TEALsim: A Guide to the Java 3D Software

NOTE: Many of the links in this document will only work if you have installed this *pdf* file in the location described in Section 2.1 below.

IN PDF FORMAT THIS DOCUMENT HAS BOOKMARKS FOR NAVIGATION

CLICK ON THE UPPER LEFT "BOOKMARK" TAB IN THE PDF READER

Version 1.1 December 16, 2007

John Belcher, Andrew Mckinney, Philip Bailey, Michael Danziger

Comments and questions to jbelcher@mit.edu

Supported by NSF Grant DUE-0618558, the d'Arbeloff Fund for Excellence in MIT Education, the Davis Educational Foundation, iCampus, the MIT School of Science, NSF Grant DUE-9950380, the Helena Foundation, and the Classes of 1951, 1955, and 1960.

Table of Contents
(*LIVE LINKS*)

Figure Captions

1 Introduction

1.1 Purpose and Intended Audience

The TEAL simulation system, *TEALsim*, is designed as a framework for authoring, presenting, and controlling simulations in a variety of domains, beginning with electromagnetism and mechanics, with extensions to biochemistry. The goal of *TEALsim* is twofold: first, it aims to provide a relatively simple, "API-style" interface, such that non-expert programmers can produce full-featured interactive simulations from the ground up, including all aspects of visualization and user interface, with minimal exposure to the inner workings of the system. Second, it offers a flexible framework for more experienced programmers to extend and expand on the functionality of the system to suit their specific needs. That is, it is built in such a way that the sophisticated user can build simulations in other domains in a straightforward and well defined way.

This document is intended as a general guide to building simulations using the *TEALsim* environment, and should be used in conjunction with the *JavaDocs* for the TEALsim project. The *TEALsim* environment was developed by the Technology Enabled Active Learning (TEAL) Project at MIT. Many examples of the software produced by this project are available on line at

http://web.mit.edu/8.02t/www/802TEAL3D/teal_tour.htm

For those interested in building simulations using the existing platform, our intended audience is undergraduates, graduate students or postdoctoral fellows in science and engineering who have some knowledge of programming but little if any knowledge of 3D graphics and conventions (e.g. scene graphs, branch groups, view platforms, and so on). We have assembled a set of instructions and tutorials that will enable the user with that background to create, package, and post 3D simulations in a short time. We have also provided enough insight into the workings of Java3D for the uninitiated to have a passing knowledge of what goes on, and we have provided references to books and online resources for those interested in expanding that knowledge beyond what we present here.

The organization of this documentation is as follows. We first give instructions for downloading and installing the code in Section 2, including instructions for running the code. We then give a general description of the *TEALsim* architecture in Section 3. In Section 4, we give examples of building electromagnetic simulations out of standard components in the *TEALsim* universe. In Section 5, we discuss the structure of the code in detail, package by package. There are cross-references in each of these sections to topics in other sections.

1.2 What's New in Release 1.01?

Compared to release 1.0, we have …...

1.3 What's New in Release 1.02?

Compared to release 1.01, we have added……...

2 Setting up the *TEALsim* Project

2.1 Downloading and Installing the *TEALsim* Code Base and Documentation

Go to http://web.mit.edu/viz/soft/ and follow links to the *TEALsim* code base. You will download three zipped files, one under the link *Source Code* and named *TEALsim.zip*, the second under the link *TEALsim Documentation* and named *TEALsimDoc.zip*, and the third under the link *General Documentation* and named *generalDoc.zip*. Note that if you have previously downloaded *generalDoc.zip* to support the *SundquistDLIC* code for *Dynamic Line Integral Convolution* programs, you do not need to download this files again, as it is identical to the one in the DLIC download.

Create a folder *C:\Development\Projects\* on your C-drive and unzip *TEALsim.zip* into the *Projects* folder. Unzip the *TEALsimDoc.zip* into the *C:\Development\Projects\TEALsim* folder. Unzip the *generalDoc.zip* file into the *C:\Development\Projects\* folder. Once you have done this you will have installed the *TEALsim* code and documentation. All of the documentation discussed below is now contained in the two folders *C:\Development\Projects\TEALsim\TEALsimDoc* and *C:\Development\Projects\generalDoc*. In particular this document in *pdf* format is contained in the folder *TEALsimDoc*. If you are not reading this document from that folder, many of the links will not work.

In what follows we will provide links to the documentation on your C-drive assuming that you have installed the source code and documentation as described above. In particular when we discuss particular *java* files, we will link directly to those files. You may want to set the default application for reading *.java* files to *Notepad*. To do this, right click on any *.java* file, choose "*Open With > Choose Program*" from the dialog box (see Figure 1.2-1) and in the dialog box that then comes up, choose "*Select the program from a list*". In the list that comes up, select *Notepad* and make sure you check the box labeled "*Always use the selected program to open this kind of file*".

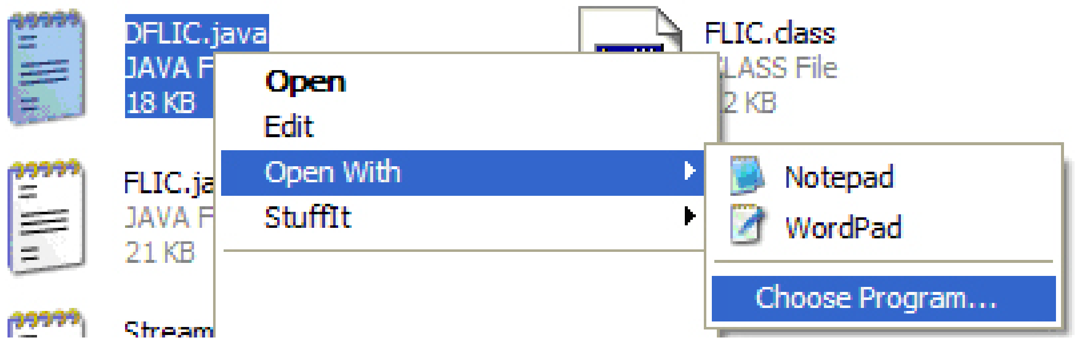


Figure 2.1-1: Setting the default *.java* application reader to *Notepad*

2.2 Installing *Java* and *Eclipse* in a Windows Environment

We discuss the *TEALsim* software in the context of *Eclipse* (http://www.eclipse.org/), a free, state-of-the-art Java Integrated Development Environment (IDE). We will assume that you are familiar with the *Eclipse* IDE interface, and developing *Java* applications in this environment. Step by step instructions for downloading *Eclipse* and the *Java SDK* (SDK stands for *Software Development Kit*) and creating a *SundquistDLIC* Java Project in *Eclipse* are given in a *pdf* file in your *generalDo*c folder at this link. To reiterate what active links in this document mean, "*right-clicking*" on the preceding link will take you to a *pdf* file named

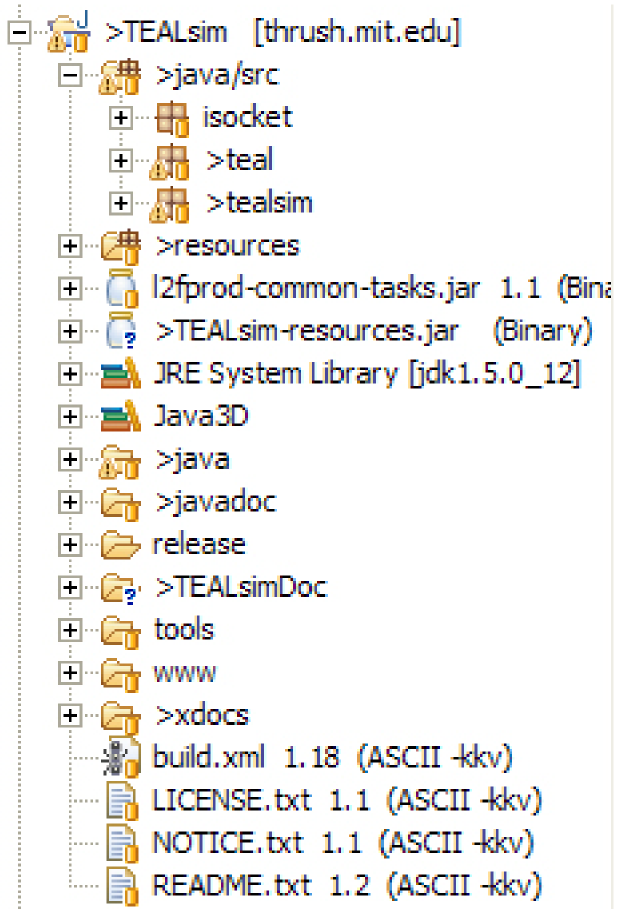
InstallingJavaEclipseInWindows.pdf

in the folder *C:\Development\Projects\generalDoc\*. If you have downloaded and installed the code base and documentation in accord with the instructions in Section 2.1 above, the preceding link will open this document.

We also provide a short document with common hints for using *Eclipse*, which explicitly tells you how to perform the various actions described below in *Eclipse*. This includes explanations for how to perform certain actions whose explanations are hard to find in the standard *Hel*p documentation on the *Eclips*e taskbar. That document in *pdf* format is at this link.

2.3 Creating a *TEALsim* Java Project in *Eclipse*

Follow the directions in Section 3.2 in the *pdf* file at this link. When you have done this, open a *Package Explorer* window for the Project. You will see a directory tree for the packages in *TEALsim* in *Eclipse* that looks like the one given on the next page in Figure 2.3-1.

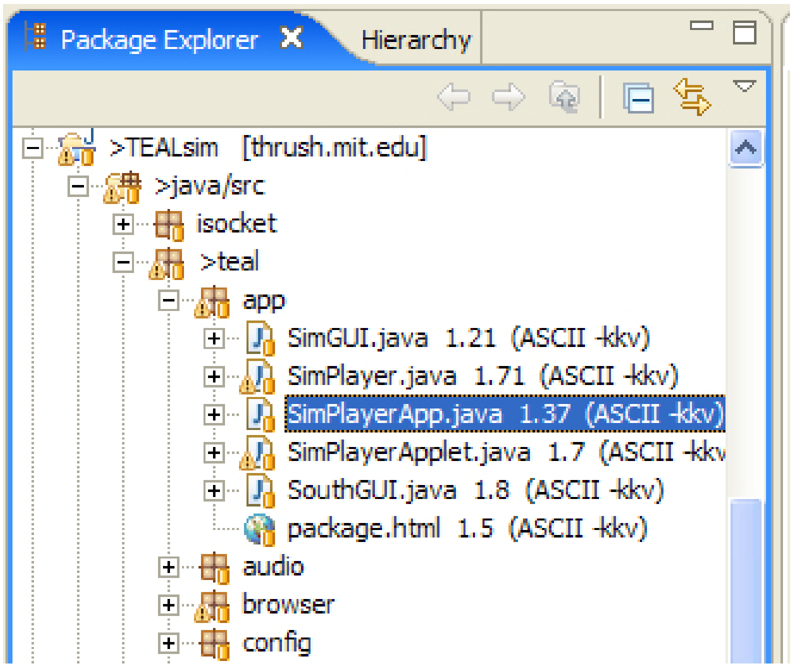


```
>TEALsim   [thrush.mit.edu]
    >java/src
        isocket
        >teal
        >tealsim
    >resources
    l2fprod-common-tasks.jar  1.1  (Bina
    >TEALsim-resources.jar    (Binary)
    JRE System Library [jdk1.5.0_12]
    Java3D
    >java
    >javadoc
    release
    >TEALsimDoc
    tools
    www
    >xdocs
    build.xml  1.18  (ASCII -kkv)
    LICENSE.txt  1.1  (ASCII -kkv)
    NOTICE.txt  1.1  (ASCII -kkv)
    README.txt  1.2  (ASCII -kkv)
```

Figure 2.3-1: The Package Structure of the *SundquistDLI*C Project

2.4 Instructions for Running a Simulation

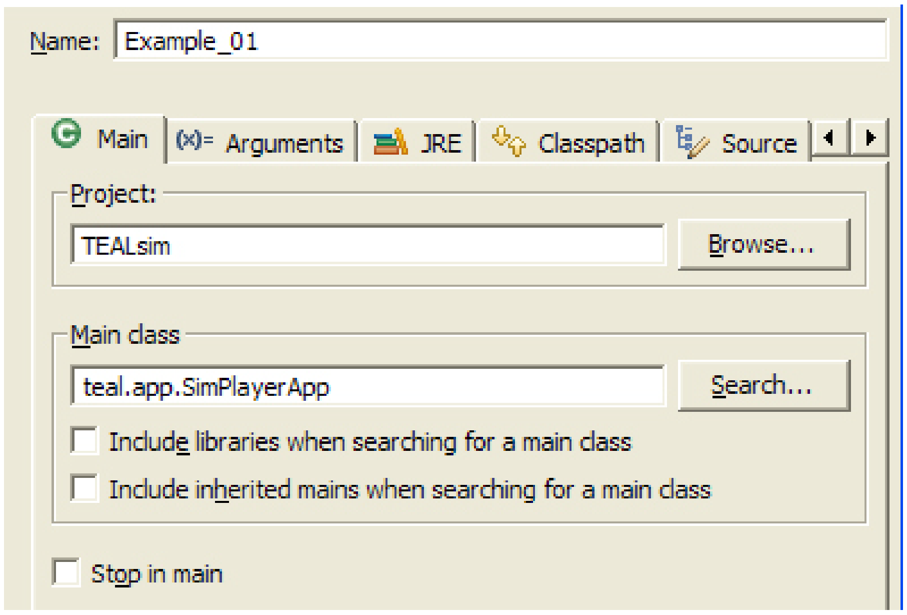
In *Eclipse*'s *Package Explorer*, choose *SimPlayerApp.java* (see Figure 2-4.1) by left-clicking on it.



```
>TEALsim  [thrush.mit.edu]
  >java/src
    isocket
    >teal
      app
        SimGUI.java  1.21  (ASCII -kkv)
        SimPlayer.java  1.71  (ASCII -kkv)
        SimPlayerApp.java  1.37  (ASCII -kkv)
        SimPlayerApplet.java  1.7  (ASCII -kkv
        SouthGUI.java  1.8  (ASCII -kkv)
        package.html  1.5  (ASCII -kkv)
      audio
      browser
      config
```

Figure 2.4-1: Starting a simulation

Right-click and choose "*Run as* > *run*" from the menu. You will see a dialog box (see Figure 2.4-2). Enter "Example_01" in the *Name* box. Left-click on the *Arguments* tab and in the box labeled *Program arguments* (see Figure 2.4-3). Type "-n tealsim.physics.examples.Example_01" in the box on top (see Figure 2.4-3). Left-click on *Run* at the bottom of the dialog box. The *Example_01* simulation will come up.



Name: Example_01

Project: TEALsim

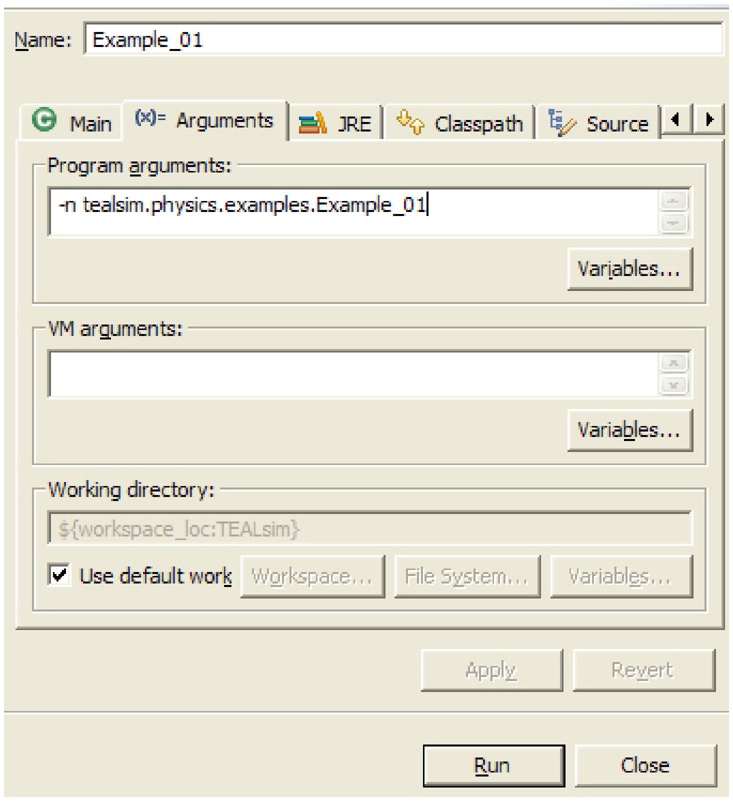
Main class: teal.app.SimPlayerApp

☐ Include libraries when searching for a main class

☐ Include inherited mains when searching for a main class

☐ Stop in main

Figure 2.4-2: The *Run* dialog box



Name: Example_01

Program arguments:

```
-n tealsim.physics.examples.Example_01
```

VM arguments:

Working directory:

${workspace_loc:TEALsim}

☑ Use default work

Figure 2.4-3: The Arguments panel in the *Run* dialog box

If you have a simulation that requests additional memory, set more memory by typing "*-Xmx512m*" (or "*-Xmx256m*") in the "VM arguments:" box shown in Figure 2.4-3.

3 *TEALsim* Overview

The TEAL simulation system, *TEALsim*, is designed as a framework for authoring, presenting, and controlling simulations in a variety of domains, beginning with physics. Architecturally, *TEALsim* follows a "*Model-View-Control*" design pattern, with several major modules representing the three components: the simulation engines make up the "model"; the renderer and viewer make up the "view"; and the user interface makes up the "control." These three components are combined into a simulation. Each component is largely defined by a set of interfaces that suggest the required functionality of that piece, so that the actual implementation details can be customized as necessary (for example, leaving the specific rendering implementation up to the developer). *TEALsim* is built using Java Standard Edition 1.5.0_12, Swing and Java3D 1.4.0_01 extensions. All major components of the system are JavaBeans. Most classes implement well defined interfaces, the most basic being *TElement*.

What follows is a brief description of the basic components.

3.1 The Simulation (TSimulation, SimEM)

The TSimulation interface defines the requirements for a complete interactive simulation, collecting together all of the components that make up the entire user experience. Typically this includes: a simulation engine (*source.java.core.teal.sim.engine.TSimEngine*), a Viewer (*source.java*.core.*teal.render.TViewer*), the UI elements, and all of the objects being simulated or otherwise displayed in the Viewer. *SimEM* (*teal\physics\physical\em)* is an example of a class that implements *TSimulation*. It incorporates the *EMEngine* and a Java 3D viewer and provides the basis for all Electromagnetic simulations.

Any simulation object added to a *simulation* must implement the *TElement* interface, which provides a standard set of functionalities for all objects in the world. That is, *any* object which is specified, included or defined within the simulation must implement the *TElement* interface. This includes physical objects (e.g. a point charge), graphical elements, control objects and simulation viewers. In particular, the functionalities inherent in the *TElement* interface include support for *Routes* and *PropertyChangeEvents*. This allows any simulation elements in the world to exchange information with any other element in the world, and in particular with User Interface (UI) components. For example, a UI slider can be wired to a property of a simulation object (for example, the mass of an electric point charge) to directly manipulate that quantity. Or, the *simulation* itself can be wired to a property of a simulation object in order to monitor that property of the object and/or take some action depending on its value.

A completely realized *TSimulation* class defines the properties of the simulation as a whole, such as the initial spatial configuration of simulation objects, initial conditions of any variables, and any special "wiring" between objects and/or interface components. Essentially, it contains the entire logic specific to a particular simulation.

A T*Simulation* object is then presented to the user using a *SimPlayer*, as discussed below.

3.2 SimPlayer (*TFramework*) and SimPlayerApp

A *SimPlayer* "plays" a T*Simulation* instance—that is it takes as input a T*Simulation* object and presents that T*Sim*ulation to the viewer. It implements the *TFramework* interface, which is the glue that holds all of the components in the T*Simulation* together. *SimPlayer* is the application inside which all of the components of the T*Sim*ulationare running. The *TFramework* interface contains methods that allow loaded simulations to customize the *SimPlayer* application window, including modification of the UI and menu bars.

SimPlayerApp contains the main routine of the *TEALsim* package. When we run *SimPlayerApp* as specified above in Section 2.4 above, it instantiates a *SimPlayer* and uses the argument *-n tealsim.physics.em.Example_01* to create an instance of the class *Example_01* and tells the *SimPlayer* to play *Example_01*, which is a *Simulation* object.

3.3 SimEngine – the Simulation Engine

The physics in a *TEALsim* simulation is contained in a *SimEngine* object, which represents the simulation engine itself (*source.java.core.teal.sim.engine.SimEngine)*. The simulation engine is responsible for all of the computation involved in the system being simulated. This includes dynamically processing and updating simulation objects (*TSimElements*) according to the rules of the simulation engine (including adding and removing them from the "world" when necessary), and performing numerical integration of simulation variables. The exact type of processing and integration will depend on the specific type of simulation being implemented; for example, the "electromagnetism" extension of *SimEngine*, *EMEngine*, computes the total electromagnetic fields created by field-generating simulation objects (*EMObjects*), as well as the resulting dynamics of the *EMObjects* (velocity, position, rotation, etc.). It also handles related tasks, such as collision detection and resolution.

In general, the *SimEngine* runs a continuous loop that performs the following actions:

1) Computes values of dependent simulation variables for the current time step.
2) Updates simulation objects to reflect new values.
3) Informs the renderer of any visual changes to the simulation.

This loop in the *SimEngine* represents the main application thread for a *TEALsim* simulation.

3.4 Viewer and Viewer3D – the Rendering Engine

The *Viewer* (*java.src.teal.render.viewer*) is the window into the simulation space, representing the rendering engine and its output. It is responsible for rendering the visual elements of a simulation to the screen in real-time 3D, and managing user interaction with the rendered image. As such, it is tightly coupled to the *SimEngine*: The *SimEngine* must inform the *Viewer* of (visual) changes to simulation elements, and each visual simulation element must have an associated visual representation that can be drawn by the *Viewer*. Conversely, the *Viewer* must report back to the simulation when a user manipulates the visual representation of a simulation object (for example, by clicking and dragging on an object in the *Viewer*).

While the actual implementation of the *Viewer* can vary, the interface *TViewer* defines a set of functionality that any *Viewer* implementation should support. This includes general rendering properties and tasks, such as camera controls, visual effects, maintaining lists of rendered objects, and handling mouse-based "picking" and manipulation of objects in the *Viewer*. In addition, it should also handle the explicit rendering of the scene.

The current default *Viewer* implementation is based on Java3D, which is a scene-graph based renderer layered on top of *OpenGL* or *DirectX*. In this case, the *Viewer* sets properties on the scene graph, which is then rendered implicitly through Java3D's rendering thread. A more direct (or "immediate") rendering implementation (such as rendering directly through *OpenGL*) should obviously include explicit rendering instructions for all visual elements.

3.5 The User Interface

The user interface (*java.src.teal.ui*) is the means by which a user interacts with the application (in addition to user interaction through the *Viewer*), and through which the user receives feedback about a simulation's properties. It is responsible for producing the types of controls and read-outs necessary to manipulate a simulation. These can include buttons, sliders, checkboxes, combo-boxes, graphs, text fields, and numerical displays.

3.6 Units

3.6.1 Length, Mass, Time, Force

A unit length in *TEALsim* is one meter, a unit mass is one kilogram, a unit time is one second, and a unit force is one Newton.

3.6.2 Charge and Current

A unit charge in *TEALsim* is $\sqrt{\varepsilon_0}$ Coulombs, or 2.975 micro Coulombs. ($\sqrt{\varepsilon_0} = \sqrt{8.85\times 10^{-12}} = 2.975\times 10^{-6}$). This unit charge arises because we set ε_0 to 1 in *teal.config.Teal* (javadocs, java). Since the force between two point charges is given by $qQ/(4\pi\varepsilon_0 r^2)$, for this force to be in Newtons if r is in meters, we must take the unit charge to have this value.

3.6.3 Electric and Magnetic Dipole Moment

A unit electric dipole moment in *TEALsim* is

4 *TEALsim* Examples

We present a number of examples below to illustrate various features of *TEALsim* and how to use them to construct simulations. These examples are contained in the package *tealsim.physics.examples*. As we present the examples we will point out how they implement the overall scheme described above in Section 3.

4.1 Example_01: 3D objects (native and imported)

We begin with an example which demonstrates how much of the usual boilerplate for stetting up a Java3D scene is hidden from the user at this level. Here and in the future the links following a reverence to a java file link to the *javadocs* for that file and the *.java* file itself, respectively. The example *tealsim.physics.examples.Example_01* (javadocs, java), extends *SimEM*, which has already created the lights, cameras, etc. We create two native Java3D objects, a flat red disk and a green sphere, and add them to the scene. We also import two .*3DS* files, an orange hemisphere and a tapered cone with a texture applied.

A screen capture from this example is shown in Figure 4.1-1. The coordinate system in Java3D is as follows: the *x* axis is horizontal and positive to the right; the *y*-axis is vertical and positive upward; the *z*-axis completes the right handed system and is out of the page. The lights and the camera in this example are all set in the viewer for this world, *teal.render.j3d.ViewerJ3D* (javadocs, java), which extends *teal.render.viewer.AbstractViewer3D* (javadocs, java). The camera is placed at (0.,0.,5.) (see Figure 4.1-2). There are four lights in the scene, one omni light and three directional lights. The directions of the directional lights are: (1) (4., -7., -12.); (2) (-6., -2., -1.); (3) (-6., 2., 1.).

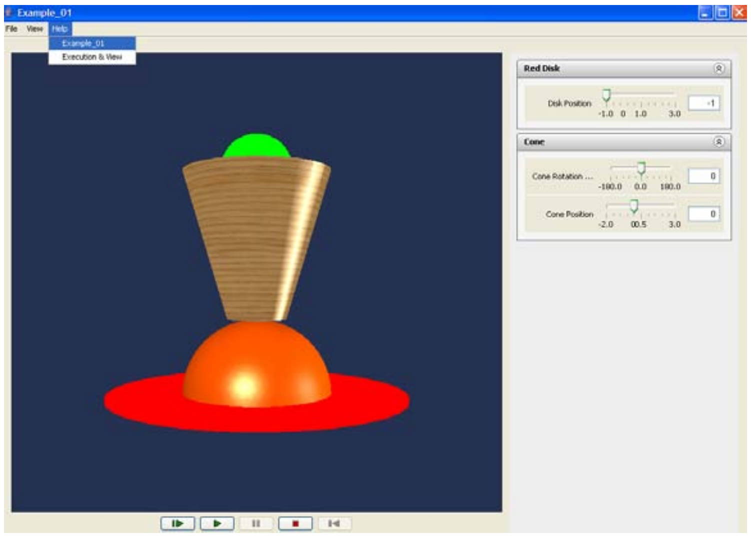


Figure 4.1-1: Screen capture from Example_01

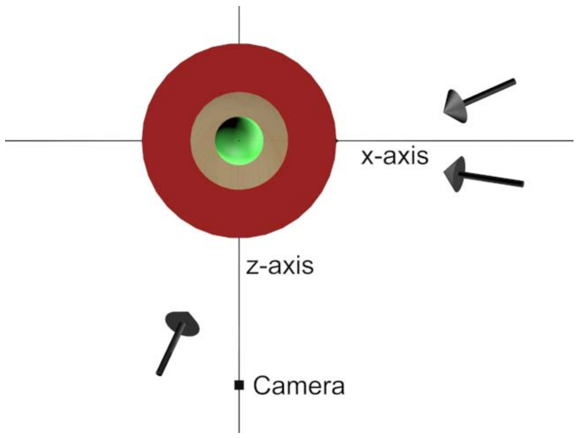


Figure 4.1-2: Top view of scene in Example_01, showing Java3D axes and the directions of the three default directional lights.

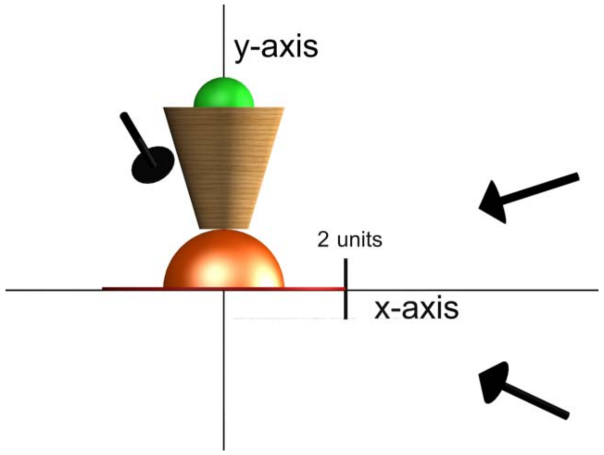


Figure 4.1-3: Front view of scene in Example_01.

We add to the upper tool bar in the simulation (see upper left of Figure 4.1-2) links to two help files, one which explains what the simulation does (*resources.help.example_01.html*) , and one which explains the view and execution controls (r*esources.help.executionView.html*) for the 3D window and simulation engine, respectively. These files are opened using the browser included in the TEAL project, *teal.browser.Browser* (javadocs, java).

In terms of the overview provided in Section 3.4 above, all of our examples extend *teal.physics.em.SimEM* (javadocs, java) which in turn extends *teal.sim.simulation.Simulation3D* (javadocs, java). *Simulations3D* sets the viewer to *teal.render.j3d.ViewerJ3D* (javadocs, java). *ViewerJ3D* extends *teal.render.viewer.AbstractViewer3D* (javadocs, java) which in turn extends *teal.render.viewer.Viewer* (javadocs, java).

The sliders are built out of components in *teal.ui.control.PropertyDouble* (javadocs, java), which extends *teal.ui.control.PropertySlider* (javadocs, java). The vertical (*y*) position of the red disk is controlled by a one slider. The vertical position of the cone is also controlled by a slider, as is its orientation in the *xy* plane (the plane of the screen as seen from the initial camera position).

4.2 Example_02: Simple simulation using *theEngine*

In *Example_01* we only illustrated properties of the Java3D viewer, with no simulation dynamics (that is, nothing happens as world time progresses). In *Example_02* (javadocs, java) which again extends *SimEM* (as does all of our examples), we introduce a simple dynamical situation to illustrate how *theEngine* and engine controls work. The simulation *SimEM* sets the simulation engine to *teal.physics.em.EMEngine* (javadocs, java), which extends *teal.sim.engine.SimEngine* (javadocs, java). In *Example_02* (javadocs, java), we set the dynamical time step to 0.02 seconds using the method *theEngine.setDeltaTime*, and add to the scene a *teal.physics.em.EMObject* (javadocs, java) object, the *teal.physics.em.PointCharge* (javadocs, java) object. This object knows how to interact with the world, as we explain in this example and others to come.

In Example_02, the *PointCharge* object *floatingCharge* has mass, which means it will interact with the general gravitation field of the world, which is set by default to (0.,-9.8,0.) in *EMEngine*. To show how we change the default gravity in the world, we add a statement which reduces the default value of *g* to 4.9 meters per second squared in the negative *y*-direction in this example. We also create a "wall" 1.25 meters beneath the initial position of the charge, which appears in the scene as a transparent rectangle. If we step the dynamics using the step control (see Figure 4.2-1), it takes about 35 steps at the 0.02 second step size in this world for the point charge to fall the 1.25 meters to the position of the wall, which is correct for a gravitational acceleration of 4.9 meters per second squared.

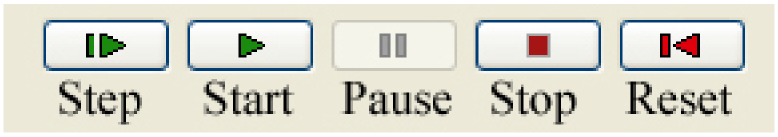


Figure 4.2-1: The simulation control icons, whose properties are as indicated.

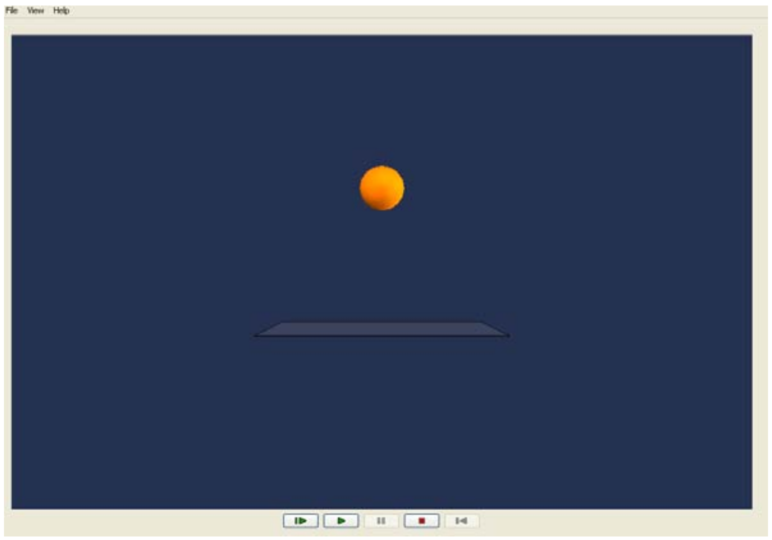


Figure 4.2-2: A charge falling toward the floor.

We added a collision controller to the *floatingCharge* so that it will be recognized as a colliding object when it touches the "Wall". The collision controller *teal.sim.collision.SphereCollisionController* (javadocs, java) is a sphere whose radius is the radius of the *floatingCharge*, placed at the center of the *floatingCharge*, This has the effect that when the *floatingCharge* center position comes within its radius of the "Wall" it "collides" with the "Wall" and the *floatingCharge* bounces. We set the elasticity of the wall to 1.0, which means the collision is perfectly elastic.

The dynamics of this falling charge is computed as follows. First of all, the six equations of motion for the position **x** and velocity **v** of the point charge that we are trying to solve are

$$\frac{d}{dt}\mathbf{x} = \mathbf{v} \qquad \frac{d}{dt}\mathbf{v} = \mathbf{F}/m \tag{4.2.1}$$

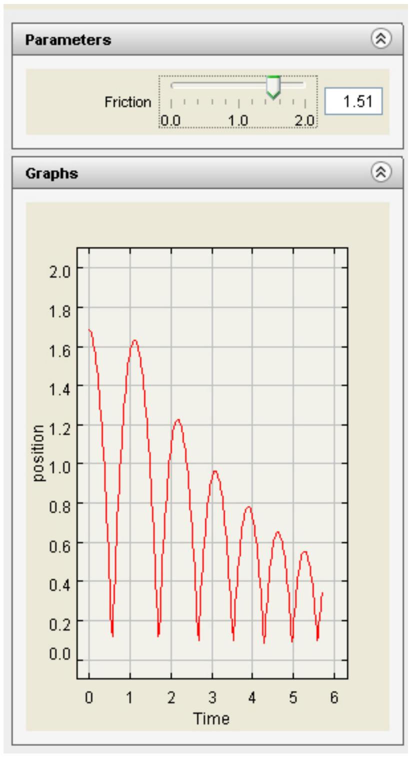
(we ignore the rotational dynamics of the object, although this can be included). Thus the dependent variables that we are integrating are the position and velocity of the floating charge. Our point charge object extends *EMObject*, which extends *teal.physics.physical.PhysicalObject* (javadocs, java), and therefore has a method *getDependentValues()*. This method simply returns the current values of the six dependent values of **x** and **v**. It inherits from *PhysicalObject* a method *getDependentDerivatives()* which calculates the derivatives of these dependent values using equation (4.2.1*).* This method accomplishes this using a metho*d getExternalForce* in *PhysicalObject* which adds up the total force on the point charge, including gravity, an over all frictional force proportional to the velocity (*-friction****v**), any electromagnetic forces *q*(**E**+**v**x**B**) acting on the point charge due to other electromagnetic objects in the world (zero in this example), and a Pauli force which is always repulsive at close distances (we discuss this further in *Example_04* below).

Once we have the current values of the dependent variables and their derivatives, we integrate the equations of motion using a fourth order Runge-Kutta scheme *teal.math.RungeKutta4* (javadocs, java), using routines in *teal.sim.engine.SimEngine* (javadocs, java). In particular, the method *nextStep* in *SimEngine* advances the world one time step using the integration routine, among other methods.

4.3 Example_03: Simple simulation with a graph and a damping slider

Example_03 (javadocs, java) is the same as *Example_02* except that we add a graph of the vertical position of the charge and a slider that controls the amount of damping in the world. To accomplish this we must import user interface classes for the panel display, for the damping control slider, and for plotting.

Note that we can change the friction value dynamically between time steps in the world. That is, in the integration for the position of the point charge, before each time step *nextStep* checks to see if the friction in the world has changed, and computes the next dynamical step using the current value of the friction.



Friction 1.51

Figure 4.3-1: The graph and slider control for the falling charge

4.4 Example_04: Two Interacting Electromagnetic Objects (Point Charges)

In Example_04 (javadocs, java) we create two point charges which interact via their electric fields. The two objects are our point charge from *Example_02*, free to move along the vertical axis, and a new point charge which is fixed in space 0.8 meters below the wall in *Example_02*, again on the vertical axis. We set the friction in the world to a high value, so that the floating charge will quickly settle down quickly to its equilibrium position.

The charge on the fixed-in-space charge can be varied between -10 and 50 charge units, where a charge unit is 2.975 micro Coulombs (see Section 3.6.2). The charge on the floating charge is constant at 1 charge unit, its radius is 0.2 meters and its mass is 0.035 kg. The force of gravity on the floating charge, m**g**, is 0.343 Newtons. For a fixed charge of about 4.4 charge units the upward electrostatic repulsion when the floating charge is resting on the wall (center 1 meter above the fixed-in-space charge) will just about balance the downward force of gravity, and the floating charge will levitate above the wall for values of the charge on the fixed-in-space charge above this value.

Note that if we change *fixedCharge.setMoveable(false)* to *fixedCharge.setMoveable(true)* in *Example_04*, the position and velocity of our fixed charge will be added to the dependent variable list of our world. When we start the simulation we will be integrating 12 dependent variables, 6 for each charge, and the "fixed" charge will begin to fall under gravity etc.

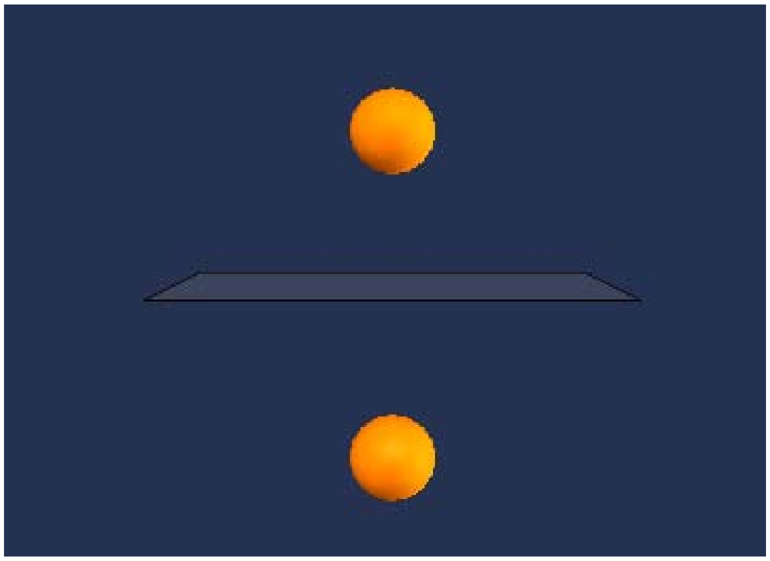


Figure 4.4-1: Two Electromagnetic Objects Interacting

4.5 Example_05: Vector Field Grid

In *Example_05* (javadocs, java) we add a vector field grid to *Example_04* to demonstrate one of three methods we use to visualizing electromagnetic fields.

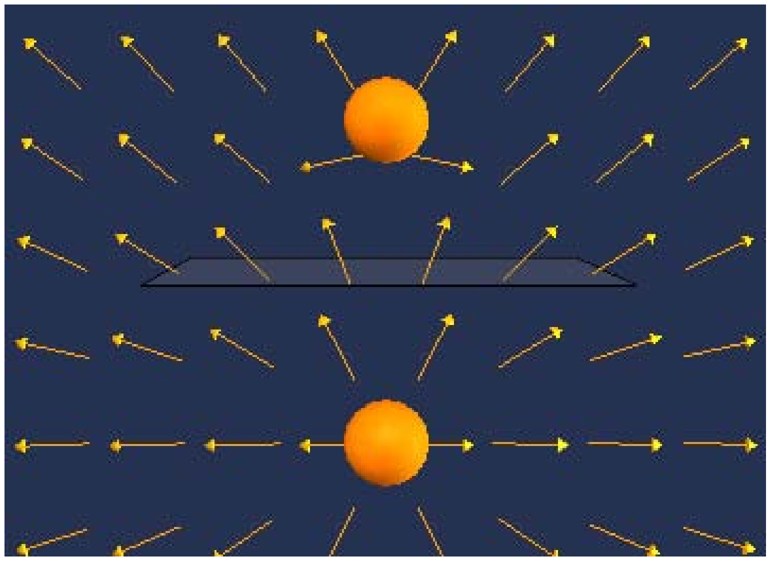


Figure 4.5-1: An example of a vector field grid.

4.6 Example_06: Line Integral Convolution

In *Example_06* (javadocs, java) we add a line integral convolution example to *Example_04* to demonstrate one of three methods we use to visualize electromagnetic fields.

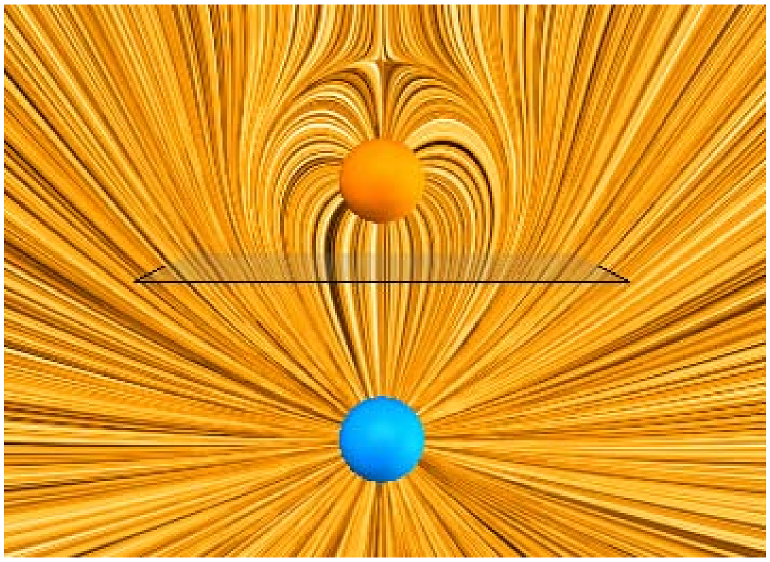


Figure 4.6-1: An example of a line integral convolution.

4.7 Example_07: Traditional Field Lines

In *Example_07* (javadocs, java) we add field lines to *Example_04* to demonstrate one of three methods we use to visualize electromagnetic fields. In this case we add two kinds of field lines.

The first kind (the two red field lines to the right of the vertical center line in Figure 4.7-1) are field lines that always go through the same spatial point, in this case (1,0,0) and (1,2,0). The point at which the field line starts for both of these field lines is shown by a lighter segment along the field line. Note that for the red field lines we construct the field in both directions from the starting point; that is both along the field and opposite the field.

The second kind of field line (the three blue field lines to the left of the vertical center line in the figure) starts out at the radius of the sphere at the same angles (in this case -45, -90, and -135 degrees) measured from the object direction of the object to which it is attached, in this case the floating charge. As the floating charge moves up and down the position at which the blue field lines start is always at the same angle from the axis of the charge direction, moving with the charge.

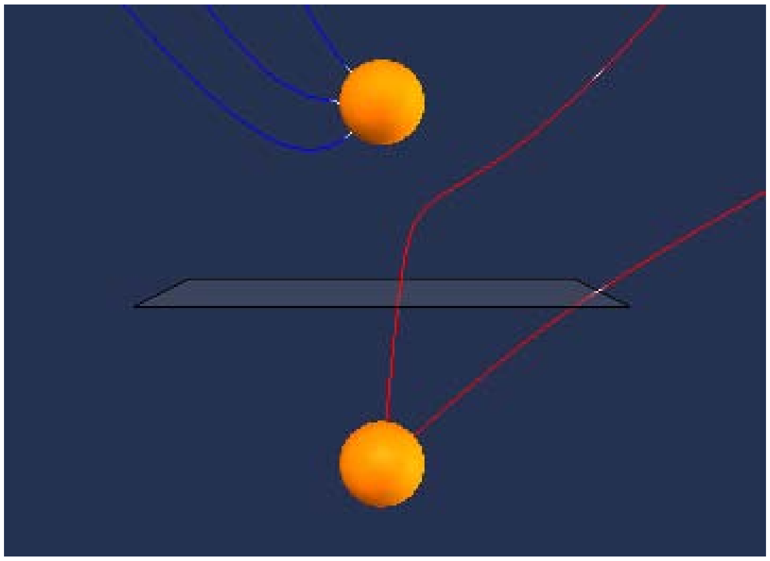


Figure 4.7-1: Examples of two fixed and three relative field lines.

4.8 Example_08: Flux Preserving Field Lines Electric

In *Example_08* (javadocs, java) we add field lines to *Example_04* to demonstrate one of three methods we use to visualize electromagnetic fields. In this case we add field lines which preserve electric flux.

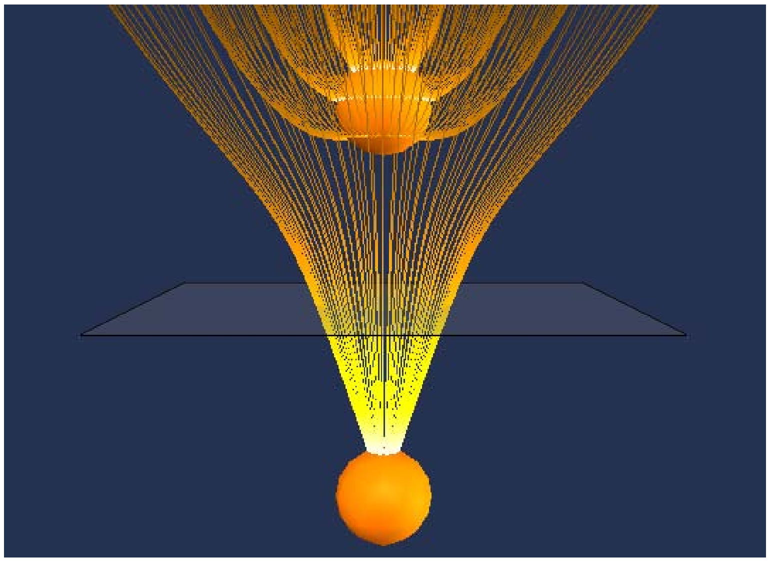
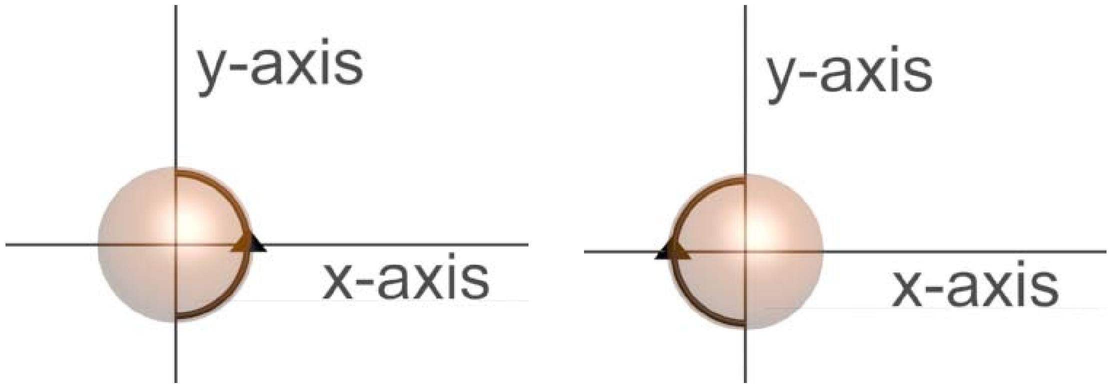


Figure 4.8-1: Flux Preserving Electric Field Lines

Circle search is 0.99*radius of the object from top to bottom on right side.



4.9 Example_09: Flux Preserving Field Lines Magnetic

In *Example_08* (javadocs, java) we add field lines to *Example_04* to demonstrate one of three methods we use to visualize electromagnetic fields. In this case we add field lines which preserve electric flux.

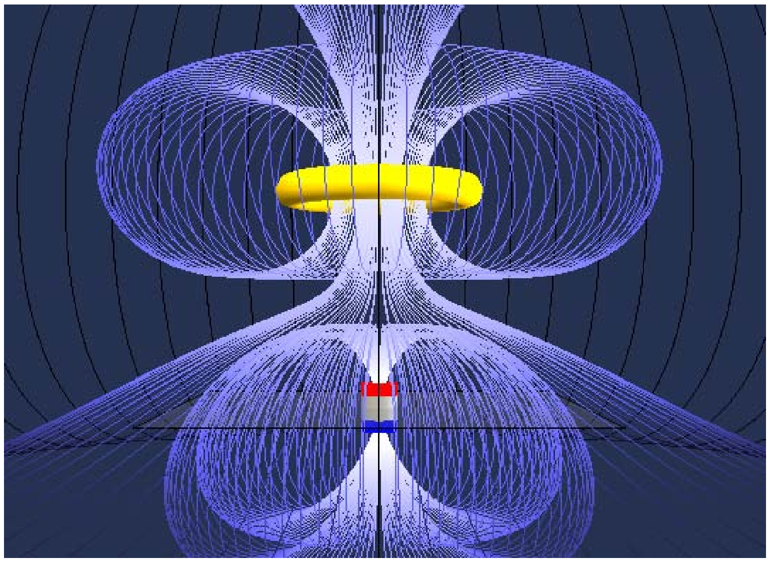


Figure 4.9-1: Flux Preserving Magnetic Field Lines

4.10 Example_10: Constraints

In this tutorial, we put everything together, and add an additional *.3DS* object.

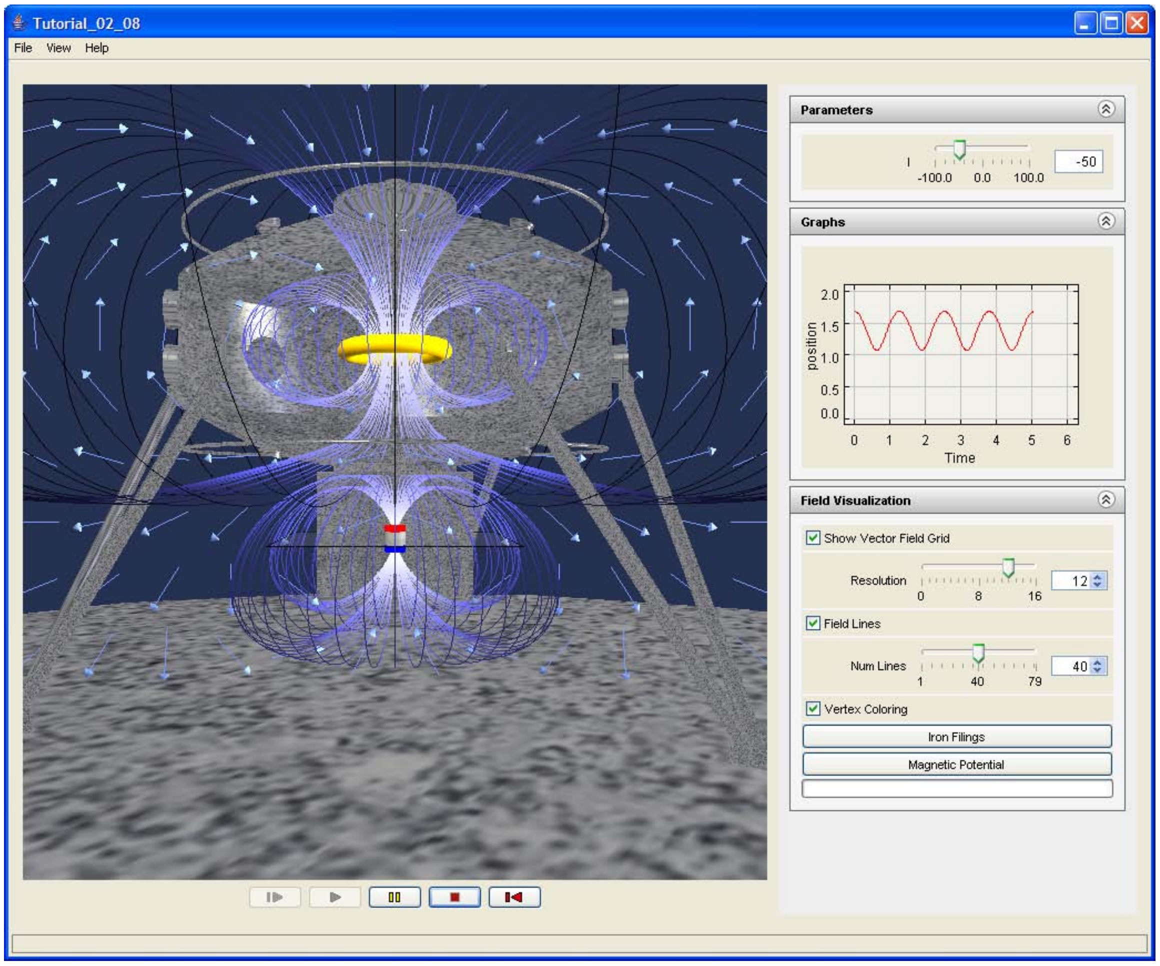


Figure 4.10-1: Screen Capture of Example_08

5 The Structure of the *TEALsim* Project

To learn the structure of the TEALsim project, the user should use this document as well as the *javadocs* and the actual *java* classes for the *TEALsim* Project. We first give an overview of the package structure of the Project. This same information is included in the package documentation in the *javadocs* for the Project.

5.1 java.src.teal .isocket

5.2 java.src.teal

5.3 java.src.teal.app

5.4 java.src.teal.audio

5.5 java.src.teal.browser

5.6 java.src.teal.config

5.7 java.src.teal.core

5.8 java.src.teal.field

5.9 java.src.teal.framework

5.10 java.src.teal.math

5.11 java.src.teal.physics

5.11.1 java.src.teal.physics.em

5.11.2 java.src.teal.physics.mech

5.11.3 java.src.teal.physics.physical

5.12 java.src.teal.plot

5.12.1 java.src.teal.plot.ptolemy

5.13 java.src.teal.render

5.13.1 java.src.teal.render.geometry

5.13.2 java.src.teal.render.j3d

5.13.3 java.src.teal.render.primitives

5.13.4 java.src.teal.render.scene

5.13.5 java.src.teal.render.viewer

5.14 java.src.teal.sim

5.14.1 java.src.teal.sim.behavior

5.14.2 java.src.teal.sim.collision

5.14.3 java.src.teal.sim.constraint

5.14.4 java.src.teal.sim.control

5.14.5 java.src.teal.sim.engine

5.14.6 java.src.teal.sim.function

5.14.7 java.src.teal.sim.properties

5.14.8 java.src.teal.sim.simulation

5.14.9 java.src.teal.sim.spatial

5.15 java.src.teal.ui

5.15.1 java.src.teal.ui.control

5.15.2 java.src.teal.ui.swing

5.16 java.src.teal.util

5.17 java.src.teal.visualization

5.17.1 java.src.teal.visualization.dlic

5.17.2 java.src.teal.visualization.image

5.17.3 java.src.teal.visualization.processing

5.18 java.src.tealsim

5.19 java.src.tealsim.physics

5.19.1 java.src.tealsim.physics.em

5.19.2 java.src.tealsim.physics.examples

This package consists of examples of simulations constructing using the elements of *TEALsim*. The examples cover the various ways of dealing with native and imported 3D objects, the dynamics of simple systems and how they are integrated into the simulation engine, and the user interface controls.

5.19.3 java.src.tealsim.physics.ilab

5.19.4 java.src.tealsim.physics.mech

5.20 resources

5.20.1 resources.help

5.20.2 resources.icons

5.20.3 resources.model3d

5.20.4 resources.models

5.21 java

5.21.1 java.jnlp

5.21.2 java.lib

5.22 javadoc

5.23 release

5.24 TEALsimDoc

5.25 tools

5.26 www

5.27 xdocs

The doc folder contains the *javadocs* for the Project, in *html* format. The main *html* link is the *index.html* link in *doc*.

5.28 resources

6 *TEALsim* Applications in 802TEAL3D

The web page http://web.mit.edu/8.02t/www/802TEAL3D/ has a number of TEALsim applications, and we discuss each of these in turn.

6.1 Vector Fields

6.1.1 Field Mapping Application

6.2 Electrostatics

6.2.1 Charging a Van de Graff Generator

6.2.2 Charges Interacting inside a Pentagon-shaped Box

6.2.3 The Capacitor

6.2.4 The Electrostatic Force Experiment

6.2.5 The Electrostatic Zoo

6.2.6 The Electrostatic Videogame

6.2.7 Two Point Charges

6.2.8 Charging by Induction

6.2.9 The Charged Metal Slab

6.2.10 Torque on an Electric Dipole in a Constant Field

6.3 Magnetostatics

6.3.1 Two Current Carrying Rings

6.3.2 The Floating Coil

6.3.3 Torque on a Magnetic Dipole in a Constant Field

6.3.4 The TeachSpin(tm) Applet

6.3.5 The Magnetic Field of a Wire and Compass

6.4 Faraday's Law

6.4.1 The Falling Coil

6.4.2 Faraday's Law

6.4.3 Faraday's Law, Part 2

6.4.4 The Falling Magnet

6.5 Light

6.5.1 Generating Plane Wave Radiation

7 A Brief Introduction to Java3D with Tutorials

7.1 Java3D

7.2 Java3D Tutorials

7.2.1 Tutorial_03_01

8 *TEALsim* Overview in the Context of Example_08

8.1 The Simulation (*SimWorld, TSimulation*) :

We now examine each of the broad categories discussed above in the context of Example_08. This tutorial extends *SimWorld.* In the tutorial we have a ring of current (the yellow torus in Figure 4-1) interacting with the magnetic field of a fixed magnetic dipole (the red/white/blue cylinder in Figure 4-1). The ring of current is assumed to have negligible inductance and is acted upon by gravity in addition to the field of the dipole.

8.2 SimPlayer (*TFramework*) and SimPlayerApp

8.3 SimEngine – the Simulation Engine:

Physically in Example_08 we have a ring of current interacting with the magnetic field of a fixed magnetic dipole. The ring of current is assumed to have negligible inductance and is acted upon by gravity. Computing the force on the ring due to the magnetic dipole is straightforward.

8.4 Viewer and Viewer3D – the Rendering Engine:

8.5 The User Interface:

The current in the ring is set by the user using the slider on the upper right as shown in Figure 4-1. Figure 4-2 shows the controls at the bottom of the view window that allow the user to start, stop, pause, and reset the simulation, as well as step through the simulation one time step at a time. Once the user presses run, the simulation runs until the user presses pause, stop, or reset. The user can also simply press step, which will advance the simulation one time step. The time step is a basic property of the *SimEngine*, and is set in this tutorial in line `086 theEngine.setDeltaTime(0.02).`

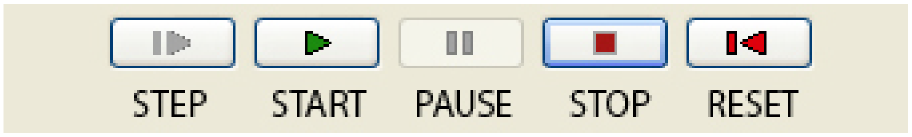


Figure 8.5-1: Controls for Running Tutorial _02_08

When the simulation begins, the ring of current is about 1.7 meters above the magnetic dipole, which is located at zero, and has a current of -50 amps (positive current is counterclockwise when viewed from above). As the simulation runs, the ring of current oscillates up and down with an average height of around 1.3 meters. The position of the ring as a function of time is plotted in the graph on the right. When the user hits reset, the ring is returned to its initial position but the current remains at the last value it was set to using the slider by the user.

At any time as the simulation runs, the user can change the current in the ring by using the slider or by entering the desired current in the box next to the slider **and hitting enter.** The various parts of the UI control appear in separate panels on the right. Controls for the various visualization methods for the magnetic field are contained in the panel entitled "*Field Visualization*". The user can hide the field lines if desired, change their number and color weighting, and change the number of nodes in the vector field grid. The user can also have the simulation stop at any point and calculate a line integral convolution representation of the field by left-clicking on the "*Iron Filings*" button; left-clicking on the “*Magnetic Potential*” button displays a field that is everywhere perpendicular to the magnetic field. The simulation is paused as this calculation is being carried out. To restart the simulation, the user simply left-clicks on run again.

9 MIT TEALsim Software License

THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

10 Appendices

10.1 Creating 3D Objects for Import Into Java3D

10.1.1 Creating Native *Java3D* Objects

10.1.2 Creating and Importing *Autodesk 3ds Max 8 .3DS* Scene Files

The scale conversion from 3ds max to java 3D is as follows. The height of the cone imported below in the max 3ds file is 200 inches. This is for "Customize/Units Setup" US Standard Decimal Inches and for "Customize/Units Setup/System Unit Setup" 1 unit = 1 inch in Autodesk 3ds Max 8. The conversion between max units and Java3D units is under these circumstances 1 Java3D unit = 1 Max inch. Thus when we scale the cone by a factor of 0.01 it has a height of 2 Java units

10.1.3 Importing *Wavefront .obj* Scene Files

The scale conversion from the .obj file to java 3D is as follows: The height of the box imported below in the .obj file is 100 obj units. We have scaled it by a factor of two, and it then appears in Java 3D as 1 unit high. Thus the conversion is 200 obj units = 1 Java 3D unit, or 1 obj unit = .005 Java 3D unit.

10.1.4 Importing *VRML .wrl* Scene Files

10.1.5 Importing *Lightwave* 3D Scene Files

More memory VM -Xmx512m

10.2 A Guide to Programming Resources for Java 3D

10.2.1 The Java Language

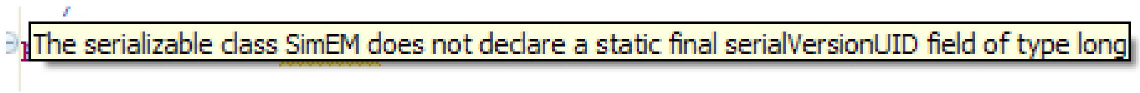
10.2.2 Java 3D

TEALsim is based on Java 3D (http://java.sun.com/products/java-media/3D/) .

This document is intended to be used in conjunction with the browser-accessible, javadoc-generated reference for the code base.

11 Questions for Phil Bailey

11.1 What is a serializable class and what does the following error mean (from public class SimEM extends Simulation3D {)?



```
The serializable class SimEM does not declare a static final serialVersionUID field of type long
```

11.2 Setter IntrospectionEx: Method not found: isValue tealsim.physics.examples.Example_01

11.3 What do I do it I want to access the direction of directional lights from a tealsim simulation?

11.4 Check with Phil to make sure that the path I put for the icons in teal.sim.engine.EngineControl will work when it is put up on the web.

11.5 Why don't we use the static light directions in teal.render.viewer.AbstractViewer3D when we are setting up the lights in ViewerJ3D.setDefaultLights().

11.6 How would I change the abstractviewer3D so I could change the light direction1?

11.7 What's with the version number, how should I change?

11.8 Can I change the levels of TDebug to printout particular things, e.g. the stepping of the dynamics, etc.

12 References

References

B. Cabral and C. Leedom, *Imaging Vector Fields Using Line Integral Convolution*, Proc. SIGGRAPH ’93, pp. 263-270, 1993. 13

J. Dori and J. Belcher, *How does technology-enabled active learning affect student’s understanding of electromagnetism concepts*, Journal of the Learning Sciences 14 (2), 2004). 6

13 Index