “no fail”, velocity, and displacement, a dialog appears for velocity and displacement that allows the user to enter the data relevant to those types. The two dialogs, shown in Figure 4.16 and Figure 4.17, are similar and allow the user to specify conditions on any axis. To specify a condition, the user must select the appropriate check box and enter a value in the corresponding field to the right. Clearing the check box leaves the direction unconstrained in the solution. The velocity dialog has an additional End Time variable used to turn off the velocity boundary condition at the specified time. The “no fail” boundary

condition needs no additional information and keeps all peridynamic bonds within the bounds from breaking.

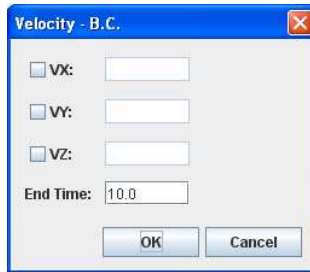


Figure 4.16 Velocity boundary condition dialog



Figure 4.17 Displacement boundary condition dialog

General

The general panel allows the user to enter all of the remaining data. The most common fields are located on the panel and many additional fields are located in a dialog accessed by clicking the More button. After entering the desired information in the panel and the dialog, clicking the Apply button saves the information.

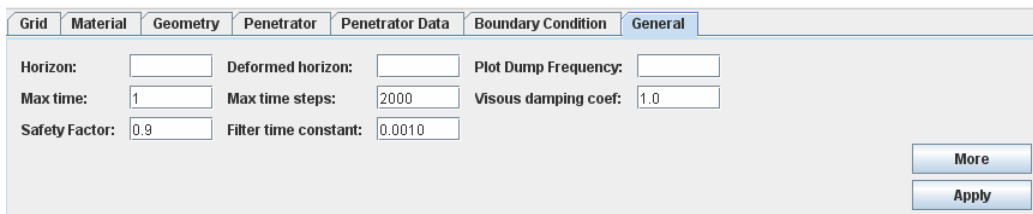


Figure 4.18 Tabbed Pane showing general

Clicking on the More button opens another dialog with a tabbed pane. The tabbed pane has four options: Restart, Output, Interface, and Misc. EMU allows the user to restart a previous EMU solution through files that are saved by EMU during a solution. The user can specify these options in the Restart tab shown in Figure 4.19. In addition to restarting an EMU solution, the

user can tell EMU to create a restart file for later use. This information is also available on the Restart tab.

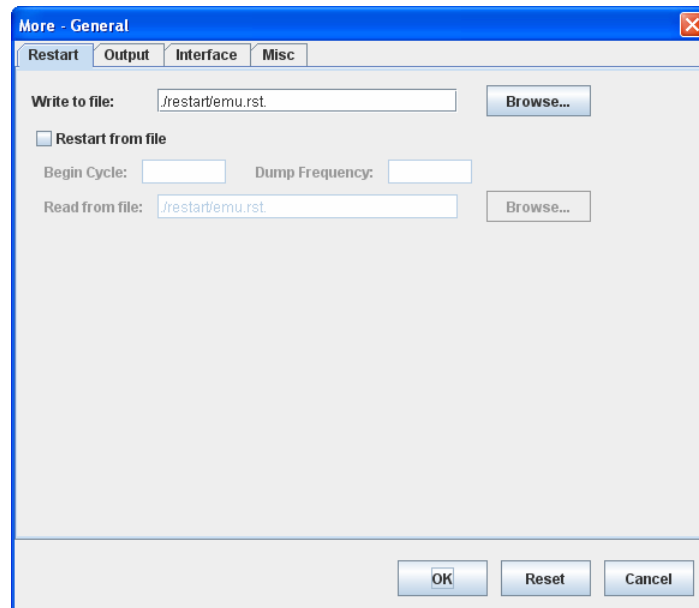


Figure 4.19 More – General dialog showing the Restart tab

The Output tab holds information that tells EMU what to display during a solution and when and where to place information for post-processing. The most common output options are located on the General panel, but the rest are located in the dialog on the Output tab shown in Figure 4.20. While an EMU solution runs, the user can see a single line output at each time step that provides information about the peridynamic grid or penetrator. The user can select their preference in the Output tab. The user can also specify the location of the plot files that will be used by EMUGR, the EMU post-processor.

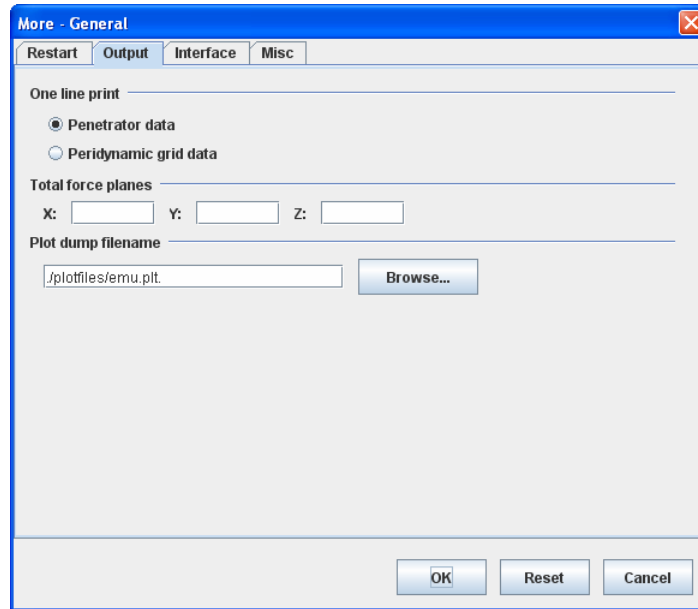


Figure 4.20 More – General dialog showing the Output tab

The Interface tab allows the user to connect and disconnect material regions. The connection refers to the peridynamic bonds between nodes mentioned previously. These bonds should not exist in some situations and need to be removed. By default, there are peridynamic bonds between all nodes within the material horizon distance mentioned previously. An exception is between rebar mesh and any other material region. By default, two non rebar mesh material regions adjoining each other act as one material region with two different material properties. To keep the two material regions separate, the Interface tab allows the user to specify which material regions are connected and disconnected. The user can also specify the interface force and strength coefficients between two material regions. To change the default settings between two material regions, the user selects two material regions from the list shown in Figure 4.21 and enters the desired information. To select more than one material region, hold down the control button and click on the second material region. Once two material regions are selected, the options below the list are enabled. More detail is given with the InterShape class in the description of the VE package.

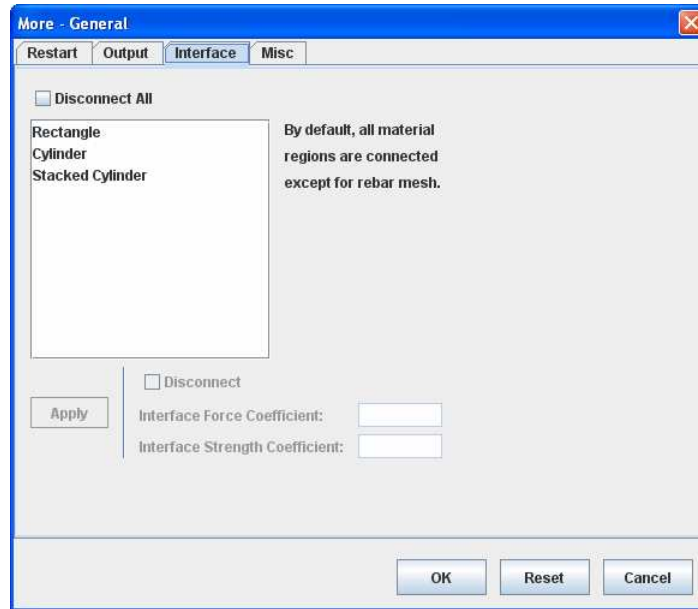


Figure 4.21 More – General dialog showing the Interface tab

The Miscellaneous tab allows the user to enter the remaining information that has no other place in Visual EMU. As shown in Figure 4.22, the options include “no fail” perimeter, damage viscosity, number of processors, fatigue loading, fixed time steps, and node history locations.

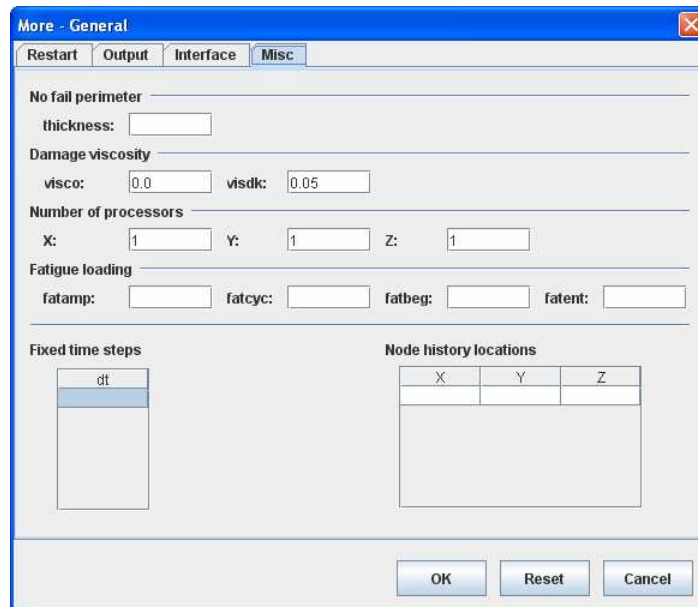



Figure 4.22 More – General dialog showing the Misc tab

Code design

Using object oriented design, all of the source code for Visual EMU is divided into nine files and packages. By definition, a package is a collection of related classes (Horstmann, 2006). These files and packages help organize the classes into the similar types mentioned in the discussion of object oriented programming and also arrange the code for easy navigation by the developer. The following section explains the purpose and functionality of each file and package.

Action

Each action class represents a specific task completed at the request of the user. By extending the Java Abstract Action class, each action can implement (gain access through the inheritance mentioned in the discussion of Java) a method called *actionPerformed()* (Java Platform Standard Ed. 6, 2006). Inside the *actionPerformed()* method of each action class are the instructions to complete the task. Though each method is different for all fifteen actions, each class is handled in the same way through the inheritance from the Abstract Action class. Each action is assigned to a button, buttons, or menu item and the *actionPerformed()* method is called immediately after selection of the item it was assigned to. As an example, the *actionPerformed()* method in *NewAction* is called from the new button () on the toolbar in section 2 of Figure 4.2 and an option on the File submenu shown in Figure 4.3. The same action is called from both places.

The UML diagram for the *NewAction* class is shown in Figure 4.23. The class needs access to the current information held in *ShapeData* and *VisualEMUWindow*. The variables that allow the access are *d_data* and *d_frame* respectively. More explanation on these two classes is provided in the *ShapeData* and *VisualEMUWindow* sections of the VE package. The *actionPerformed()* method confirms that all current data will be lost with this action before clearing all of the current data and resetting to the default information.

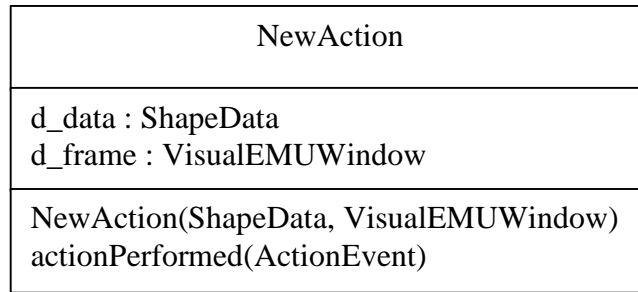


Figure 4.23 NewAction UML

The following list gives a brief explanation of each class in the action package:

- AboutAction: Opens a dialog that displays information about the author and version of Visual EMU.
- AddBCAction: Adds a new boundary condition shape.
- AddShapeAction: Adds a new shape that is not a boundary condition. This includes the penetrator, material regions, slits, precracks, and voids.
- ExitAction: Closes the Visual EMU program.
- NewAction: Clears all Visual EMU data and resets to default information.
- OrbitStateAction: Sets the 3D view to orbit state. The view can be rotated, translated, and zoomed with the use of the mouse.
- ReadInFileAction: Opens and reads the information from an EMU infile.
- ReadStateAction: Opens a saved Visual EMU file.
- RedoAction: Executes the task at the top of the redo list.
- ResetAction: Resets the 3D view to default orientation.
- RunEMUAction: Opens a dialog that gives the options for running EMU. After the options are successfully entered, the action completes the necessary setup and executes EMU.
- SaveAction: Saves the current Visual EMU settings to a “*.vem” file.
- SelectionStateAction: Will enable selection of objects in the view panel in the future.
- UndoAction: Execute the task at the top of the undo list.
- WriteInFileAction: Writes an EMU infile from the current Visual EMU settings.

Command

Each command class represents a task that can be undone and redone. When a new command is created, the task is executed and the command is added to the undo list held in the CommandManager class. The list allows the user to undo, or reverse, each task in the opposite order they were executed. The commands are kept in order, as shown by the numbers one to three in Figure 4.24 (a). The first command is labeled one and is at the bottom of the list while the last command is labeled three and is at the top of the list. The last command added to the list is always the first to be removed and its execution undone. When the undo action is selected, the task of the last command executed is reversed and the command is moved to the redo list (Figure 4.24 (b)). The undo action can be used for each command in the undo list. With a command in the redo list, the redo action is available. If selected, the task is executed and the command placed back on the undo list. The result is a return to the state of Figure 4.24 (a).

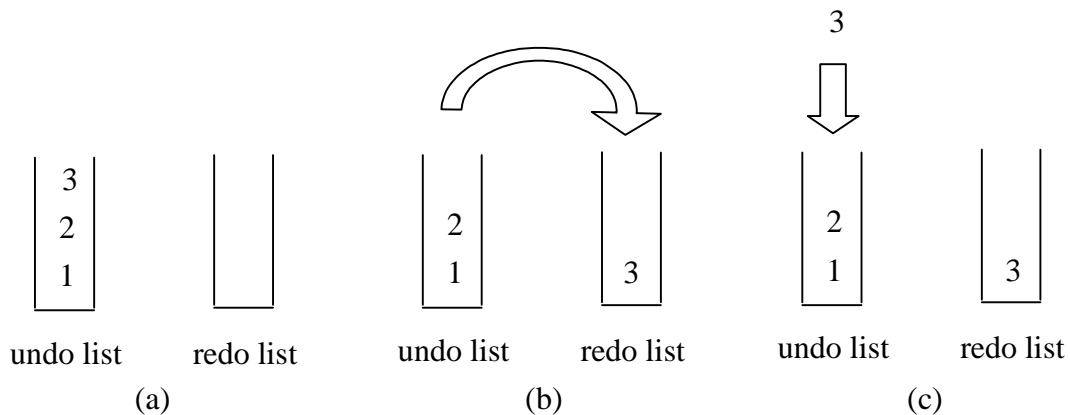


Figure 4.24 The undo and redo lists with three commands (a), undo action (b), and new command (c)

Commands can go back and forth from the undo list to the redo list an infinite number of times. This can continue until a new command is added to the undo list, one not from the redo list, as shown in Figure 4.24 (c). When the new command is added to the undo list all commands on the redo list (command three in the redo list in Figure 4.24 (c)) are deleted. The new command (the three above the undo list) is placed in order in the undo list. In this way, the order of commands remains constant. The new command three is the first undone and the last redone. If the commands on the redo list were not removed, it could be possible to have a shape on the

redo list with the same name, a unique identifier, as a shape added to the undo list. Allowing the command to be redone would bring back the shape with a duplicate name. Many potential problems exist in the logic of the code if the name field is not unique.

The `CommandManager` class is implemented as a singleton, with the UML diagram shown in Figure 4.25. A singleton ensures that only one instance of the class is created and provides a global point of access (Geary, 2003). The `CommandManager()` constructor is therefore a private method. This means no class other than the `CommandManager` can create the undo and redo lists. All classes can call the public static method `getCommandManager()` though. The method checks to see if an instance of `CommandManager` exists and returns the existing `CommandManager` or makes a new one. In this way only one set of lists are ever created.

The `execute(Command)` method adds the new command to the undo list and clears the redo list for the reasons previously described. The `undo()` method moves the command most recently added to the undo list over to the redo list. The `redo()` method moves the command most recently added to the redo list to the undo list. The `undoValid()` method returns a boolean value that is true if there are any commands in the undo list and the `redoValid()` method does the same for the redo list. These methods help determine if the undo and redo actions can be used. The `clearLists()` method removes all commands from the undo and redo list.

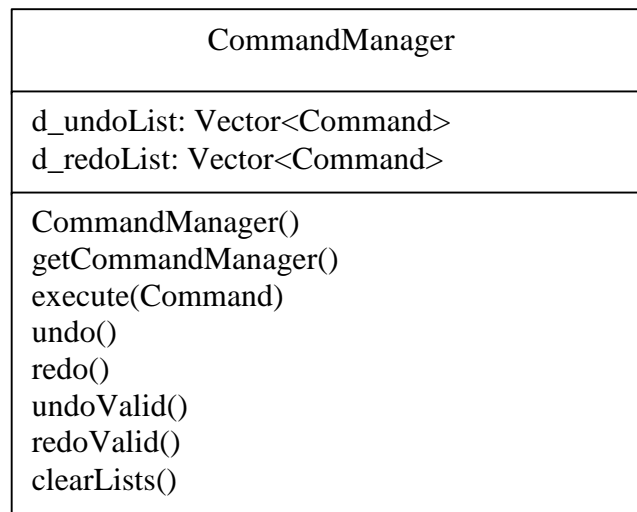


Figure 4.25 CommandManager UML

The following is a brief description of each class in the command package:

- `AddBoundaryRegionCommand`: Holds the boundary condition shape addition task.

- AddCylinderCommand: Holds the cylinder shape addition task.
- AddCylinderExteriorCommand: Holds the cylinder shape of type exterior addition task.
- AddCylinderVoidCommand: Holds the cylinder shape of type void addition task.
- AddLayerCommand: Holds the layer shape addition task.
- AddMaterialCommand: Holds the material data addition task.
- AddPenetratorCommand: Holds the penetrator shape addition task.
- AddPrecrackCommand: Holds the precrack shape addition task.
- AddRebarMeshCommand: Holds the rebar mesh shape addition task.
- AddRectangleCommand: Holds the rectangle shape addition task.
- AddRectangleExteriorCommand: Holds the rectangle shape of type exterior addition task.
- AddRectangleVoidCommand: Holds the rectangle shape of type void addition task.
- AddSlitCommand: Holds the slit shape addition task.
- AddSphereCommand: Holds the sphere shape addition task.
- AddStackedCylinderCommand: Holds the stacked cylinder shape addition task.
- Command: The interface that requires each command to inherit the methods *undo()*, *redo()*, and *execute(Command)*.
- CommandManager: Contains the undo and redo lists and controls the movement of commands between the two.
- DeleteMaterialCommand: Holds the task that removes a material data.
- DeleteShapeCommand: Holds the task that removes a shape.

Dialog

Each dialog class represents a unique window used to show or gather information. The dialog is a stand alone window that appears in front of the Visual EMU program. When a dialog appears, it is impossible to interact with the window behind until the dialog is closed. This is called a modal dialog (Morelli and Walde, 2006). The control is imposed on Visual EMU only though, and the user can switch to a different program. While some dialogs only display information, others act as portals for the passing of data. This is most common with the More button on many of the tabbed panels. Each More button opens a modal dialog with additional

options and allows the user to enter information that could not fit on the main panel. The following is a brief description of each class in the dialog package:

- **BoundaryConditionDispDialog:** Collects displacement information for the boundary condition.
- **BoundaryConditionVelDialog:** Collects velocity information for the boundary condition.
- **FinalAdvancedDialog:** Collects additional information for the general panel.
- **GridAdvancedDialog:** Collects additional information for the grid panel.
- **MaterialAdvancedDialog:** Collects additional information for the material panel.
- **MaterialDefaultDialog:** Opens a dialog that contains all of the default keywords that apply to materials. This data is used by EMU when not specified for a material.
- **MaterialInitialDialog:** Opens a dialog that collects information on the initial condition of the material.
- **MyDialog:** The interface for each dialog. It requires each dialog to contain the *doModal()* and *validData()* methods that help produce the modal nature of the dialogs.
- **RunEMUDialog:** Opens a dialog that allows the user to choose locations and options before running EMU.

DisplayHelper

DisplayHelper is not a group of related classes. For this reason it is not a package but a folder, a holder for the organization of files. The classes held in the DisplayHelper folder aid the classes in the panel and dialog packages. Each of these classes is used in multiple locations to speed up development and create dialogs and panels with consistent content. The following is a brief description of each class in the DisplayHelper folder:

- **DialogHelper:** This helper class aids the creation of different objects used in the dialogs. An example is the text label with a horizontal line extending to the right such as no fail perimeter shown in Figure 4.22.
- **DoubleField:** This class controls each text field used for numerical entry.
- **DoubleHighBoundException:** A special addition to the DoubleField that helps control data entry. A maximum value is set and any number above is refused. The evaluation of the value and the warning message, if required, are held here.

- **DoubleLowBoundException:** A special addition to the DoubleField that helps control data entry. A minimum value is set and any number below is refused. The evaluation of the value and the warning message, if required, are held here.
- **FormatDouble:** When a number is turned into a string to display or write to a file, the format can be controlled through this class.
- **GridBagHelper:** This helper class simplifies the code by allowing an easy way to add objects to the dialogs and panels. This helper class is a product of Thunderhead Engineering and has been licensed for use in Visual EMU.

Images

All of the images needed for Visual EMU are loaded into a central holding class called ImageHolder. The pictures are then retrieved by any class without the need to find or reload the image each time it should be displayed. This is most advantageous for the tree shown in section 4 of Figure 4.2. Each object in the tree has a representative picture and the number of possible items in the tree is virtually unlimited. The speed of Visual EMU increases by having each picture ready when needed.

Panel

The panel package holds all of the classes that control each of the panels in section 3 of Figure 4.2 and were explained in detail in the Tabbed Pane section. These classes not only display the options to the user but in most cases they save the desired information. The GeometryPanel class, however, is used only to pass information between the ShapeData class and the user and no information is stored in the class. The ShapeData class is explained in more detail in the VE package.

Shapes

A shape represents any geometry that needs to be drawn in the view panel. Having these geometries extend Shape allows them to be held together with one variable. The variable only knows it holds a class of type Shape and any subclass of Shape counts. Each shape can then be retrieved from the variable and used in the same way without differentiating the type. The shapes are further broken down into material regions and non material regions. To easily

distinguish between the two, all shapes inherit a method named *isRealMaterialRegion()*. The method returns true for material regions and false for everything else.

Material regions apply material properties to the nodes within their bounds. Any nodes without a material region are dropped from EMU before the solution begins. Visual EMU allows the user to see the EMU node configuration before running a solution. As mentioned previously, this saves time that may have been wasted on an incorrect solution. To display the correct nodes, each material region has a list that holds Point3d objects (*d_gridPoints*). Each Point3d on the list represents a node the shape applies material properties to and can be shown to the user. When a material region is created, Visual EMU checks the peridynamic grid and saves the nodes within the material region. For efficiency, the code steps through each axial direction until it reaches the bound of the shape and stops checking a direction when the other bound is reached. The nodes that are saved can then be drawn and redrawn without recalculating. Each material region is drawn quickly which allows the view panel to rotate, translate, and zoom more smoothly. All nodes only need to be recalculated when a change is made to the grid or a void region is added or deleted.

Shapes that are not material regions do not apply a material to the nodes in the peridynamic grid. These shapes do not create nodes but may have an effect on the nodes created by material regions. The shapes that are not material regions are the penetrator, grid boundary, slit, precrack, boundary condition, and void. These shapes share methods associated with material regions and therefore fall into the same type. The non material regions shapes are explained in more detail in the following paragraphs.

The penetrator, as mentioned previously, is not defined by nodes and always retains the same geometry regardless of node size and spacing. It is generally used to impact material regions and has five different types as seen in Figure 4.26: sphere (a), flat nose (b), cone nose (c), ogive nose (d), and flared ogive (e). All five penetrator types need a diameter along with a variety of parameters to define the nose and tail. Not shown in the figures is how to define the curve of the ogive nose for types d and e. The radius of curvature for the ogive nose is found by multiplying the diameter (D on each figure) with a variable named *crh* that is required from the user.

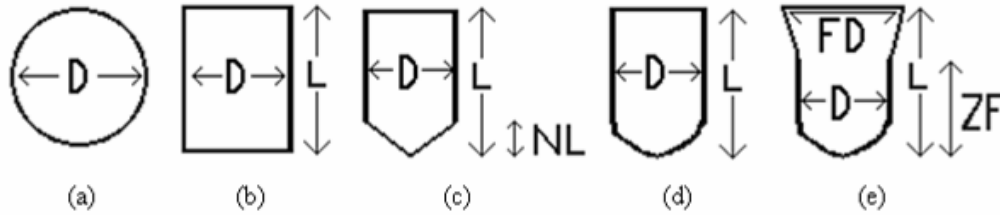


Figure 4.26 Penetrator types

The grid boundary is an aid for the user and shows the region in space that contains the internally generated nodes. The grid boundary quickly shows the user if any material region is outside the grid boundary and will have missing nodes in the EMU solution. Nothing stops the user from creating a material region outside of the grid boundary in EMU or Visual EMU, but a material region has no effect on the solution without nodes. The grid boundary, represented by the thick black lines in Figure 4.27, can be turned on and off from the grid panel (Figure 4.6).

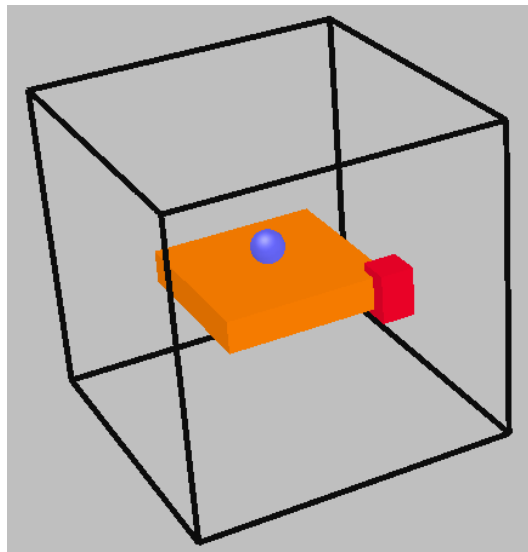


Figure 4.27 View panel showing the grid boundary (black)

The slit defines a plane that cuts peridynamic bonds. This is similar to the disconnect keyword for two material regions that was mentioned previously. There is no peridynamic interaction across this plane and the nodes on either side act as separate objects sitting beside each other. As shown in Figure 4.28, the slit breaks the rectangle material region into two separate blocks. After the bonds are broken the material region acts like two blocks adjacent to each other.

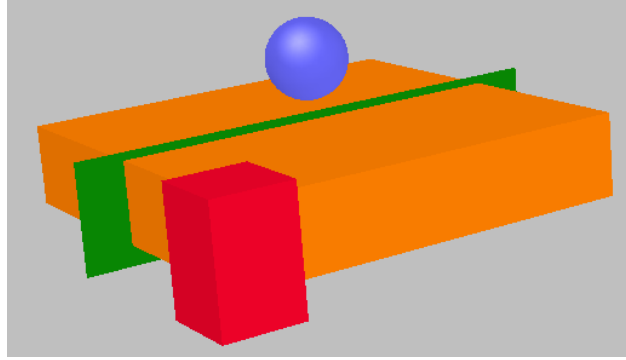


Figure 4.28 View panel with a slit plane (green)

The precrack is similar to the slit but has different options. While the slit must be parallel to the x, y, or z plane, the precrack can have any orientation. The precrack also has a thickness and therefore a volume. Any peridynamic bonds that touch the volume of the precrack are broken. As shown in Figure 4.29, the precrack is drawn as two planes separated by the given thickness. Any peridynamic bonds in the volume are broken.

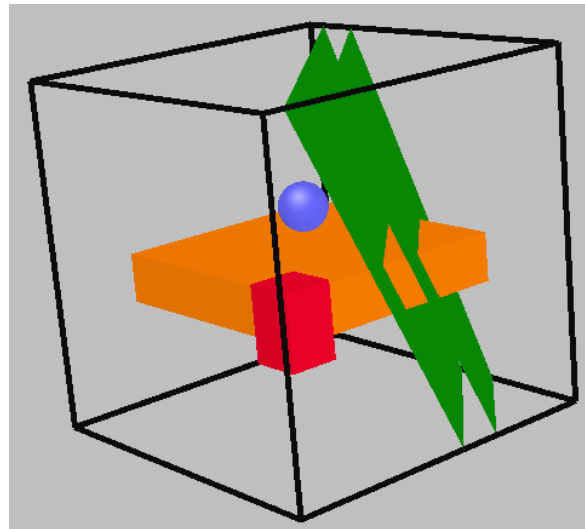


Figure 4.29 View panel with a precrack (green)

The boundary condition adds a displacement or velocity gradient to the nodes contained within its bounds. The gradients only apply to nodes that are part of a material region but can contain any number of material regions. The red block in the middle of Figure 4.29 is a boundary condition. The boundary conditions are drawn in the same way as material regions in the solid frame and wire frame viewing options. They are then drawn as a wire frame when viewing nodes so the user can see which nodes are affected by the boundary condition.

The void removes all nodes within its bounds from calculation. All other information applied to the nodes is irrelevant. The void shapes are colored yellow to distinguish them from other shapes. Void shapes are also drawn the same as material regions in the solid frame and wire frame viewing options and do not remove anything from other shapes, a useful feature for future addition. When viewing nodes however, the void regions are drawn as expected in EMU. All nodes within the bounds of the void are not shown. This is consistent with EMU and represents the node configuration of an EMU solution. A cylinder void is shown in all three view options in Figure 4.30.

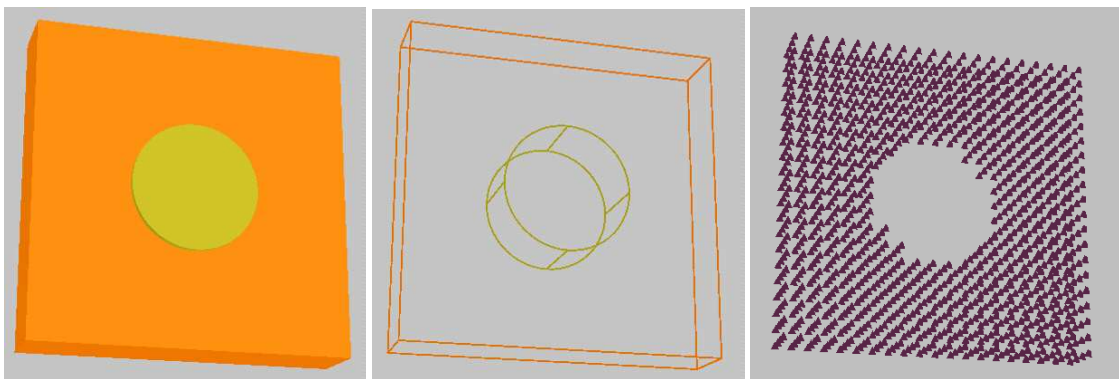


Figure 4.30 Cylinder void shown in solid (left), wire (middle), and grid (right) frame views

The UML for the Shape class is shown in Figure 4.31 and the following sections give a brief explanation of some of the important methods.

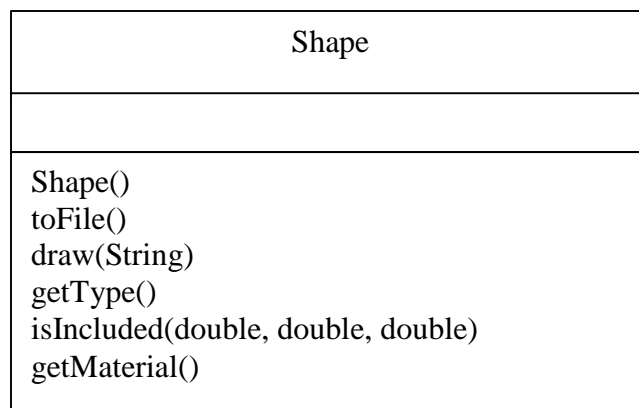


Figure 4.31 Partial Shape UML

toFile()

Each shape class contains the data entered by the user that needs output to the infile for EMU. The *toFile()* method returns a string formatted according to the type of shape. For example, the *toFile()* method of the cylinder class is shown in Figure 4.32. The three indicates what type of material region is being defined and each of the following parameters define the geometry and must follow the order required by EMU.

```
public String toFile()  
{  
    return " 3 "+d_radius+" "+d_xCen+ " "+d_yCen+" "+d_zL+" "+d_zH;  
}
```

Figure 4.32 *toFile()* method from the Cylinder class

draw(String)

The *draw(String)* method is called each time the view panel changes. Each call to the *draw(String)* method is passed a string that signifies what type of draw is taking place. The three options are solid, wire, and grid frame. The solid view is drawn with solid triangles for each shape. The triangles are large for a flat surface and small when used to represent curved surfaces such as cylinders and spheres. The wire view only draws the outline of the shape making it is possible to see shapes inside other shapes and where shapes overlap. The grid view shows the nodes that will appear in EMU with the given conditions. Each type of shape responds differently depending on the draw option given. This information is held in the *draw(String)* method. For example, a sphere has a solid, wire, and grid structure while all voids, which represent the absence of nodes, draw nothing in grid view. Though boundary conditions have no nodes, a wire frame is drawn in grid view to help the user determine which nodes are affected.

getType()

Each shape class is unique but can fall into different categories such as: material region, void, slit, precrack, or boundary condition. This information is held by a string variable in each class and cannot be changed by the user. The *getType()* method returns the string as a way of identifying and categorizing the shape. For example, a rectangle void region has the type “Rectangle Void” and a cylinder void region has the type “Cylinder Void.” The string returned

from *getType()* can then be tested to see if “Void” is included. Even though these two shapes are different, they are both voids and are identified and used appropriately.

isIncluded(double, double, double)

Each material region defines a boundary that applies material properties to the nodes within. The nodes are set by the grid and each node is tested to see if it is included in the material region or void. If the node is within a void region, it is ignored to keep it from being drawn. If the node is not within a void region but within a material region, the node is added to the list mentioned previously to be drawn. In this way, each shape is tested uniquely with the same inherited method.

getMaterial()

To keep material information with the appropriate material region, a string variable holds the unique name of the material. This method is called when displaying the material region properties or when writing the information to the infile. Some shapes that are not material regions use the *getMaterial()* method also, such as the precrack. A precrack can apply to only one material region so that only peridynamic bonds within the specified material region are broken.

State

The state package contains all of the classes that control how the user can interact with the view panel. At the moment, the two options are selection and orbit state. The orbit state allows the user to rotate, zoom in or out, and translate the camera. The camera is explained in more detail in the Camera class of the VE package. By clicking in the view panel and moving the mouse while holding down the left mouse button, the shapes appear to rotate in the direction of mouse movement. By clicking in the view panel and moving the mouse while holding down the right mouse button, the shapes appear to translate in the direction of mouse movement. By rolling the mouse wheel in the view panel, the shapes appear to move closer or farther away. The selection state is designed to hold the camera in one orientation and allow the user to select shapes in the view panel with the mouse. The ability to select is not yet included in Visual EMU and all shape selection is handled through the tree. Allowing the user to select and manipulate shapes in the view panel is a useful feature for future addition.

Each class in the state package that controls user interaction extends the state class. The UML for the state class is shown in Figure 4.33 and uses mostly abstract methods. An abstract method has no body and requires all classes that extend it to implement the method (Morelli and Walde, 2006). This means there is no method definition in the state class and each class that extends state must define the method. For example, the *mouseDragged(MouseEvent)* method is called when the user moves the mouse with a button held down. In the selection state, this method does nothing. In the orbit state, however, this rotates the camera if the left button is held and translates the camera if the right button is held. The only method in the state class that is not abstract is the *stateHasChanged()* method which is used to update the view panel when changes are made.

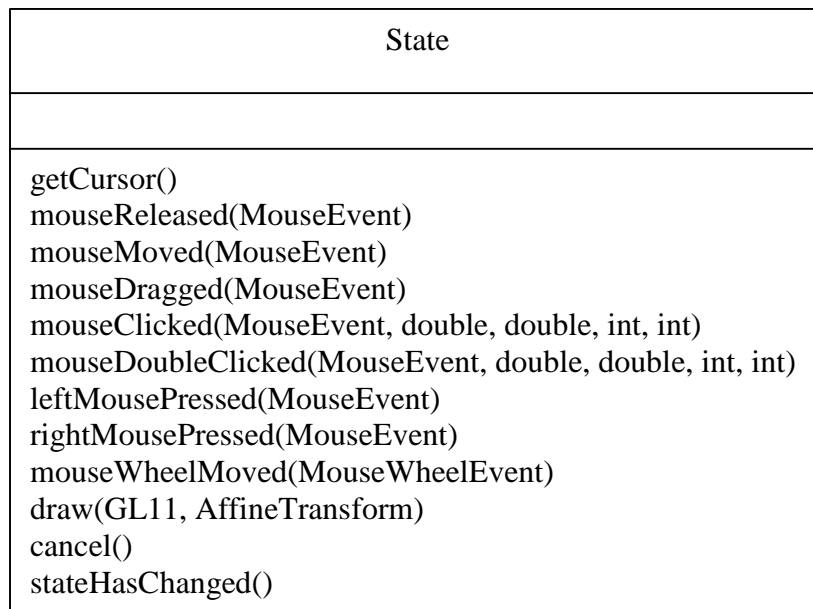


Figure 4.33 State UML

VE

VE is another group of classes that are not also a group of related classes. Though the classes in this folder are not alike, they are the foundation for Visual EMU. Included in this package are the main class where Visual EMU begins, the class that manages all visual components, and the class that manages all shapes and materials. This section gives an explanation of the most important classes.

Camera

The view panel shows the shapes as if viewed through a camera lens. Though it may look like the shapes are rotating, the location of the shapes never changes. The camera can be moved up and down, zoomed in and out, and rotated around a center point. The change in view is created by proper mouse movement. Each mouse command that changes the view of the camera is divided into rotation, translation, or zoom. These changes are made to the current view through the creation of a temporary transformation matrix and then applied to the overall transformation matrix. As you can see in Figure 4.34, a temporary transformation matrix, *mRX*, is created with the rotation information that comes from the mouse, *thetaX*, and multiplied with the current transformation matrix, *d_mTransform*, to perform a rotation about the x axis. The *d_mTransform* matrix is then used to draw the shapes as if viewed from the desired location.

```
public void rotateX(double thetaX)
{
    Matrix4d mRX;
    double cosTheta = Math.cos(thetaX/180.*Math.PI);
    double sinTheta = Math.sin(thetaX/180.*Math.PI);

    mRX = new Matrix4d( 1., 0., 0., 0.,
                       0., cosTheta, -sinTheta, 0.,
                       0., sinTheta,  cosTheta, 0.,
                       0., 0., 0., 1.);
    d_mTransform.mul(mRX, d_mTransform);
}
```

Figure 4.34 *rotateX(double)* method from the camera class

The methods *rotateX(double)*, *rotateY(double)*, and *rotateZ(double)* control the rotation in each axial direction and are shown along with all other Camera methods in the UML diagram in Figure 4.35. Other notable methods are *translate(double, double, double,)*, which controls translation and *setDistance(double)*, which controls the zoom.

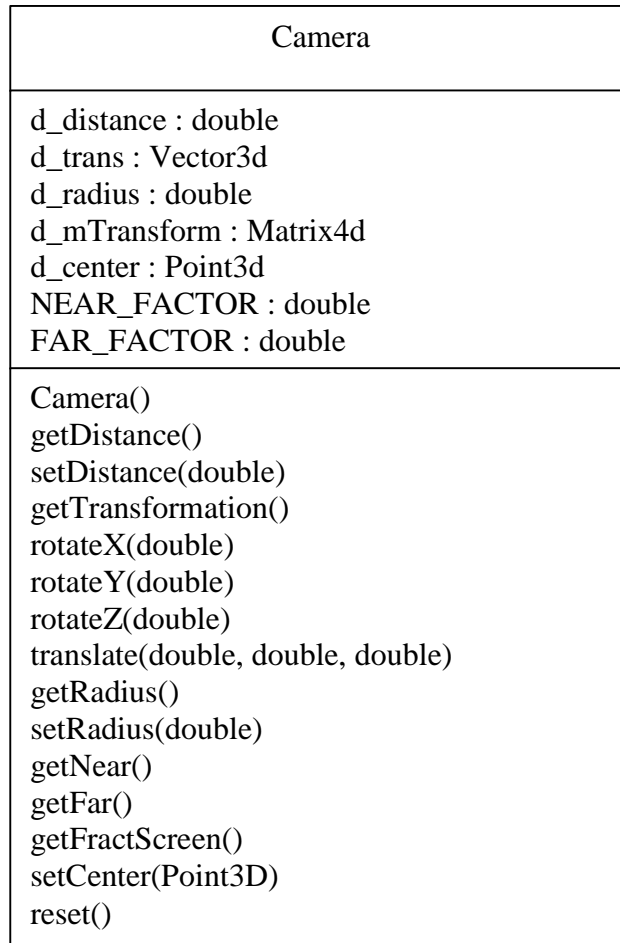


Figure 4.35 Camera UML

Initialize

To aid the user, three materials are predefined in Visual EMU. They represent the properties of glass and two types of metal. As mentioned previously, the creation of material regions in Visual EMU requires a material to associate with the region. The additional materials are useful to any user wishing to use Visual EMU for geometry and grid generation or to simply visualize the problem. The material assigned to the material region can be edited or replaced at any time. An experienced EMU user can use Visual EMU to quickly create and visualize geometry and then edit the infile. The Initialize class creates the additional materials. Any materials not used with a material region have no affect on the EMU infile. The additional materials are available for use without being a hindrance.

InterShape

As mentioned previously, there are certain interactions that can be specified between material regions. The control for this behavior is held in the `InterShape` class, the UML of which is shown in Figure 4.36. The `connect` and `disconnect` keywords mentioned previously are controlled using a simple boolean value, identified as `d_isOppositeConnection`. All material regions are connected except rebar meshes. This default behavior changes with the addition of one keyword, `disconnect_all`, which disconnects all material regions. Regardless of the default setting, when the boolean is set to true the two material regions have the opposite behavior of the default at that time. This allows the same class to control all material regions including rebar mesh. If a rebar mesh and any other shape have `d_isOppositeConnection` set to true, they are connected, which is the opposite of default. If any two non rebar mesh materials are set to true, they are disconnected, which is opposite the default. If the `disconnect_all` keyword is used, the default is the same for all material regions and a true `d_isOppositeConnection` for any two shapes connects them.

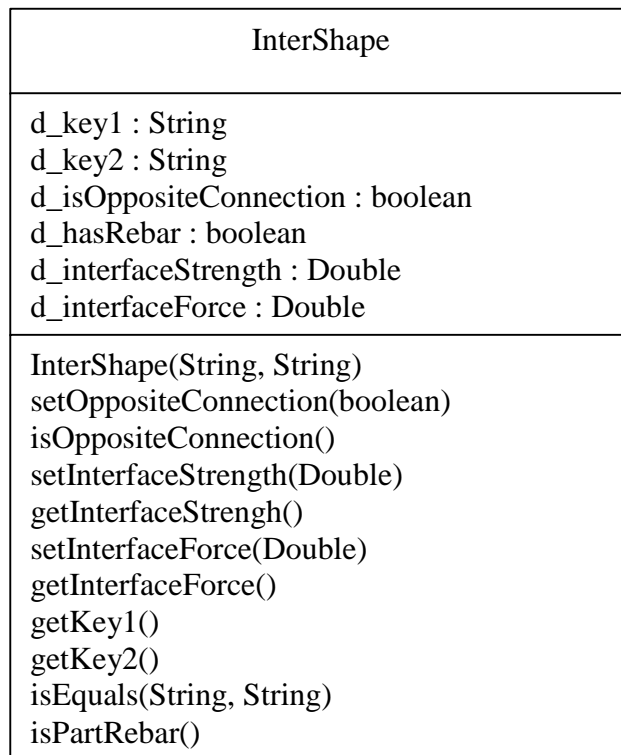


Figure 4.36 InterShape UML

The get and set methods for interface strength and interface force provide additional information for the interaction of two material regions. The material regions that the InterShape instance applies to are saved as *d_key1* and *d_key2*. These string values hold the names of each material region. The *isEqual(String, String)* method compares two material region names with the two saved names, as shown in Figure 4.37. If the strings are the same, there is already an InterShape instance for the pair of material regions and there should only be one instance for each pair of material regions. The *isPartRebar()* method is used to identify if one of the two material regions is a rebar mesh. The instance is then handled differently since rebar mesh has a different default.

```
public boolean isEqual(String key1, String key2)
{
    return((key1.equals(d_key1) && key2.equals(d_key2)) ||
           (key1.equals(d_key2) && key2.equals(d_key1)));
}
```

Figure 4.37 *isEqual(String, String)* method from the InterShape class

MaterialData

The MaterialData class has a public constructor and when a new material is created, a new MaterialData instance is created with the values given by the user. All the material properties are held in the instance of MaterialData. Any values that are not set by the user are not written to the infile and left to be the EMU default. The instance can be linked, by the unique name, to any material region. The MaterialData class has 55 variables and 85 methods. The majority of the variables hold material parameters and the methods manage those parameters. A partial UML of the MaterialData class is shown in Figure 4.38 to give an example of how the class works.

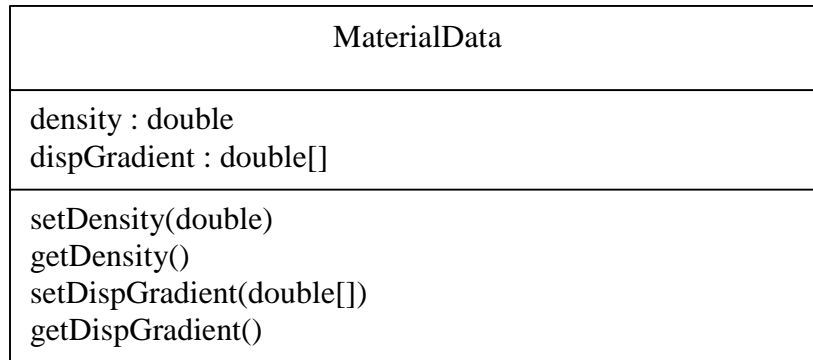


Figure 4.38 Partial MaterialData UML

The material density is stored as a double called *density*. The value is set by the *setDensity(double)* method and returned by the *getDensity()* method. The displacement gradient variable is a bit more complicated and is stored as a double array called *dispGradient*. The 12 parameters associated with the displacement gradient are shown in Figure 4.11. The first three are for the reference point and the remainder for the coefficients. All 12 parameters are set with the *setDispGradient(double[])* method and returned with the *getDispGradient()* method.

When outputting the material information for EMU, only the non default values are written. This consolidates the infile and allows the user to quickly see the changes from EMU default. For this reason, entering the EMU default information into Visual EMU has no effect on the infile Visual EMU creates. If the user reads an infile into Visual EMU and then writes an infile, making no changes, the files may be different. Any default information in the original infile does not appear in the new infile.

ReadInFile

The ReadInFile class controls the input of EMU infiles. The two methods in the ReadInFile class are shown in the UML diagram in Figure 4.39. The constructor method, *ReadInFile(ShapeData, VisualEMUWindow)*, saves the ShapeData and VisualEMUWindow instances for use in the *readData(String)* method. The *readData(String)* method takes the infile location as a string and opens the correct file. The method then takes two passes through the infile. The first pass reads in the majority of the keywords and initiates all of the necessary setup for keywords to come. After initiation, all of the remaining keywords are read. For example, material properties are read on the second pass and applied to the correct materials identified on the first pass. Any lines not recognized are stored and displayed to the user upon completion. An

infile that is out of order can be read and then written by Visual EMU. Also, it is sometimes difficult to find spelling or other small errors in an EMU infile. By reading the infile in Visual EMU, each line with an error is shown to the user along with error messages for some specific problems.

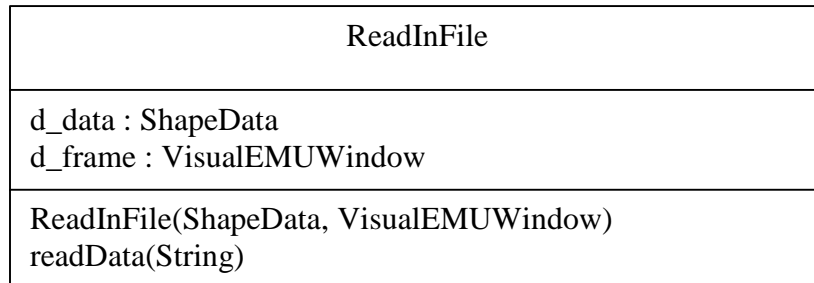


Figure 4.39 ReadInFile UML

ShapeData

The ShapeData class is the control center for Visual EMU. It holds all materials and shapes as well as the logic for their addition, removal, and organization. It holds the information for saving and opening Visual EMU files and reading and writing EMU infiles. The ShapeData class has 14 variables and 61 methods. A partial UML is shown in Figure 4.40 to give some examples of the content and function of the ShapeData class.

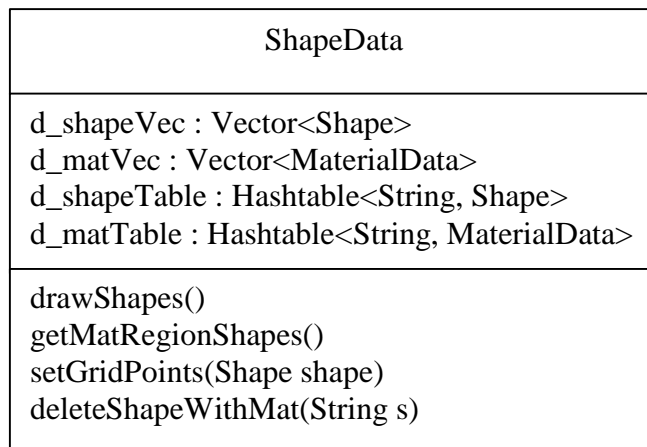


Figure 4.40 Partial ShapeData UML

As shown in the UML, there are four variables to hold the shapes and materials. Each of the variables has a different function. The vectors keep an order of creation while the hash tables allow quick access to a specific shape or material. The shape vector is used in the *drawShapes()*

method, shown in Figure 4.41, along with an inherited process of iteration. The **for** loop takes each shape from the *d_shapeVec* vector and calls the *draw(String)* method. The **if** test in the *drawShapes()* method is used to determine if the grid boundary should be drawn. The grid boundary is a shape and could be held in the *d_shapeVec* variable but is held outside the *d_shapeVec* for special treatment.

```
public void drawShapes()
{
    for(Shape shape : d_shapeVec)
    {
        shape.draw(d_geomType);
    }

    if(GridDataPanel.getGridDataPanel(null,null).isShowGrid())
    {
        GridBound.getGridBound().draw(SOLID);
    }
}
```

Figure 4.41 *drawShapes()* method in the ShapeData class

The *getMatRegionShapes()* method uses the inherited process of iteration again to collect all material region names. These names are then displayed and used in the interface panel (Figure 4.21). The *setGridPoints()* method is 180 lines long and checks voids and material regions to apply nodes. When new material regions are created, the method checks for nodes within the bounds by iterating through the grid as mentioned previously. The *deleteShapeWithMat(String)* method allows the user to delete a material being used by a material region. Material regions are required to have a material associated with them. If a material is deleted that is being used, the user is warned that all associated material regions are also deleted. If the user chooses to continue, the *deleteShapeWithMat(String)* method is called and all associated shapes are deleted.

VisualEMUView

The VisualEMUView class controls the view panel by coordinating the camera with the shapes to provide a 2D view of the 3D objects. The view panel needs to be updated when a change is made in the view panel. The VisualEMUView class also coordinates actions from the mouse when in the view panel. The information is passed to the current state, selection or orbit,

and the necessary changes are made. After making changes, the view panel is updated to show the desired results.

VisualEMUWindow

The VisualEMUWindow class controls the main window of Visual EMU. This is the first class called after the main class where Visual EMU begins. From here, most of the main classes are initialized and passed to the classes that need them. The ImageHolder class mentioned previously is initialized and all the pictures are found and loaded. Other classes that are initiated here are the ShapeData, CommandManager, StateManager, and VisualEMUView classes. Once the necessary initializations are complete, VisualEMUWindow places all five sections of the main window, shown in Figure 4.2, where they belong and then allows the user to see Visual EMU. Visual EMU is then ready to perform for the user.

CHAPTER 5 - Examples

The following sections walk through two examples using Visual EMU to verify accuracy and display the ease of use. Original infiles from Sandia National Laboratories are used as the template and compared with the results from Visual EMU. EMU results shown with EMUGR, the post-processor provided with EMU, are also included. The two examples are a spherical penetrator impacting a cylindrical glass plate and a small pipe impacting a square glass plate.

Sphere into Glass Plate

The first example evaluates a spherical penetrator impacting a cylindrical plate of glass at an angle of 45 degrees. The initial setup is shown in Figure 5.1. The first verification is to read the original infile into Visual EMU and write an infile from Visual EMU without making changes. The second verification is through user input where all data is entered into Visual EMU by the user instead of through an infile. In both cases, any changes to the infile show the affect of Visual EMU.

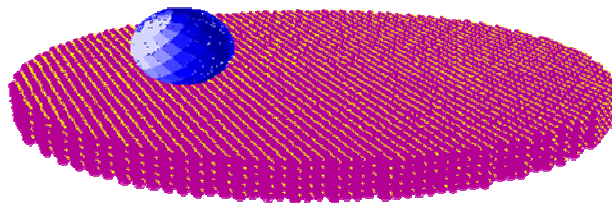


Figure 5.1 EMUGR plot of damage at time 0

Read an EMU infile

EMU requires a text file of keywords in a specific format. Visual EMU can open an EMU infile and apply the settings to Visual EMU. To open an EMU infile, on the **File** menu, click **Read Infile**. In the Open dialog, navigate to the infile named “emu.in.glassplate.” Click **Open** to read the infile. A dialog will appear with a list of lines from the infile that Visual EMU could not recognize. Visual EMU does not recognize any keywords for EMUGR, the EMU post-processor.

Write an EMU infile


Visual EMU can create an EMU infile. To create an infile, on the **File** menu, click **Write Infile**. In the Open dialog, enter “emu.in.glassplateve” as the file name and use the file chooser to navigate to the location you would like to save the infile. Click **OK** to begin writing the infile. A dialog will appear asking for the job description. Enter “Sphere into Glass VE.” This job description becomes the first line in the infile and is also displayed by the post-processor. The infile can be used at any time to run an EMU solution or read the settings into Visual EMU to continue working.

Results

The two infiles are shown in Appendix A with the differences highlighted. The first line is the title line mentioned previously and is different between the two infiles. This has no effect on the solution and gives a unique description to the infile. Other lines that have no effect on the solution are those that begins with a * character. This character indicates a comment and any lines that begin with one are ignored by EMU. Visual EMU takes advantage of this when writing an infile and adds ** followed by the name of the material region when writing material regions to the infile. If the file is read back into Visual EMU, the name is recognized by the ** and applied to the material region when created.

There are three keyword differences between the two files. These differences are keywords that appear in the original infile and not in the resulting infile. These three keywords (*processors*, *one_line_print*, and *density_1*) are not included in the output from Visual EMU because they are EMU default values. As mentioned previously, values that are not different from the EMU default are not written to the infile. All other keywords and the resulting solution from both infiles are the same. The EMU results are shown and explained in greater detail in the EMU results section to come.

Reset Visual EMU

Before starting the second method of verification, the information entered from the infile must be removed. This ensures that the resulting infile contains only information entered by the user. To reset Visual EMU, on the **File** menu, click **New** or click the new button () located on the toolbar. A dialog appears warning the user that all the current information will be lost. Click

OK to continue and restore the default Visual EMU settings. Closing and restarting Visual EMU also resets the settings.

Define the internal grid

The first step in the second method of verification is to define the internal grid. The grid defines a region of nodes that can be assigned to a material region. To edit the internal grid, select the **Grid** tab. Under the **Grid Dimensions** section, enter **X=60**, **Y=60**, and **Z=5**. Also change **Grid Spacing=.001** as shown in Figure 5.2.

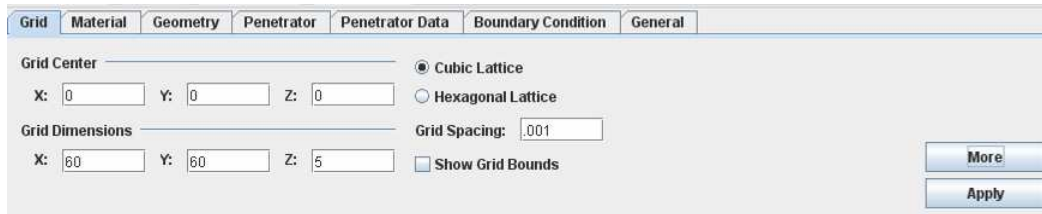


Figure 5.2 Defining the internal grid

During an EMU solution, nodes can move within the grid region and also into a region around the grid region called the grid margin. To change the size of the grid margin, click **More** and enter **X Max=.02**, **X Min=.02**, **Y Max=.02**, **Y Min=.02**, **Z Max=.02**, **Z Min=.05** as shown in Figure 5.3. Click **OK** to close the More - Grid dialog and then click **Apply** on the Grid tab to save the changes.

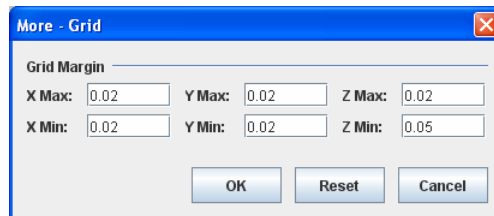


Figure 5.3 Defining the grid margin

Create a material

Materials in Visual EMU hold all material properties. Each material can be applied to any number of material regions. To create a new material, select the **Material** tab. In the **Name** field, enter “Material1” or allow Visual EMU to provide a default name when the material is created. Select **MicroElastic**, select **Linear-Flat**, and enter **sspnom=2600**, **yld=1000e6**, and **ecrit=.001** as shown in Figure 5.4.

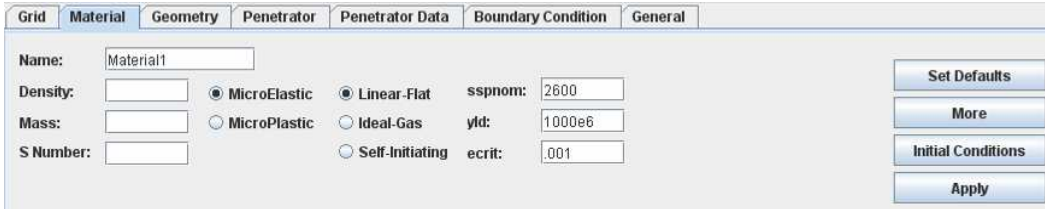


Figure 5.4 Creating a new material

To view additional material property information, click **More**. Select the **Material Properties 1** tab in the More - Material dialog and enter **Failure Stretch Exp=-1** and **Min Stretch Coef=.25** as shown in Figure 5.5.

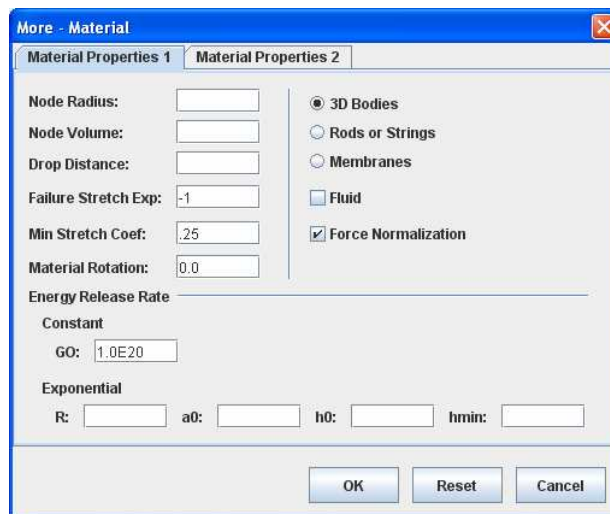


Figure 5.5 Define material properties 1

Select the **Material Properties 2** tab and under **Damage Stretch Coefficient** enter **dc1=.35**, **dc2=1**, and **dc3=2** as shown in Figure 5.6.

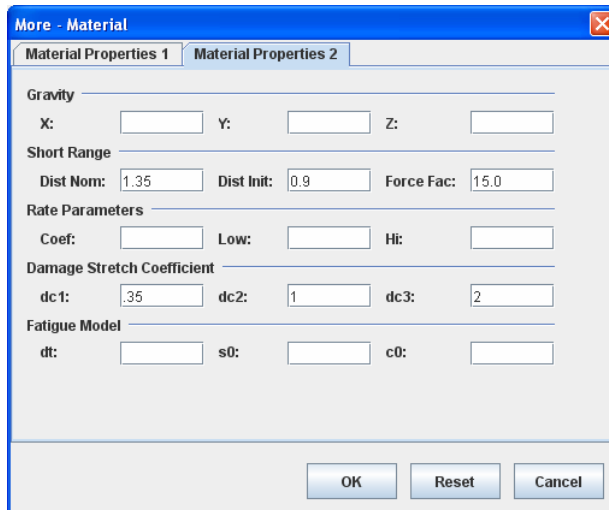



Figure 5.6 Define material properties 2

Click **OK** to close the More - Material dialog and click **Apply** on the Material tab to create the material with the chosen properties. Notice the addition of a new material under Materials on the tree. To edit the material at any time, double click the name of the material or right click the material on the tree and select Edit. Click Apply when finished editing to save the changes.

Create a material region

Material regions apply material properties to the nodes within a boundary. To create a material region, select the **Geometry** tab. Each button across the top of the Geometry tab represents a different material region. When selected, the area below the buttons changes to enter the information necessary to define the specific material region. Select the cylinder button () to define a cylinder geometry. In the **Name** field, enter “Cylinder1” or allow Visual EMU to provide a default name when the material region is created. Enter **X Cen=0**, **Y Cen=0**, **Z Max=0**, **Z Min=-.005**, and **Radius=.03**. Also select “Material1” (or the default name given by Visual EMU) to be the material applied to the region as shown in Figure 5.7.

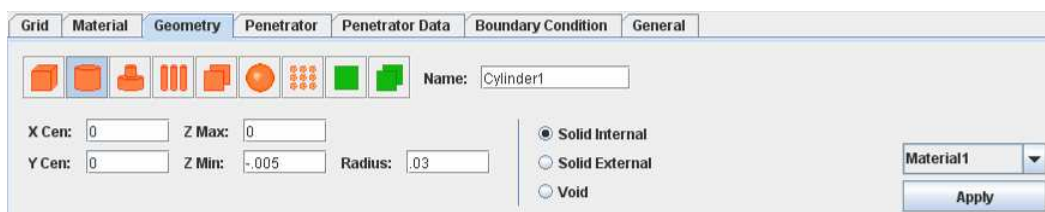


Figure 5.7 Creating a material region

Click **Apply** to create the material region with the information provided. Notice the addition of a new shape under Shapes on the tree. To edit the shape at any time, double click the name of the shape or right click the name and select Edit. Click Apply when finished to save the changes.

Manipulate the view

To manipulate the view of the shape, click the orbit button (⊕) located on the toolbar at the top of Visual EMU. While in the orbit state, use of the mouse in the view panel changes how the shape is viewed. To rotate, click and drag the left mouse button inside the view panel. The shape rotates in the direction of the mouse. To translate, click and drag the right mouse button inside the view panel. The shape translates in the direction of the mouse. To zoom in or out, roll the mouse wheel back or forward respectively. The view can be reset at any time by clicking the reset button (↺) located to the right of the orbit button.

Create a penetrator

Penetrator shapes are primarily used to impact material regions. To create a penetrator, select the **Penetrator** tab. Select **Sphere** and enter **Diameter**=.01 as shown in Figure 5.8. Click **Apply** to create the penetrator. Notice the addition of a new shape under Shapes on the tree. The penetrator can be edited in the same way described previously.

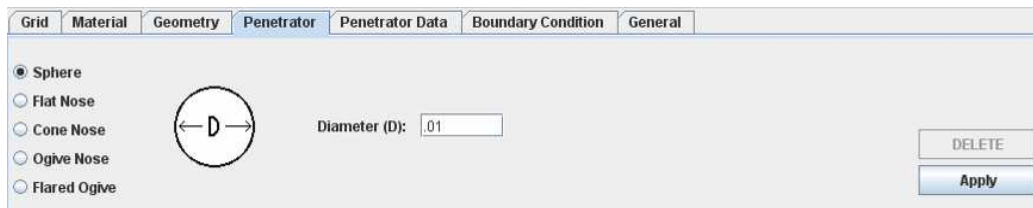


Figure 5.8 Creating a penetrator

To edit the penetrator properties, select the **Penetrator Data** tab. There can only be one penetrator per EMU solution so the penetrator properties are independent of the specific penetrator created. Enter **Penetrator Mass**=4.16e-3, **Angle of Impact**=45, and **Impact Velocity**=100. Under **Penetrator tip** enter **X**=-.01, **Y**=0, and **Z**=.001. Under **Friction** select **Linear** and enter **fricco**=0 as shown in Figure 5.9. Click **Apply** to save the changes which are applied to the penetrator.

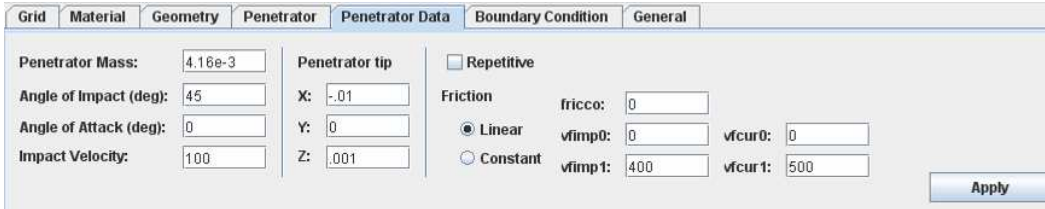


Figure 5.9 Changing the penetrator properties

Set EMU solution parameters

There are a variety of additional settings for EMU. To change these settings, select the **General** tab. Enter **Max time=999**, **Safety Factor=.8**, **Filter time constant=1e-9**, and **Plot Dump Frequency=100** as shown in Figure 5.10. Click **Apply** to save the changes.

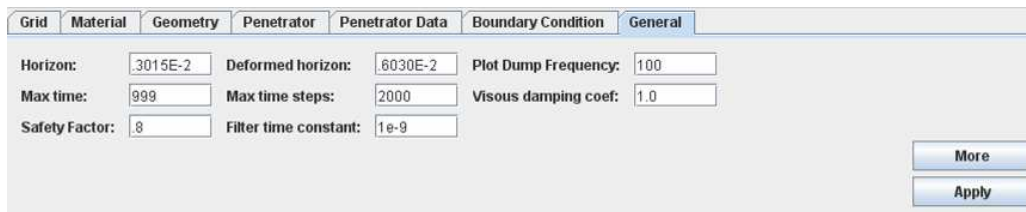


Figure 5.10 Changing additional settings

Write an EMU infile

Write an infile from Visual EMU as previously described. In the Open dialog, enter “emu.in.glassplateve” as the file name and use the file chooser to navigate to the location you would like to save the infile. The resulting infile is shown in Appendix A with the differences highlighted. The title and material region comment are different from the original infile as expected. The same three keywords highlighted in the original infile are still not present in this output for the same reasons explained previously.

The unique difference of this infile is defining the material region. The original infile uses -999 as the minimum z bound while the Visual EMU infile from user data entry has -.005 as the minimum z bound. As described previously, the parts of a material region outside the peridynamic grid have no effect on the solution. For this reason, the larger bound only ensures that the material region reaches the edge of the peridynamic grid. The value of -.005 is enough to reach the edge of the peridynamic grid and allows the user to better view the shape in Visual EMU. Using excessive bounds is an EMU trick that allows the user to ensure the boundary of

the peridynamic grid is reached without calculation or trial and error. The bounds entered through user input result in the same EMU solution.

Run an EMU solution

An EMU solution can be started directly from Visual EMU. To run an EMU solution from Visual EMU, on the **File** menu, click **Run EMU**. Visual EMU can use the current settings or settings from a previously created infile as shown in Figure 5.11. Select **Current settings** and change the **Save location** to the location you would like the EMU solution results to be saved.

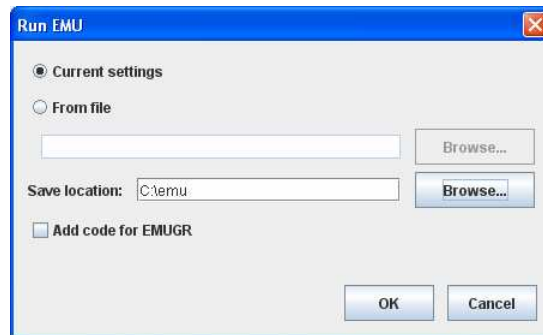


Figure 5.11 Running an EMU solution

The current program used to view EMU results is EMUGR. EMUGR is similar to EMU and requires keywords to operate. The keywords define how the solution should be displayed. Select **Add code for EMUGR** in the Run EMU dialog. This adds some basic keywords to the infile, shown in Figure 5.12. If the code for EMUGR is added, the EMUGR program can be used on the same infile after EMU is finished. To do this, navigate to the chosen save location of Figure 5.11 and double click the “plot.bat” file created by Visual EMU. After EMUGR is finished, the results are viewed by double clicking the “viewPlot.bat” file created by Visual EMU in the same location. These two files are separate allowing the user to view the results without spending the time to run EMUGR again. In addition, the user can change the EMUGR plot variables and run EMUGR without running EMU again.

```
plot_all
plot_all_variables
zoom
150
view_angles
0 80
```

Figure 5.12 EMUGR code added to infile

Click **OK** to begin the EMU solution. A dialog will appear asking for the job description. Enter “Sphere into Glass User Creation from VE.” This job description becomes the first line in the infile and is also displayed by EMUGR. The solution runs in the background and allows the user to manipulate Visual EMU while the solution is running. Future work on Visual EMU should add the ability to view the progress of a solution and cancel a solution.

EMU results

The EMU results are saved in the results folder created by Visual EMU in the location specified by the user. These results are then used by EMUGR along with keywords to display the desired results for the user. The results can be used any number of times by EMUGR without being affected. Figure 5.1 shows the damage, or fraction of broken bonds, before iterations begin. Figure 5.13 shows the damage at time 5.286e-5. The colors indicate different levels of damage and are ordered similar to the colors of a rainbow (ROYGBIV). The red side of the spectrum indicates the max damage, where all bonds are broken, while the violet side indicates little to no damage, where no bonds are broken.

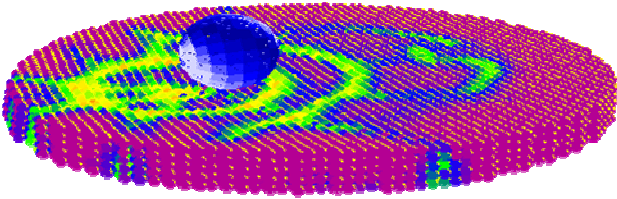


Figure 5.13 EMUGR plot of damage at time 5.286e-5

Figure 5.14 shows the damage at the last time step of the solution. The green color in the middle of the ROYGBIV scale indicates about half the bonds are broken. This implies a crack or fracture in the glass. The red areas where all bonds are broken imply nodes have broken away

altogether. A small section surrounded in green, such as the left of Figure 5.14, indicates a large piece has broken away from the plate.

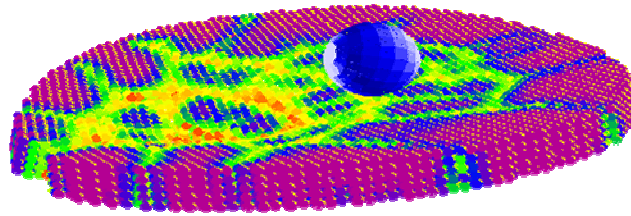


Figure 5.14 EMUGR plot of damage at time 2.646e-4

Small Pipe into Glass Plate

This example evaluates a small pipe impacting a square glass plate. The pipe is created through external grid generation and impacts the glass plate with a velocity normal to the surface. The initial setup is shown in Figure 5.15. To verify Visual EMU, the same two methods of verification are used as in the previous example. Read and write an infile following the same procedure as mentioned previously looking for the file named “emu.in.smallpipe” and creating a file named “emu.in.smallpipeve.”

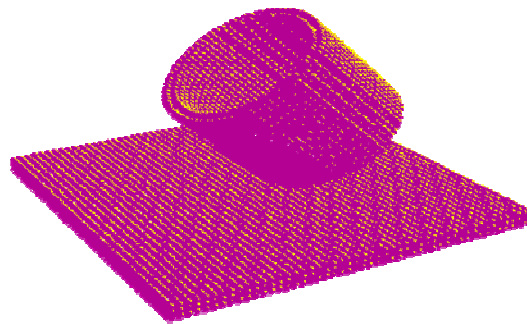


Figure 5.15 EMUGR plot of damage at time 0

Results

The infiles are shown in Appendix A with the differences highlighted. As with the previous example, the title, inclusion of default values (*processors* and *density_1*), and material region comments are different between the two files. The unique difference of this infile is the change in the name of the grid file. When grid files are read, Visual EMU saves and displays the information, allowing the user to change the material assigned to each grid file material region.

When writing an infile, Visual EMU also writes a new grid file to the same location as the new infile. The grid file contains all the changes and keeps the original grid file unchanged and unmoved.

Define the internal grid

As before, reset Visual EMU before continuing to the second method of verification. The internal grid of this example only needs defined for the glass plate. The nodes defined externally in the grid file are placed where specified regardless of the internally generated nodes. To edit the internal grid, select the **Grid** tab. Under **Grid Dimensions**, enter **X=50**, **Y=50**, and **Z=20**. Also enter **Grid Spacing=.001** as shown in Figure 5.16. Click **Apply** to save the changes.

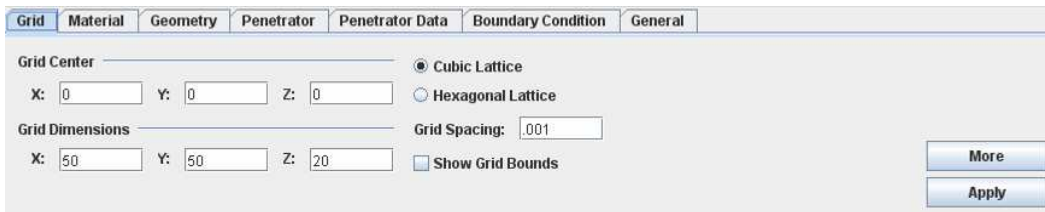


Figure 5.16 Setting the grid

Create materials

To create the first material, select the **Material** tab. In the **Name** field, enter “Mat1” or allow Visual EMU to provide a default name when the material is created. Select **MicroElastic**, select **Linear-Flat**, and enter **sspnom=2600**, **yld=200e6**, and **ecrit=.001** as shown in Figure 5.17. Click **Apply** to create the material with the chosen properties.

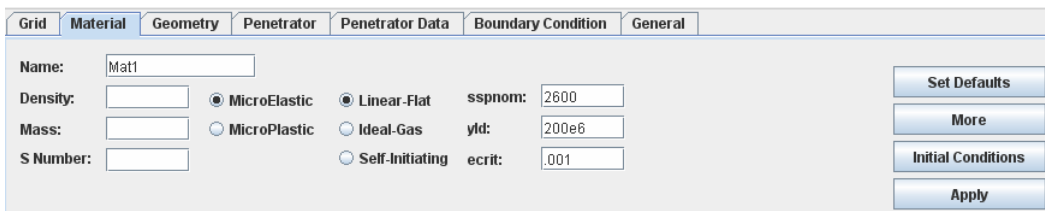


Figure 5.17 Creating the first material

To create the second material, in the **Name** field, enter “Mat2” or allow Visual EMU to provide a default name when the material is created. Enter **Density=8000**, select **MicroElastic**, select **Linear-Flat**, and enter **sspnom=4000**, **yld=400e6**, and **ecrit=.2** as shown in Figure 5.18.

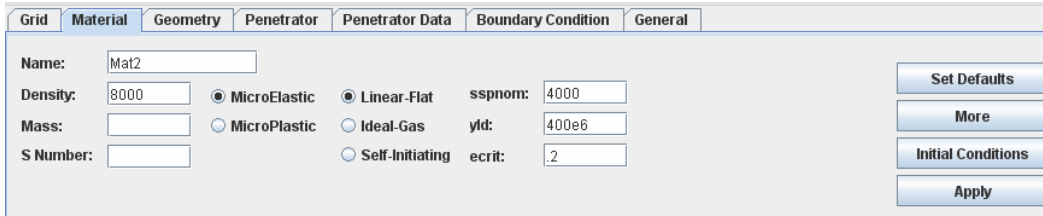


Figure 5.18 Creating the second material

The pipe needs an initial velocity to impact the glass plate. The initial velocity is applied through the material assigned to the material region. To apply the velocity to the material, click the **Initial Conditions** button and select the **Velocity** tab in the Initial Conditions – Material dialog. Enter **VZ=-100**, as shown in Figure 5.19, and click **OK** to save the changes and close the dialog. Click **Apply** on the Material tab to create the material with the chosen properties.

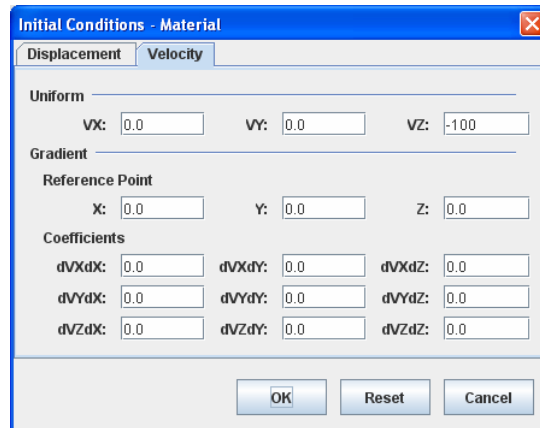



Figure 5.19 Specifying an initial velocity

Create a material region

To create the material region, select the **Geometry** tab. Select the rectangle button () to define a rectangle geometry. In the **Name** field, enter “Rectangle1” or allow Visual EMU to provide a default name when the material region is created. Enter **X Max=.025**, **X Min=-.025**, **Y Max=.025**, **Y Min=-.025**, **Z Max=-1e-6**, and **Z Min=-.0031**. Also select “Mat1” (or the default name provided by Visual EMU) to be the material applied to the region as shown in Figure 5.20.

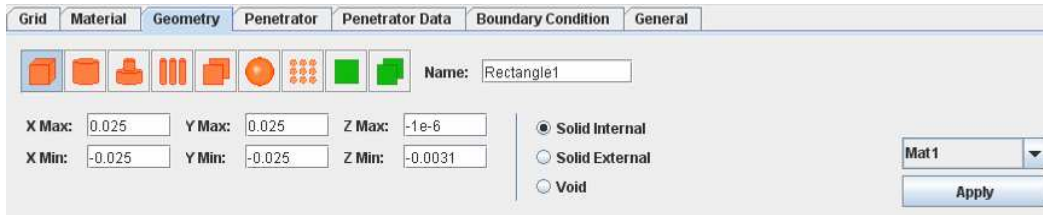


Figure 5.20 Creating a material region

Click **Apply** to save the changes and create the material region. Notice the addition of a new shape under Shapes on the tree. To edit the shape at any time, double click the name of the shape or right click the name and select **Edit**. Click Apply when finished to save the changes.

Manipulate the view

To manipulate the view of the shape, click the orbit button (↻) located on the toolbar at the top of Visual EMU. While in the orbit state, use of the mouse in the view panel changes how the shape is viewed. To rotate, click and drag the left mouse button inside the view panel. The shape rotates in the direction of the mouse. To translate, click and drag the right mouse button inside the view panel. The shape translates in the direction of the mouse. To zoom in or out, roll the mouse wheel back or forward respectively. The view can be reset at any time by clicking the reset button (↺) located to the right of the orbit button.

Add a grid file

The second material region comes from an externally generated grid file. To add the grid file, select the grid file button (📄) on the Geometry tab. In the **Name** field, enter “Grid File1” or allow Visual EMU to provide a default name when the grid file is added. Click **Browse...** to navigate to the grid file named “smallpipe.grid” as shown in Figure 5.21.

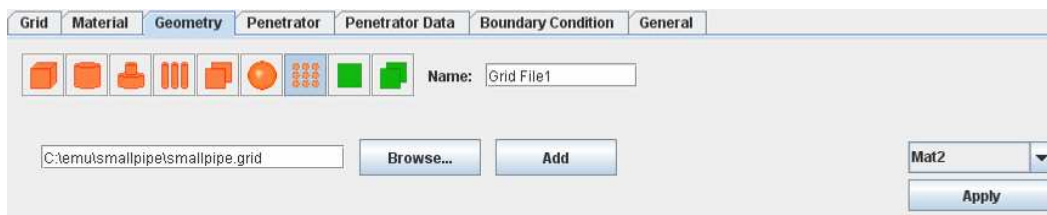


Figure 5.21 Adding a grid file

Click **Add** to open the grid file and begin adding the nodes. Each node of the grid file is assigned a material region. Visual EMU allows the user to choose what material should be

applied to each region. In the Select Material dialog that appears, select “Mat2” (or the default name provided by Visual EMU) to be assigned to material region 2 in the grid file as shown in Figure 5.22.



Figure 5.22 Assigning Mat2 to material region 2

Set EMU solution parameters

To change additional settings, select the **General** tab. Enter **Max time**=999, **Safety Factor**=.8, and **Plot Dump Frequency**=50 as shown in Figure 5.23.

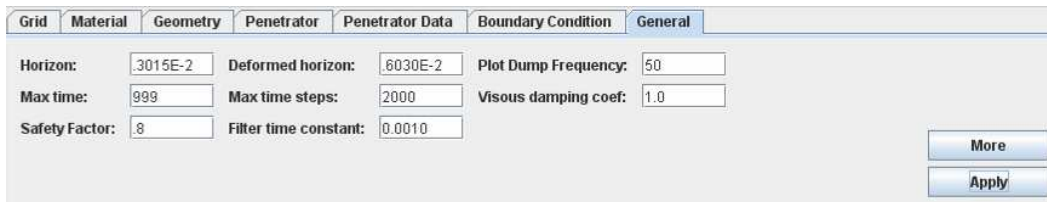


Figure 5.23 Changing additional settings

The two material regions start close together and, by default, have peridynamic bonds between the nodes within the material horizon distance. To keep the two material regions separate without peridynamic bonds between them, click the **More** button. In the More – General dialog click the **Interface** tab. When both material regions are selected, the options below the list are enabled. To select both of the material regions, use the mouse to left click on the first material region on the list. To select the second while keeping the first selected, hold down the control button and left click on the second material region. When both material regions are highlighted, as shown in Figure 5.24, select **Disconnect** and click **Apply** to save the change. Click **OK** to close the More - General dialog and click **Apply** on the General tab to save all changes.

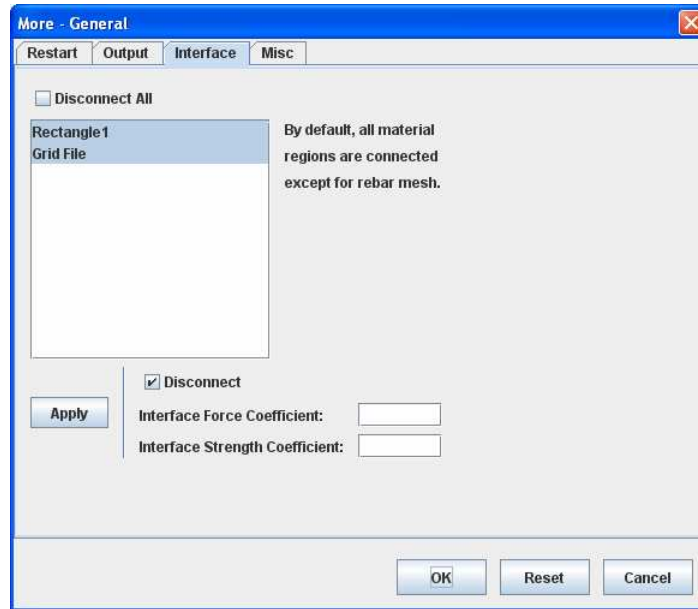


Figure 5.24 Disconnecting the material regions

Write an EMU infile

Write an infile from Visual EMU as previously described. In the Open dialog, enter “emu.in.smallpipeve” as the file name and use the file chooser to navigate to the location you would like to save the infile.

Results

The resulting infile is shown in Appendix A with the differences highlighted. As with the infile created from reading the original infile, the title, inclusion of default values (*processors* and *density_1*), material region comment, and the grid file name are different from the original infile.

The unique difference of this infile is the material region parameters. Similar to the previous example, the original infile uses 999 and -999 as minimum and maximum values for the x and y bounds of the material region. As described previously, the larger bounds only ensure that the material region reaches the edge of the peridynamic grid and the bounds entered through user input result in the same EMU solution.

EMU Results

After running EMU as described previously or outside of Visual EMU, view the results with EMUGR. Figure 5.15 shows the damage at time 0, Figure 5.25 the damage at time 8.154e-

5, and Figure 5.26 the damage at time $1.816e-4$. The pipe, which is four times denser than the glass, hits the plate with an edge and quickly breaks through the glass. The region around where the pipe hits the plate shatters, indicated by the red and orange nodes. The cracks are indicated primarily by the green lines from the center to the edge of the plate. As the solution continues, the pipe continues into the plate and nodes that have broken free are visible inside the pipe.

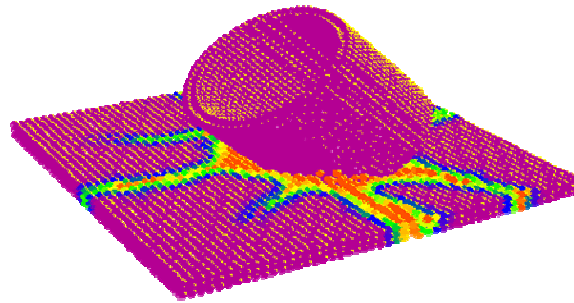


Figure 5.25 EMUGR plot of damage at time $8.154e-5$

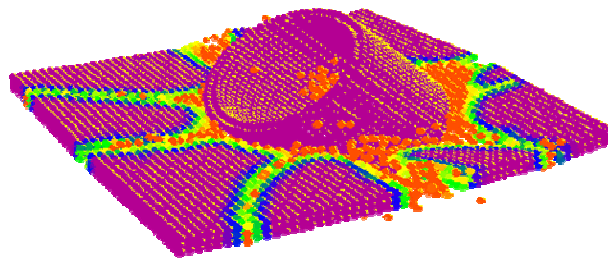


Figure 5.26 EMUGR plot of damage at time $1.816e-4$

CHAPTER 6 - Conclusions

Summary

Visual EMU accurately reads and writes EMU infiles. The user can visualize the material regions and their placement relative to the peridynamic grid before performing an EMU solution. The user can create a material once and apply that same material to any number of material regions without the need to repeat entering information. Multiple view options are present allowing the user to see all shapes as solids, wire frames, or a preview of the nodes in EMU. Visual EMU allows the user to run EMU from the current settings or from a saved infile. The user interface is adequate for the entry of all EMU keywords though areas for improvement exist and are mentioned in more detail below.

Future work

The view panel would benefit greatly from the ability to select shapes. Selecting three dimensional shapes viewed with a two dimensional screen can be a complicated process. Once selection is available though it would be a trivial addition to edit and delete shapes directly from the view panel.

The entry of all keywords in Visual EMU is functional but not ideal in some cases. Some dialogs (Figure 4.10 for example) organize keywords by the number of inputs and should organize them by function or usage. Each dialog and panel would also benefit from additional guidance. Some ideas are help buttons that explain keywords and interactive equations that show how keywords are used.

The penetrator is an important feature in EMU and would benefit from more advanced visualization in Visual EMU. Improvements include drawing the penetrator at an angle when the angle of impact is changed and adding a direction vector or some indication of the angle of attack specified by the user. Another improvement that is now in progress aims to show the user feedback from EMU as the solution progresses and add the option to cancel a solution in progress.

The concept of a new post-processor for EMU has also already begun. The solution would be displayed from Visual EMU and allow the user to create, run, and view results from one program. Ideally, the user will be able to view any parameter at any time step in 3D and even automate the display to step through frames at a given speed. The user will also be able to rotate the solution to view any angle.

References

- Geary, D., 2003, "Simply Singleton," Java World, <http://www.javaworld.com/javaworld/jw-04-2003/jw-0425-designpatterns.html>.
- Horstmann, C., 2006, *Big Java*, Hoboken, NJ, John Wiley & Sons, Inc.
- "Java Platform Standard Ed. 6," 2006, Sun Microsystems Inc.,
<http://java.sun.com/javase/6/docs/api/>
- Macek, R., and Silling, S. A., 2006, "Peridynamics via Finite Element Analysis," Report LA-14300, Los Alamos National Laboratory, Los Alamos, NM.
- Microsoft Manual of Style for Technical Publications*, 2004, Redmond, WA, Microsoft Press.
- Morelli, R., and Walde, R., 2006, *Java, Java, Java Object Oriented Problem Solving*, Upper Saddle River, NJ, Pearson Prentice Hall.
- Potyondy, D. O., and Cundall, P. A., 2004, "A Bonded-Particle Model for Rock," *Int. J. Rock Mech. & Min. Sci.*, 41(8), 1329-1364.
- Silling, S. A., 1998, "Reformulation of Elasticity Theory for Discontinuities and Long-Range Forces," Sandia National Laboratories, Albuquerque, NM.
- Silling, S. A., 2002, "Peridynamic Modeling of the Failure of Heterogeneous Solids," Sandia National Laboratories, Albuquerque, NM.
- Silling, S. A., and Askari, E., 2004, "A meshfree method based on the peridynamic model of solid mechanics," Sandia National Laboratories, Albuquerque, NM.
- Silling, S. A., Demmie, P. N., Cole, R. A., and Taylor, P., 2006, "EMU User's Manual," Sandia National Laboratories, Albuquerque, NM.

Appendix A - Infile results

The following infiles help verify the results of Visual EMU. For each of the two examples, the original infile is part of the code package from Sandia National Laboratories. The other two infiles are products of Visual EMU. The first is made after reading in the original infile and the second is made after entering the information through the user interface.

Sphere into glass plate

The following three infiles are from the example of a spherical penetrator impacting a glass plate.

Original Infile

```
Sphere Into Glass
processors
  1 1 1
grid_dimensions
  60 60 5
grid_spacing
  0.001
grid_margin
  0.02 0.02  0.02 0.02  0.05 0.02
max_time
  999
safety_factor
  0.8
max_time_steps
  2000
plot_dump_frequency
  100
one_line_print
  0
* start run
*
number_of_material_regions
  1
material_region_geometry_1
  3
  0.03  0 0  -999 0
*
density_1
  2200
microelastic_1
  1 2600 1000.0e6  0.001
min_stretch_coef_1
  0.25
```

```

damage_stretch_coef_1
  0.35 1 2
failure_stretch_exponent_1
  -1
*fnorm_off_all
*
angle_of_attack
  0
angle_of_impact
  45
impact_velocity
  100
penetrator_shape
  4 0.010
penetrator_mass
  4.16e-3
penetrator_friction_coef
  0
penetrator_tip_location
  -0.01 0 0.001
filter_time_constant
  1.0e-9

```

Read/Write Infile

```

Sphere into Glass VE
grid_dimensions
  60 60 5
grid_spacing
  0.0010
grid_margin
  0.02 0.02 0.02 0.02 0.05 0.02
max_time_steps
  2000
max_time
  999
plot_dump_frequency
  100
safety_factor
  0.8
filter_time_constant
  1.0E-9
number_of_material_regions
  1
**Cylinder 1
material_region_geometry_1
  3 0.03 0.0 0.0 -999.0 0.0
microelastic_1
  1 2600.0 1.0E9 0.0010
failure_stretch_exponent_1
  -1.0
min_stretch_coef_1
  0.25
damage_stretch_coef_1
  0.35 1.0 2.0
penetrator_shape
  4 0.01

```

```
penetrator_tip_location
  -0.01 0.0 0.0010
penetrator_mass
  0.00416
penetrator_friction_coef
  0.0
angle_of_impact
  45.0
angle_of_attack
  0.0
impact_velocity
  100.0
```

User Visual EMU infile

Sphere Into Glass User Creation from VE

```
grid_dimensions
  60 60 5
grid_spacing
  0.0010
grid_margin
  0.02 0.02 0.02 0.02 0.05 0.02
max_time_steps
  2000
max_time
  999
plot_dump_frequency
  100
safety_factor
  0.8
filter_time_constant
  1.0E-9
number_of_material_regions
  1
```

**Cylinder

```
material_region_geometry_1
  3 0.03 0.0 0.0 -0.005 0.0
microelastic_1
  1 2600.0 1.0E9 0.0010
failure_stretch_exponent_1
  -1.0
min_stretch_coef_1
  0.25
damage_stretch_coef_1
  0.35 1.0 2.0
penetrator_shape
  4 0.01
penetrator_tip_location
  -0.01 0.0 0.0010
penetrator_mass
  0.00416
penetrator_friction_coef
  0.0
angle_of_impact
  45.0
angle_of_attack
  0.0
```

```
impact_velocity
100.0
```

Small pipe into glass plate

The following three infiles are from the example of a small pipe impacting a glass plate.

Original infile

Pipe Against a Block

```
processors
1 1 1
grid_dimensions
50 50 20
max_time
999
max_time_steps
2000
plot_dump_frequency
50
grid_spacing
0.001
one_line_print
1
number_of_material_regions
2
grid_file
1
smallpipe.grid
material_region_geometry_1
1
-999 999 -999 999 -0.0031 -0.000001
density_1
2200
density_2
8000
microelastic_1
1 2600 200.0e6 0.001
microelastic_2
1 4000 400.0e6 0.2
material_region_ic_2
0 0 0 0 0 -100
disconnect
1
1 2
safety_factor
0.8
```

Read/Write Infile

```
Pipe Against a Block VE
grid_dimensions
50 50 20
```

```

grid_spacing
  0.0010
max_time_steps
  2000
max_time
  999
plot_dump_frequency
  50
safety_factor
  0.8
one_line_print
  1
number_of_material_regions
  2
**Rectangle 1
material_region_geometry_1
  1 -999.0 999.0 -999.0 999.0 -0.0031 -1.0E-6
microelastic_1
  1 2600.0 2.0E8 0.0010
grid_file
  1
  gridfile792.grid
density_2
  8000.0
microelastic_2
  1 4000.0 4.0E8 0.2
material_region_ic_2
  0.0 0.0 0.0 0.0 0.0 -100.0
disconnect
  1
  1 2

```

User Visual EMU infile

```

Pipe Against a Block User VE
grid_dimensions
  50 50 20
grid_spacing
  0.0010
max_time_steps
  2000
max_time
  999
plot_dump_frequency
  50
safety_factor
  0.8
one_line_print
  1
number_of_material_regions
  2
**Rectangle
material_region_geometry_1
  1 -0.025 0.025 -0.025 0.025 -0.0031 -1.0E-6
microelastic_1
  1 2600.0 2.0E8 0.0010
grid_file

```

```
1
gridfile409.grid
density_2
8000.0
microelastic_2
1 4000.0 4.0E8 0.2
material_region_ic_2
0.0 0.0 0.0 0.0 0.0 -100.0
disconnect
1
1 2
```