Virtual 3D World for Physics Experiments in Higher Education

Implementation of Physics Experiments in a Virtual World

Author:
Stefan Berger,
Graz University of Technology

January 10, 2011

Supervisor:
Univ.-Ass. Dr. Christian Gütl,
Graz University of Technology

Co-Supervisor:
Associate Director V. Judson Harvard,
Massachusetts Institute of Technology

Massachusetts
Institute of
Technology

Abstract

E-learning software has the ability to enhance students performance significantly. The improvements in hard- and software over the last decade enables developers to implement extendible real-time simulations to increase the learning effect. Not only e-learning software does benefit from the advancement of technology but also virtual worlds. Especially high-speed internet which is broadly available nowadays helps to build virtual 3D worlds with complex functionalities.

Over the last view years e-learning research has discovered the potential of virtual worlds for its purpose. Within the teaching field it has several advantages over conventional "real-world" teaching. Stundents can connect to a teaching session from all over the world and use the provided virtual world functionality over long time periods. Time slots for groups of students can be used if a class with many students uses e-learning software within a virtual world.

MIT's TealSim physics simulation software aims to improve the students performance of well-visited courses as well. It does so by providing a wide range of simulations the students can study prior to the classes. Teachers can adopt the software by defining simulations specifically for their courses without sophisticated software development skills. To combine the advantages of the TealSim software and virtual worlds this work focuses on putting TealSim into such a 3D virtual world. This creates a collaborative learning environment with the potential to improve the students performance and to lower the teaching effort for teaching personal. During the implementation process the TealSim software was also improved by the means of software design to fit into the Open Wonderland virtual world.

Contents

List of Figures

1 Introduction

Virtual worlds have been evolving for about twenty years now. Within such worlds many people represented by their in-world avatars can work together or meet for other purposes. One of the first applications using this technology were multi-player games as Doom (1993) or Quake (1996). When the computational power as well as the connection speed between the peers increased more complicated and realistic applications as Second Life[1] (launched in 2003) were feasible [9]. Due to this improvements in technology web-based e-learning systems are becoming an interesting field in computer education [16].

In [12] a prove of concept showed that MIT's internet-accessible physics experiments (iLabs) can be integrated within a virtual 3D world. In order to visualize a 3D model of the chosen experiment in-world parts of MIT's TealSim physics e-learning software where used. With the proof of concept the idea was born to enable students to use the TealSim software, yet a pure desktop application, within a 3D virtual world. As target environment Open Wonderland[2] version 0.5 was chosen. When executed within a virtual 3D world the e-learning software can be used as a multi-user online learning environment. Such a collaborative online learning system can increase the learning performance of students significantly compared to conventional collaborative learning [10]. It covers a wide field of application and can be used for preparation to a lecture as well as in the lecture. In the long term an online learning environment enables students to attend a course regardless of the students location [7].

This work mainly describes the way the TealSim physics simulation software and the Open Wonderland 3D virtual world are joined together in order to create a collaborative learning environment. The next section explains the two frameworks from the users and the programmers perspective. Section 3 describes the actual implementation in detail. Subsequently, section 4 explains the resulting software from a users point of view. This covers the installation of the module as well as the usage. Finally, section 5 quickly reviews the outcome of this work and points out some future work suggestions.

[1] `http://secondlife.com/`

[2] `http://openwonderland.org/`

2 Frameworks

Two frameworks were used for this work. Both of them will be explained in this sections from the user perspective and the developers perspective.

2.1 Open Wonderland

Open Wonderland is an open source virtual 3D world entirely written in Java programming language [15]. It provides a set of functionalities for different types of users:

- Content developers can build up a 3D world with a vast variety of tools. 3D models can be added by simply dragging them into the world[13]. Cells with many different functionalities can be added to the world.
- Users can explore a world and communicate to other users. Their avatars can be customized to a high extend giving every user a unique appearance. The user interface to the elements placed in-world are intuitive and easy to use.
- Open Wonderland server administrators can enable applications which are runnable on the server to be shown in-world. They can save snapshots of worlds to be restored later. A variety of extensions, so-called "modules" to be found in the module warehouse[3] can be added to extend the functionality of the Open Wonderland server. The administrator can also enable security features to restrict access to the administration page as well as user log ons.
- Software developers can add new functionality by implementing modules. Open Wonderland provides an API which makes it fairly easy to develop a module. Extensions can be added almost arbitrarily in terms of what parts of Wonderland should be extended. However, most modules will add visible models and/or functionality to such models.

Open Wonderland is designed as a client-server architecture. A user can log into a server without installing any additional software. Servers can be set up on many platforms.

2.1.1 Software Architecture of Open Wonderland

Open Wonderland consists of several components. In figure 1 an overview of this components is given. On the server side a Glassfish application server[4]

[3] `http://openwonderland.org/module-warehouse/module-warehouse`

[4] `http://glassfish.java.net/`

is responsible for managing the other three server side services. The client can start, stop or change parameters of the services using the web interface provided by Glassfish. The shared application server enables Wonderland to

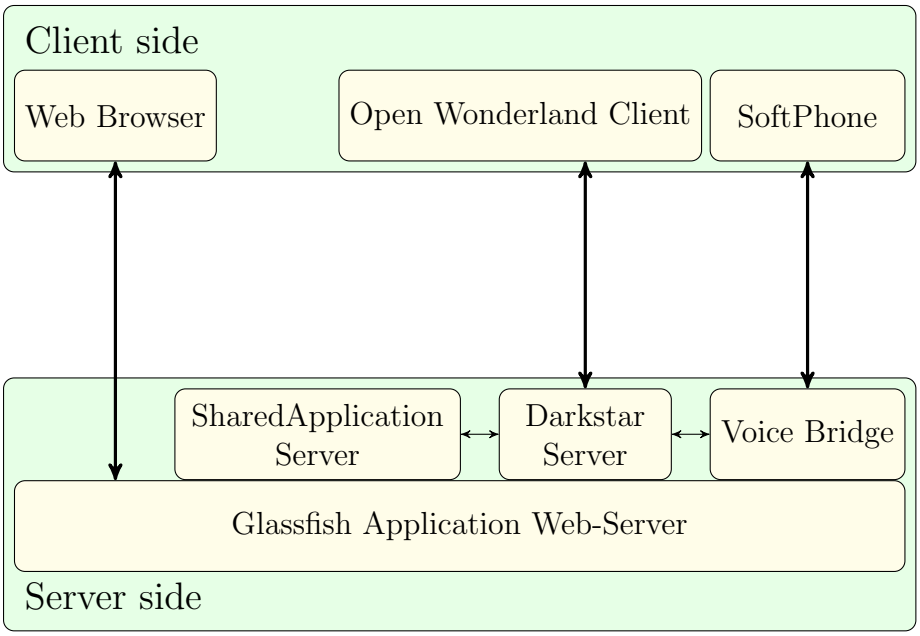

Figure 1: Client and Server Components of Open Wonderland

show applications started on the server within the virtual 3D world. Such applications can then be rendered within the virtual world on a two-dimensional pane. For the voice bridge is a jVoiceBridge[5] audio mixer is used. It supports high-fidelity stereo sound and is used for sharing the voice among the users. It also supports connections to remote soft phones.

The Darkstar server represents the heart of Wonderland which runs on top of that server. Project Darkstar is a high-performance game server with high scalability. It provides a communication interface to exchange messages to the clients as well as an API for the application running on a server. The Wonderland application running on the Darkstar server makes use of the shared application server and the voice bridge. The Open Wonderland client is a Java application which establishes a connection to the server during a log-in process. For the 3D graphics the JMonkeyEngine[6] library is used. This game engine lacks multi-threading support. For that reason MTGame runs on top of it to add the required functionality.

[5] `http://java.net/projects/jvoicebridge/`
[6] `http://www.jmonkeyengine.com/`

One of the most powerful features of Open Wonderland is its extendibility. To extend the functionality so-called "Modules" are used. They can be managed using the Glassfish web interface. Many built-in features Wonderland comes with are already implemented as modules[14]. In order to be noticed by Open Wonderland a module has to fulfill requirements as implementing Wonderland interfaces. It can contain server-side code, client-side code and artwork which is mostly 3D models or texture images.

2.2 TealSim

TealSim is a part of MIT's Technology-Enabled Active Learning (TEAL[7]) project. This project was launched in 1994 [1] and defines a learning structure for courses with larger numbers of students. The project aims to improve the students understanding by using simulation and visualization software[4]. TealSim is such a simulation software. It provides a simple interface for defining new simulations. This enables even unexperienced Java programmers to

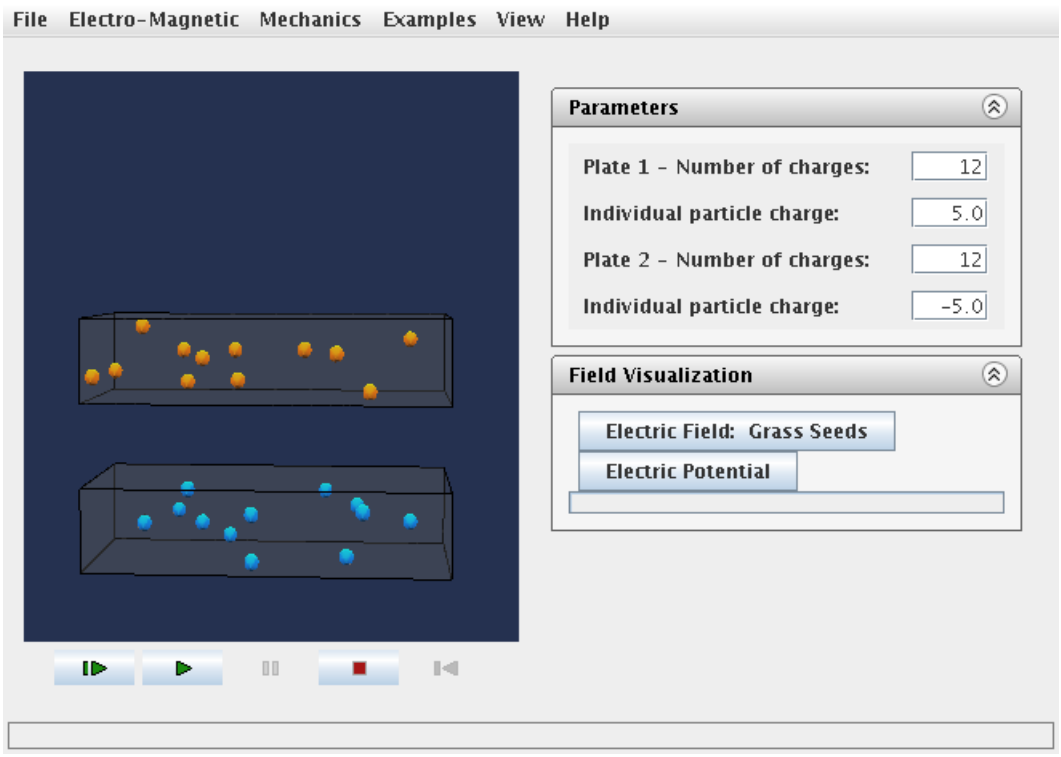

Figure 2: TealSim Screenshot

implement simulations with extended features like the use of external 3D

[7] http://web.mit.edu/8.02t/www/802TEAL3D/

models. On the other hand TealSim is extendible for programmers if new functionality of physical objects are needed. In figure 2 the user interface is shown. A user can choose a simulation in the menu bar and change simulation parameters on the right-hand side. A simulation can be started by pressing the "play" bottom at the bottom. Parameters can be changed during the simulation. Within the frame on the left-hand side the 3D spatial representation of the simulation is shown. With most simulations the user can change the view with a drag on the shown 3D model. Such mouse interaction behaviors can be defined when writing a simulation. TealSim also provides analysis features as showing fields in different representations (e.g. grass seeds). Some of those features as e.g. field lines are calculated in real-time while the simulation is running. With showing such elements during the whole simulation execution a better understanding of the physics behind the simulation can be achieved.

2.2.1 Software Design

TealSim is implemented following the Model-View-Controller (MVC) design pattern. With this pattern the software is split logically into three components[8] which are:

- The *model* representing the visual elements as descriptive data. It also defines the behavior of the elements.
- The *view* is responsible for rendering the elements and is usually directly part of the user interface.
- The *controller* is the glue between the components. The user input goes through the controller which then reacts properly by updating model and view.

TealSim consists of several components. The *simulation* makes the model within the MVC pattern. Whenever a simulation is implemented this element is the only object to be defined. It creates all the simulation elements; e.g. the 3D elements, the user interface and the type of the simulation engine needed.

The *simulation engine* is responsible for all the physics calculations. It knows about all the objects within the simulation and can retrieve their physical parameters. Currently the basic simulation engine and an electromagnetic simulation engine is provided. However, new engines with additional capabilities can easily be added by TealSim developers. This can be necessary when new simulations with new physical behaviors are used.

A simulation is loaded by the *SimPlayer*. It creates all the components including the simulation engine and the simulation itself and also represents the user interface. User interface elements defined within the simulation are added to the Java swing GUI and the components are connected to each other if necessary. Within the MVC pattern the simulation makes the controller.

The last element to mention is the *viewer*. It displays the 3D simulation elements on a canvas using the Java3D[8] graphics library. In the course of this work and additional library support was added (see section 3.1). Another responsibility of the viewer is to report user interactions on the 3D elements of the currently loaded simulation.

[8] `http://j3d.org/`

3 Implementation of the required Components

This section describes the implementation details of this work. In order to understand this section properly some software programming knowledge is required. Some basics in Java programming language and in software architecture are recommended.

In order to have as many simulation running within Open Wonderland the TealSim framework is ported run within Wonderland. When this goal is achieved the resulting software can be used for a variety of fields within physics. From the perspective of software development the requirements can be defined as follows:

- As many TealSim simulations as possible should be ported to Open Wonderland. This should be done with little special treatment for single simulations. Most of the features of the simulations should be supported.
- The changes in the TealSim software should not make the definition of a new experiment more complicated. The way how a simulation is defined should basically remain the same as before the changes.
- Neither the software design nor it's performance should be influenced in a negative way. Possible improvements to this measures should be implemented.
- The software must be easily installable on every Open Wonderland server. Thus the software should be packed into a Wonderland module which can be uploaded by the web interface as shown in 4.
- TealSim should still be runnable as desktop version after the changes.
- The advantage of having TealSim run in a virtual world should be pointed out by defining a new simulation.

To meet those requirements, several implementation steps were necessary. Those are described in the next subsections.

3.1 Porting TealSim's 3D Output

As described in 2.2.1, TealSim uses Java3D for the three dimensional graphics output. Open Wonderland uses JMonkeyEngine with MTGame on top. The latter was developed for Wonderland but can also be used by other software outside of Wonderland. The concepts in Java3D are quite different from the

ones in a JMonkeyEngine/MTGame application. The scene graph is built up with different objects. Because TealSim uses abstractions of scene graph elements, those abstractions can be defined for JMonkeyEngine/MTGame in the same way as they are defined for Java3D.

3.1.1 Preparing TealSim for JMonkeyEngine/MTGame

To be able to use JMonkeyEngine and MTGame with the desktop version of TealSim those libraries have to be provided to the code. For that purpose MTGame has to be compiled and packed into a jar file. The code can be obtained via subversion[9] from the MTGame repository[10]. There are several libraries MTGame depends on:

- A slightly adopted version of JMonkeyEngine,
- the physics library JBullet[11],
- the JOGL[12] OpenGL front-end for Java,
- and the Java real-time library javolution[13].

JMonkeyEngine does not only work with JOGL, but also with the lwjgl OpenGL library. For the TealSim desktop version JOGL was used. In order to avoid problems when TealSim is used in Open Wonderland the right JMonkeyEngine jar libraries should be used. Those are exactly the same as used in Open Wonderland. For compiling the MTGame jar file some tweaking of the ant file `build.xml` was necessary. The resulting jar file `mtgame.jar` has to be put into the classpath. Additional classpath directories can also be specified with the `-cp` option on the `javac` and the `java` command. Alternatively when using a development software the classpath can often be set up within the options.

Since JOGL uses OpenGL calls which are not possible in Java, it uses Java's native interface. This way native code can be called by Java code. JOGL comes with four native libraries, namely `libgluegen-rt`, `libjogl_awt`, `libjogl_cg` and `libjogl`. On Linux those libraries have the file extensions `.so` ("shared object"). On Windows platforms `.dll`'s are used. To run the Java virtual machine, the `-Djava.library.path` switch on the command line has to be set to the directory where the native libraries can be found.

[9] `http://subversion.tigris.org/`
[10] `http://openwonderland-mtgame.googlecode.com/svn/trunk`
[11] `http://jbullet.advel.cz/`
[12] `http://kenai.com/projects/jogl/`
[13] `http://javolution.org/`

Many development environments allow the user to specify that path in the project properties as well.

3.1.2 Keeping Java3D Output

For the new desktop version of TealSim it would be nice if it worked with both, Java3D and JMonkeyEngine/MTGame graphics system. This would also help to reduce dirty code. All hard-coded Java3D specific code has to be removed from outside the low-level Java3D package, which is *teal.render.j3d*. All other packages have to use the abstractions in order to work with both graphic engines.

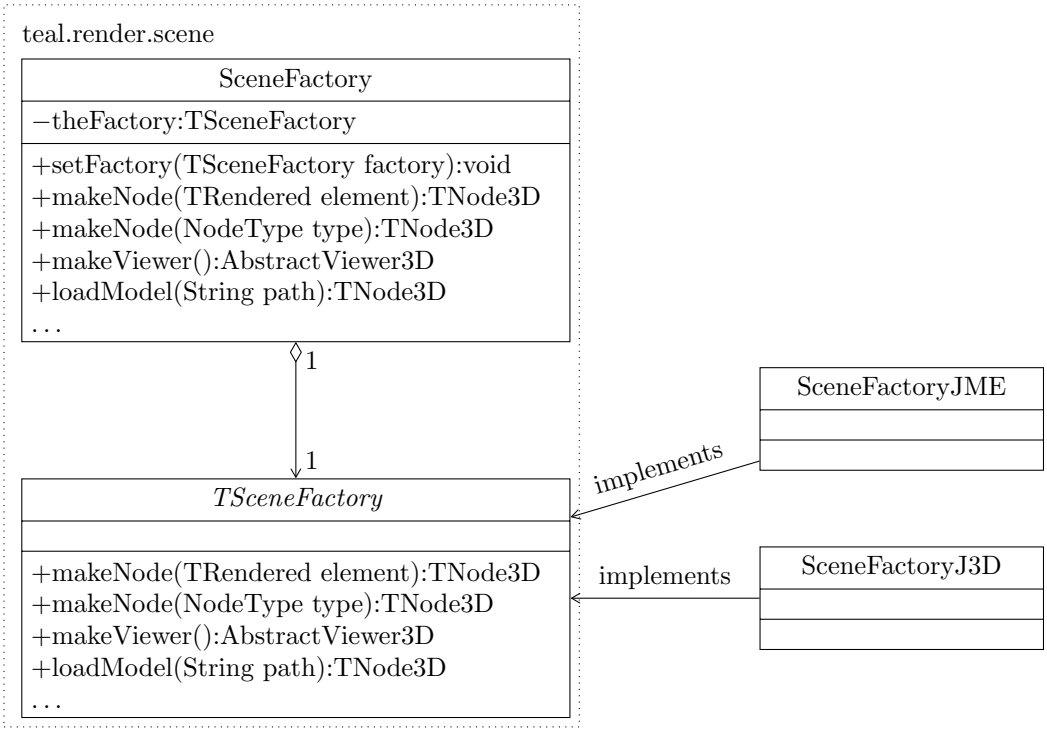

Figure 3: Scene Factory Class Diagram

To instantiate the needed version of the primitive a factory following the factory method design pattern as described in [5] was introduced. In figure 3 a simplified class diagram of the factory and its components is shown. The general *teal.render.scene.SceneFactory* class is ***static*** and contains a reference to an object implementing the interface *TSceneFactory*. This forces the referenced classes to override all the methods declared in this interface. For every graphic output system one SceneFactory implementing *TSceneFactory* has to be implemented.

The *TNode3D* interface is the generic type for an actual scene graph node. An implementing class is a specific node that maps the calls of the *TNode3D* interface to the actual scene graph node class of the specific library (in our case JMonkeyEngine or Java3D). Previously a scene graph node was created directly:

```
TNode3D boxNode = new teal.render.j3d.BoxNode();
```

Now the code is replaced by the one using the factory:

```
TNode3D boxNode = SceneFactory.makeNode(NodeType.BOX);
```

This way it is possible to make the code independent of a specific graphic library. The *makeNode* method simply calls the same method on the concrete Factory:

```
public static TNode3D makeNode(NodeType type){
  return theFactory.makeNode(type);
}
```

The concrete factory can contain all the code for creating a library-specific scene graph object implementing the *TNode3D* interface.

According to previous TealSim developers interfaces as *TNode3D* were introduced because more than one graphic output system was intended to be used in the past. Therefore most of the code outside the specific packages (*teal.render.j3d* and *teal.render.jme*) already use *TNode3D* instead of the Java3D-specific classes. However, many classes are using type-casts to Java3D scene graph classes or created such classes by themselves. This code had to be replaced by the factory-using code. This prevents from using Java3D specific code anywhere else than in the specific packages and thus cleans up the implementation.

As shown in figure 3, the factory defines a *setFactory* method. This method can be used to pass an implementation of *TSceneFactory*. This feature will be important when it comes to the implementation of an Open Wonderland module (see section 3.3.2).

3.1.3 Implementing JMonkeyEngine Primitives

In this section the already available Java3D source code is analysed first. Subsequently, the implementation of the JMonkeyEngine primitives which have to build 3D objects looking exactly as their Java3D equivalents is described.

The graphic library specific scene graph objects have to be implemented to be returned by the factory described in section 3.1.2. In the package *teal.render.j3d* primitives as *BoxNode*, *FieldLineNode* or *ImageNode* were

already defined. The most important class is the one all other primitives are derived from, called *Node3D*. It implements the *TNode3D* interface and is derived from the Java3D *BranchGroup*. It contains several methods to change scale, rotation, transform, visibility and so on. Figure 4 shows the components of *Node3D*.

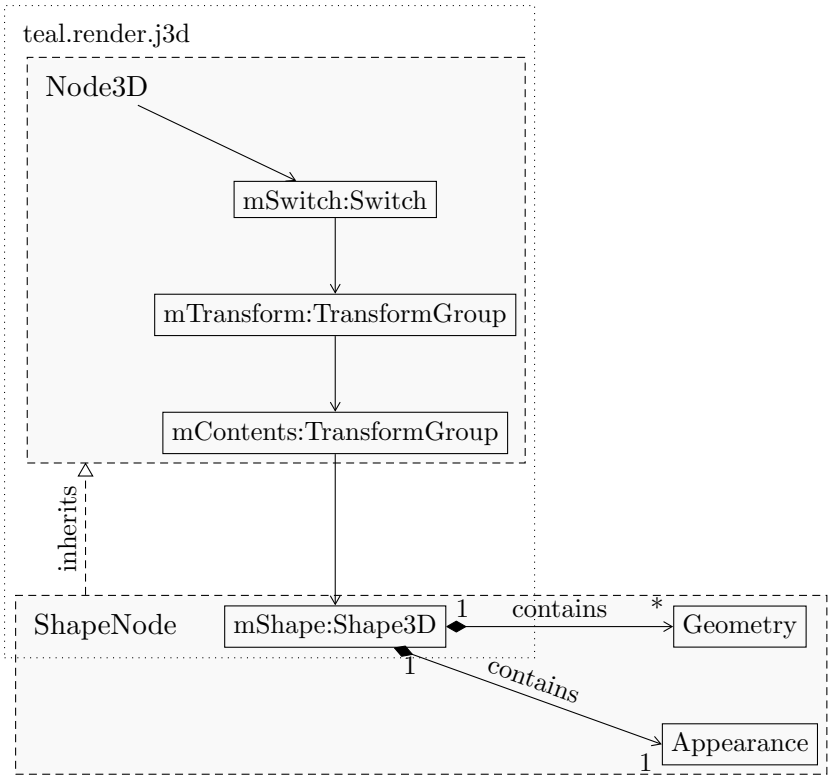

Figure 4: Java3D Scene Graph with Class hierarchy

The class holds references to three descendent scene graph nodes. The *mSwitch* node is added directly as a child of the *Node3D* object. It is used for a change of the node's visibility. The *mTransform* node is attached to the *mSwitch* node. It is responsible most of the transforms. E.g. whenever the *setScale* method is called, the scale is applied to that node. However, on some nodes two different transform operations are applied. Among others the *setModelOffsetTransform* method transforms the *mContents* node. Whenever an instance of *Node3D* is created the three nodes are added accordingly.

The leaves of the scene graph are added by inheriting from *Node3D*. The most generic subclass is *ShapeNode*. It contains a member *mShape* containing the leaf of the scene graph. In Java3D such leafs usually hold the

geometry with all the vertices as well as appearance data. In the appearance colors, transparency and so on are defined. With Java3D one single geometry object can be shared among many different scene graph leaves. This is not possible with JMonkeyEngine because the relation of leaf node to geometry is an *"is a"* as opposed to a *"has a"* relation in Java3D.

Another main difference with the scene graphs of the two graphic libraries is how the transforms are applied. In Java3D *TransformGroup* objects are used to transform the graph with all the descending scene graph objects. In JMonkeyEngine scale, translation and rotation can be applied to all the scene graph objects including the leaves. Thus, the scene graph can be kept flatter. In our case the JMonkeyEngine's *Node3D* class does not contain a reference to any descendent nodes. The class is derived from a JMonkeyEngine *Node* class which is the most abstract version of an inner scene graph node. Visibility and transforms can directly be applied to that node at the same time. Therefore it takes the role of the *mSwitch* and the *mTransform* node in the Java3D version.

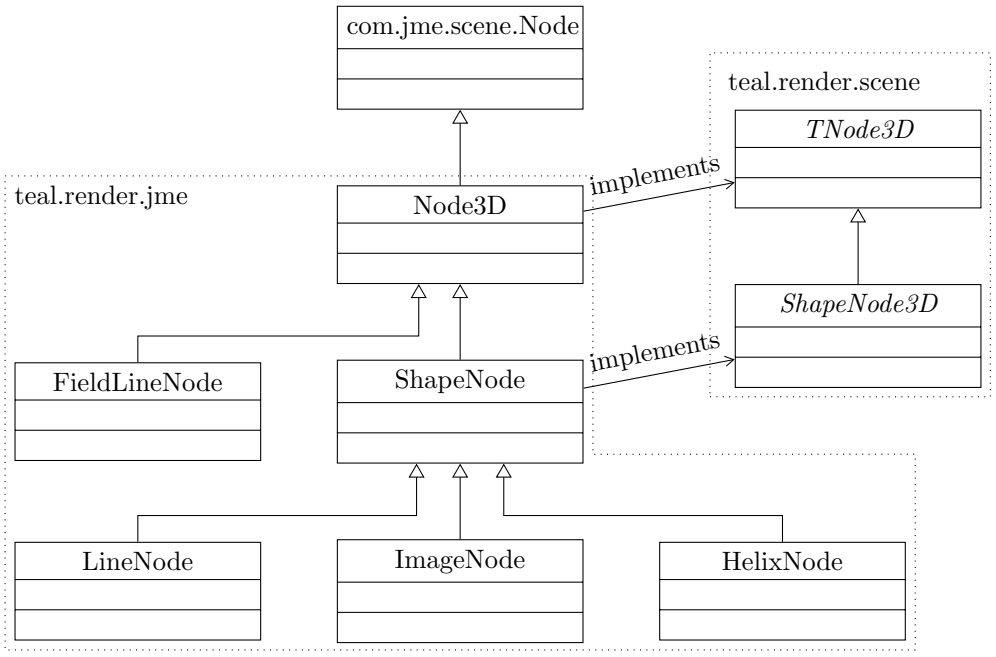

Figure 5: Node Class hierarchy in JMonkeyEngine Part (incomplete)

As with Java3D's *Node3D*, the class is extended by inheritance. A part of the class hierarchy is shown in figure 5. Because the class hierarchy is as shallow as the scene graph hierarchy the SceneFactory (see section 3.1.2) can use it as an ordinary JMonkeyEngine *Node* class:

```
TNode3D sphereNode = new teal.render.jme.Node3D();
((com.jme.scene.Node)sphereNode).attachChild(new Sphere());
```

This way classes as the *SphereNode* in the *teal.render.j3d* package do not need to be defined for JMonkeyEngine. Although many of the 3D objects could be created within the SceneFactory, most of them are put into their own class. One reason for that is the better code structure with this approach. If more code is put into the *SceneFactoryJME* class it becomes larger and is not as readable any more. The code is split by shape type as well.

Another reason why separate classes are defined for many shapes is that the *TNode3D* interface is not powerful enough with some of the 3D-objects. Thus, some more interfaces have to be defined for some of the primitives. This gives the code outside the low-level Java3D or JMonkeyEngine packages the ability to change some other shape-specific parameters. Figure 5 shows the additional *ShapeNode3D* interface which is implemented by *ShapeNode* and therefore by all derived classes. Previously the objects were sometimes casted directly to the actual Java3D class. With a second 3D output library this is not possible any more so some additional interfaces as *TFieldLineNode* or *TArrayNode* were introduced for that reason as well. Such interfaces are only implemented by two classes, namely a JMonkeyEngine and a Java3D specific class. Basically, the code should work without such additional sub-interfaces. For ideas how this could work see section 5.1. In the following paragraphs some of the nodes requireing special treatment are described.

FieldLineNode

The *FieldLineNode* class is the 3D representation of a field line. In TealSim field lines are always symmetric around some axis (mostly the y axis). Because of this characteristic one single line can be referenced by many others. Figure 6 shows a simulation with only 3 field lines, but each of them is shown 25 times.

The main advantage of referencing a line is the lower memory consumption and the higher performance. The vertices of the line are stored and the referencing nodes change only their rotation around the rotation axis. Without making use of this feature the computationally intensive drawing of the field lines would take too much time.

Both Java3D and JMonkeyEngine contain a mechanism for sharing nodes. With Java3D the prototype is a *SharedGroup* node. With a *Link* node this *SharedGroup* is referred. The *Link* object is then put into the scene graph. There can be many links to one shared group. In JMonkeyEngine there is a *SharedNode* derived from *Node* as well as a *SharedMesh* derived from

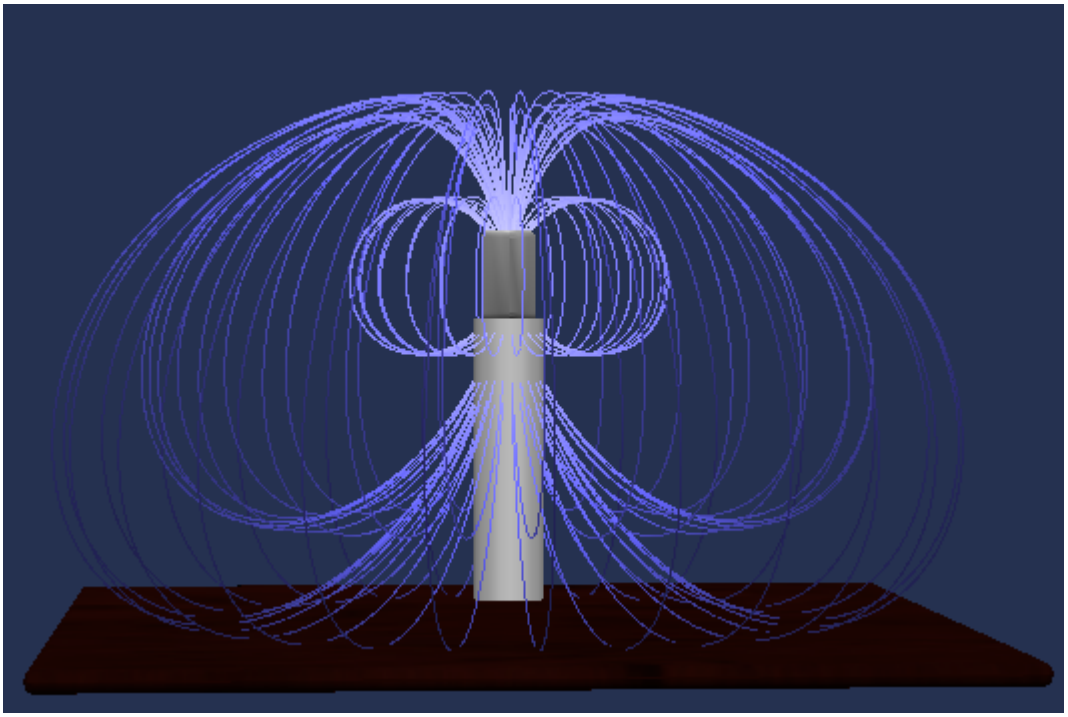

Figure 6: Field Lines with Clones

TriMesh. When instantiating one of those shared objects the target is given to the constructor. Unfortunately, only inner nodes and triangle meshes are supported by JMonkeyEngine for sharing. The line nodes are not derived of any of those. To solve this problem a *SharedLine* class was implemented. With every render cycle the *draw* method of every scene graph object is called. With that call the *SharedLine* sets its properties to the target and causes the target to be drawn instead of itself.

Some additional method calls from outside are to be made on the field line primitive. Thus, an interface *TFieldLineNode* is introduced. This interface is implemented by both, the Java3D and the JMonkeyEngine *FieldLineNode*. The most important methods of this interface are:

```
public void setLineGeometry(int len1, float[] line1,
                            int len2, float[] line2);
public void setLineGeometry(int len1, float[] line1,
                            float[] colors1,
                            int len2, float[] line2,
                            float[] colors2);
public void setSymmetry(int count, Vector3d axis);
```

The first two methods set the line geometry. The ***float*** arrays contain all

vertices for two lines belonging to a single field line. With most experiments the number of coordinates for each lines is 100. Since there are two lines and each vertex consists of three numbers (in 3D-space) this leads to 600 floats. The second method takes a float array with one three-dimensional color by vertex. These two methods are called every time the field changes. During the simulation this usually happens with every new frame meaning that the methods are called 20 times a second. For that reason it is very important to keep the runtime of the methods low. The third method is to change the symmetry axis and the number of field line copies. Symmetry axis are rarely changed in the simulations. The symmetry count changes mostly on user interaction. Thus, performance issues will occur in the *setLineGeometry* methods rather than in *setSymmetry*.

To keep the execution time of the methods stated above low some considerations have to be made. As opposed to Java3D which takes the line vertices as ***float[]***, JMonkeyEngine uses *java.nio*-buffers. For floats the class *java.nio.FloatBuffer* is used. The advantage of such buffers is the increased performance on the access operations. This is due to the fact that such a buffer allows bulks of data to be written or read. With ordinary arrays an index bounds check is done with every single array access operation. With *java.nio*-buffers a whole bulk of data can be read or written with one single bounds check.

Since the *setLineGeometry* methods are getting the data in float arrays a *FloatBuffer* has to be created and filled with the data. The vertices can be written as a whole bulk. With the colors it is more difficult. The given arrays *colors1* and *colors2* contain the data for three dimensional colors, i.e. index 0 to 2 contain the red, green and blue value for the first vertex. Index 3 contains a red value again, in this case for the second vertex and so on. Since JMonkeyEngine uses four dimensional colors only three values can be written as a bulk. Then a *1* is added as forth value for each vertex. This way, as many bulk writes as vertices are necessary. In Java3D the parameters can be used directly and be passed to the low level objects. Java3D uses tree dimensional colors and float arrays. The *setLineGeometry* methods are obviously designed for Java3D output. This way the JMonkeyEngine code can not be made as performant as the Java3D code as soon as the interface is keep as it is now.

All three discussed methods are ***synchronized*** in the Java3D version. This is done to be thread safe. Adding the ***synchronized*** keyword to a method is the quickest way to keep a class thread safe. However, with this approach performance is given away. For all members the same lock, namely "***this***" is used. This often locks a thread which would not need to be locked. In JMonkeyEngine the *Line* scene graph nodes are the only members of the

FieldLineNode class to be read by another thread, namely the MTGame renderer thread. Since the changes of a node should always occur in the renderer thread they are simply passed to that thread. This way no additional synchronization is needed. Because the renderer thread has to render the whole scene many times a second it should get as little additional jobs taking as little time as possible. For that reason the *java.nio.FloatBuffer*s are local variables filled by the method-calling thread. This buffers are then passed to the renderer thread which only has to update the line geometries.

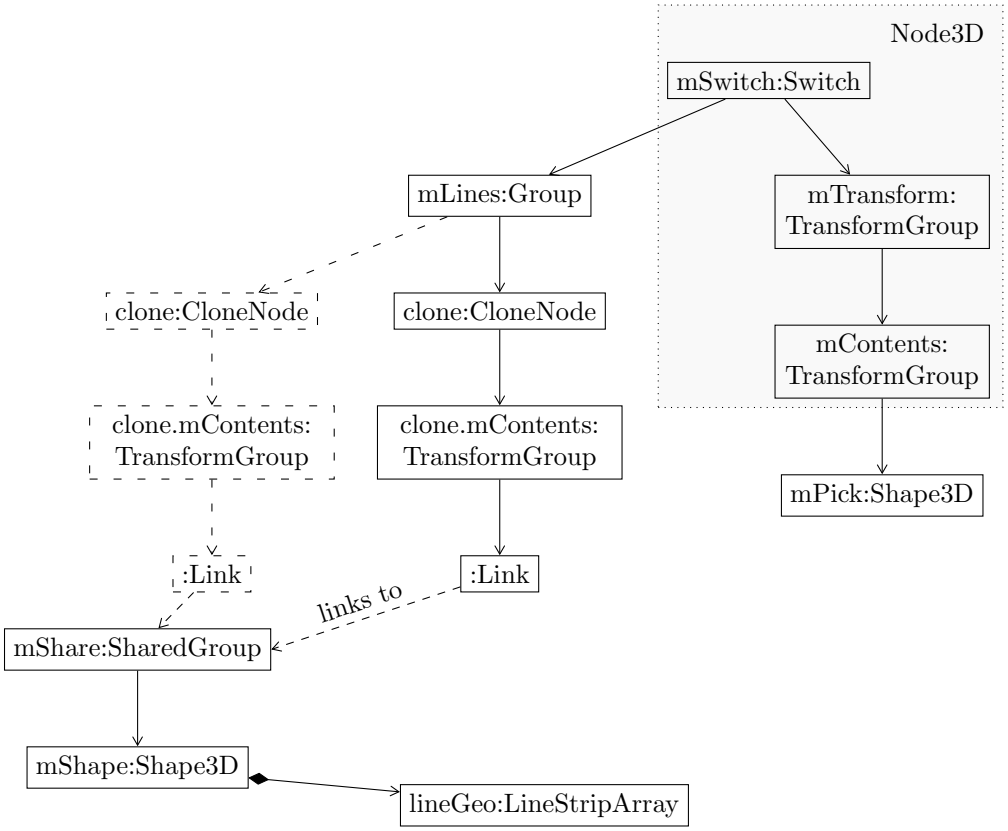

Figure 7: FieldLineNode Scene Graph in Java3D

Since the scene graphs of Java3D and JMonkeyEngine are different a new graph for JMonkeyEngine was designed. In figure 7 the scene graph of the Java3D *FieldLineNode* is shown. The part of the scene graph which is specified in *Node3D* is used to show pick marks. The actual field line part of the scene graph is attached to *mSwitch*. This way all transform methods defined in the *TNode3D* interface only affect the pick shapes but not the field lines. Those can only be influenced by the methods defined in the *TFieldLineNode* interface. All the clones needed are attached to the *mLine*

node. The dashed path to the left in figure 7 would create a second field line. An arbitrary number of clones can be attached to *mLines* without increasing the memory usage significantly. This is done with the *setSymmetry* method. The *Link* node at the bottom of the scene graph links to a *SharedGroup*; every single *Link* node of each clone links to it. Thus the *FieldLineNode* has got only one shared node. Its child is the shape of the field line. It contains the geometry of the field line, in this case a Java3D *LineStripArray*. When changing the shape of a field line, only the *lineGeo* needs to be changed. This is done by the *setLineGeometry* methods.

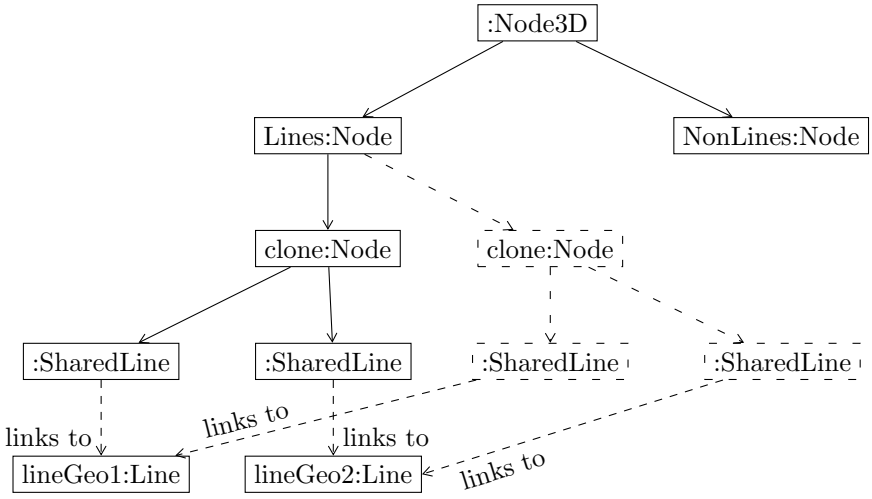

Figure 8: FieldLineNode Scene Graph in JMonkeyEngine

In figure 8 the scene graph of a JMonkeyEngine *FieldLineNode* is shown. One difference is that the *Node3D* does not contain several sub-nodes. It *is* the whole Node3D. As stated previously in this section the transforms applied on Java3D's *mContents* node are applied on all the childs of JMonkeyEngine's *Node3D* by default. In order to be consistent with the Java3D code this behavior needs to be overwritten in the *FieldLineNode*. Otherwise all the transforms are also applied to the lines. Furthermore all other transform methods declared in *TNode3D* are to be overwritten because they should only apply to the *NonLines* node.

The clones are attached to the *Lines* node. This is again done by the *setSymmetry* method. Because the line geometry can most easily be represented in JMonkeyEngine as two *Line* nodes a clone node has two *SharedLine* nodes as children. That way each node refers to *lineGeo1* as well as to *lineGeo2*. Similarly to the Java3D version as many copies can be added as needed. The dashed path to the right in figure 8 shows an additional clone.

ArrayNode

An array node handles a matrix of *Node3D* nodes. In TealSim this is mostly used for field direction grids. They contain an array of Arrows. All the arrows shown in figure 9 are held by a single *ArrayNode*. Such a field direction grid is a bottleneck concerning the performance of the simulation. All the arrows are calculated according to the field and it is possible to have dozens of arrows. When drawing the arrows the rotation and scale of every single arrow is changed.

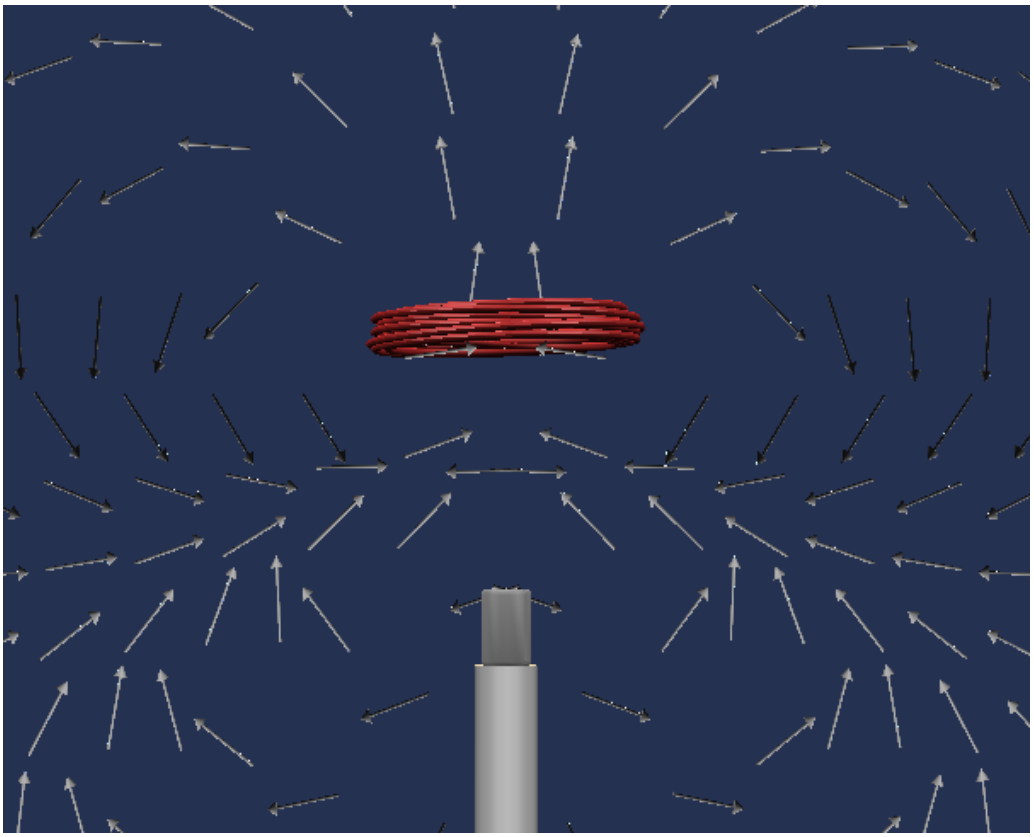

Figure 9: Field Direction Grid

Similarly to the field line node an interface *TArrayNode* has to be declared to allow both, Java3D and JMonkeyEngine graphic output. This interface consists of the following methods:

```
public void setVisible(int fromIdx, int toIdx, boolean state);
public void addNode(TNode3D node);

public TNode3D get(int idx);
public int getNodeCount();
```

```
public Iterator iterator();

public void removeNode(TNode3D node);
public void removeNode(int idx);
public void removeAll();
```

Those methods enable the calling code to access the single nodes inside the array. An iterator to step through the nodes is also provided. In Java3D the *ArrayNode* contains a *Vector* holding a reference to all the shapes inside the array. This way it is simple to provide the required functionality. All the methods that require indices can get the elements from the *Vector*. The *Iterator* can also be obtained this way. However, with this approach the vector and the scene graph must always kept synchronous. If e.g. a node should be added it has to be added to both, the scene graph and the array. Such code-parts are usually to be avoided.

In JMonkeyEngine indexed child access is already provided by the *Node* class. Thus, *Node3D* inherits this functionality and there is no need for another member list, storing the same objects. However, because MTGame's renderer thread accesses all the nodes the methods have to be synchronized. E.g. removing a node while the renderer thread processes it would leat to a runtime error.

In order to provide the iterator functionality a new iterator class was defined as an inner class of *ArrayNode*. *Node*'s can now be added to the scene graph without using the *TArrayNode* interface. The *getChildren* method of the *Node* class returns a *List*. The defined iterator takes the one obtained from this list as back end. It also stores the number of elements added by the *TArrayNode* method calls. This way the iterator does not access the wrong elements in the scene graph.

3.1.4 Colors and Materials

Colors can be coded in different ways. In TealSim the RGB model where colors are written as an additive mixture of a defined red, a green and a blue color is used. To store the values there are different approaches. Java awt e.g. uses a single 32 bit integer, 8 bits for each color. The remaining 8 bits are used to store the transparency of the color. Since this are four values, the *java.awt.Color* class stores four dimensional colors. As mentioned in section 3.1.3 Java3D uses three dimensional colors. The transparency is stored as a separate value. Java3D uses the class *javax.vecmath.Color3f* which stores the values as three floats ranging from 0.0 to 1.0. Each float has a precision of 32 bit. This way the values can be defined more precisely

than with the awt colors using 8 bits per color channel. JMonkeyEngine uses its *com.jme.renderer.ColorRGBA* color class. The colors are stored as floats, one for each channel. As with the awt colors the transparency is the fourth value. A value of 1.0 means opaque, 0.0 means totally transparent.

Another difference of JMonkeyEngine compared to Java3D is how different colors are applied. As shown in figure 4, the leave nodes of the Java3D scene graph reference to a so-called appearance. Such an *Appearance* object holds all kinds of render state informations. This can be colors, textures, materials, line attributes and so on. In JMonkeyEngine the render states are attached to the scene graph nodes directly without a containing appearance class. Furthermore the states are different and the interface to such states is not very similar to Java3D. E.g. a colored material can be set directly to the appearance in Java3D:

```
// applying color to appearance
app.setMaterial(new Material(new Color3f(Color.BLUE),
                             new Color3f(Color.BLUE),
                             new Color3f(),
                             new Color3f()));
```

Note, that the four parameters for the *Material* constructor are the ambient, diffuse, specular and emissive color. In JMonkeyEngine they have to be added to the *MaterialState*:

```
// obtaining state frome scene graph node (spatial)
MaterialState ms = (MaterialState) spatial.
                   getRenderState(RenderState.StateType.Material);

// creating material state, if not already there
if (ms == null) {
  ms = DisplaySystem.getDisplaySystem().getRenderer()
                                       .createMaterialState();
  spatial.setRenderState(ms);
}
ms.setEnabled(true);

// setting colors
ms.setAmbient(ColorRGBA.BLUE);
ms.setDiffuse(ColorRGBA.BLUE);
```

Since there was only Java3D previously some interfaces contained methods with a Java3D *Appearance* as parameter. Such a specific code can obviously not be used with different graphic libraries. To overcome that problem a container with material information namely the *TealMaterial* class was introduced. All material information needed in TealSim can be stored in an object of this container class. This information contains

- colors (ambient, diffuse, specular and emissive),
- transparency,
- shininess,
- culling
- and face mode.

The first decision to be made was about the dimensions of the colors. Since JMonkeyEngine uses four dimensional colors a different transparency value can be specified for each of the four colors. With Java3D there are only three dimensional colors and one transparency value for all four colors. In order to support the feature of having different transparency values for each color four dimensional colors are used in *TealMaterial*. For that purpose the class *javax.vecmath.Color4f* was chosen. With Java3D the transparency would be the same for all four colors. In order to make it possible to specify the transparency first and the colors later, the transparency must be stored in a separate value as well. This way it is also possible to give information about the transparency only without specifying any colors. This is useful if the class is used for changing single values. The interface *TMaterial* declares all the methods defined in *TealMaterial*. The ones for the colors and the transparency are as follows:

```
public void setTransparency(float trans);
public float getTransparency();

public Color4f getAmbient();
public Color4f getDiffuse();
public Color4f getSpecular();
public Color4f getEmissive();

public void setAmbient(Color3f col);
public void setAmbient(Color4f col);
public void setDiffuse(Color3f col);
public void setDiffuse(Color4f col);
public void setSpecular(Color3f col);
public void setSpecular(Color4f col);
public void setEmissive(Color4f col);
public void setEmissive(Color3f col);
```

The Java3D specific code will mostly use the *set* methods with the *Color3f* parameters and the transparency methods. JMonkeyEngine specific code will use the *set* methods with the four dimensional colors containing the transparency value. Since the transparency *and* the four dimensional colors

need to be stored in the *TealMaterial* class the fourth dimension of the colors and the transparency value needs to be kept synchronous. Every time one of the *set* methods stated above is called the value has to be kept consistent. Within a data model it is always a disadvantage to store data redundantly. To overcome that problem the colors would have to be stored in only three dimensions. The feature of giving different transparency values for each of the four colors would have been removed with this approach.

The range of the values given by the methods has to be specified. The red, green and blue color channel of the colors contain values between 0.0 and 1.0. This is because Java3D as well as JMonkeyEngine use them in the same range. The *getTransparency* and the *setTransparency* methods work with floats of the same range where a value of 0.0 means opaque. As mentioned before, these two methods are mostly used by Java3D specific code. This library uses a value of 0.0 for "opaque". However, the fourth dimension of the colors has to be set to the inverse value, where 0.0 is transparent and 1.0 is opaque. With JMonkeyEngine the *BlendState* has to be enabled to turn on transparency.

The value range definitions had to be made for the shininess as well. There are two methods regarding the shininess in the *TMaterial* interface:

```
public void setShininess(float shine);
public float getShininess();
```

Those methods work with values ranging from 0.0 to 1.0. However, JMonkeyEngine uses float values between 0 and 128. This means that the value stored in the *TealMaterial* object has to be multiplied by 128 in the JMonkeyEngine specific code parts. The value is then set on the *MaterialState* as it is done with the colors.

With culling a either the front, the back or both faces can be culled,i.e. they are not shown. With culling performance can be gained when rendering, because there are less things to be rendered. A getter and a setter method are defined for culling:

```
public void setCullMode(int mode);
public int getCullMode();
```

This integer value is a bit mask. 0 means no culling, 1 means backface culling and 2 means frontface culling. If both faces are to be culled the value can be set to the bitwise or value of frontface and backface culling namely 3. To enable culling in JMonkeyEngine, the *CullState* has to be set up accordingly.

The last material attribute contained within the *TealMaterial* class is the face mode. Most 3D shapes are built up with triangles, but they may also be built with quads. The face mode defines whether the triangles or quads

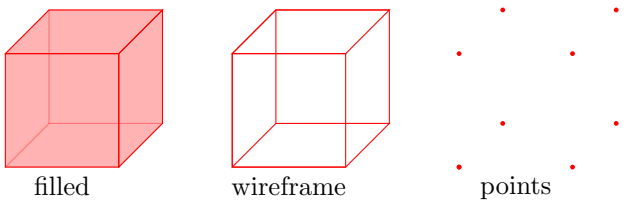

Figure 10: Different Face Modes

should be drawn filled, in wire-frame mode or in point mode. In figure 10 a cube built of quads is shown in those three modes. There is again a getter and a setter method for the face mode in *TMaterial*:

```
public int getFaceMode();
public void setFaceMode(int mode);
```

As with culling a bit mask is used for the three different face modes. JMonkeyEngine provides a *WireframeState* which can be enabled to show the scene graph node as wire-frame. There is no native support for the point mode in JMonkeyEngine. This state can only be realized by replacing the object with one point per vertex, i.e., another scene graph node would be used instead. Since this is currently not used by any simulation it is not implemented.

A big advantage of having an own material class is that it can be used easily with both graphic libraries. Two new methods can now be added to the *TNode3D* interface:

```
public TMaterial getMaterial();
public void setMaterial(TMaterial material);
```

Both, the Java3D and the JMonkeyEngine version of *Node3D* have to implement those methods. Every time some of the properties the *TealMaterial* class contains should be changed the *setMaterial* can be called. If e.g. only the ambient color should be set, all other colors can be set to ***null***. This means that they remain as they are. In JMonkeyEngine the following protected static methods were defined in *Node3D*:

```
protected static TMaterial getMaterial(Spatial shape);
protected static void setMaterial(final TMaterial material,
                                  Spatial shape);
```

Those methods can be called by every subclass of *Node3D*. The setter and getter methods for the materials inherited from *TNode3D* simply call this static methods with ***this*** as parameter. The static methods are also very useful for setting materials on objects located lower in the scene graph.

By default JMonkeyEngine applies the material state only to one material face namely the front face. This does not affect closed shapes visibly because

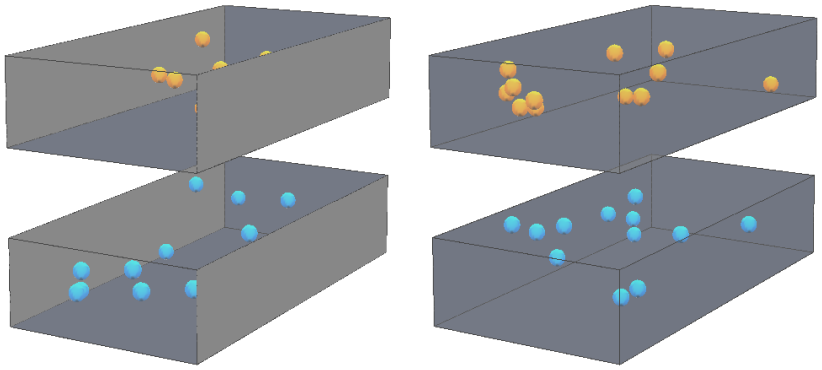

Figure 11: Capacitor Simulation with different Material Faces

the inner part of them can never be seen. Applying the material on only one side effects the rendering performance positively. However with surfaces which can be seen from both sides the material has to be applied on the whole surface. In TealSim the walls for example are two dimensional with thickness zero. Figure 11 shows the capacitor simulation. The charges (displayed as spheres) are surrounded by walls. In the left image the material is only applied on the front faces. On the faces showing the back the transparency is not applied, i.e. the walls are opaque. The right image shows the material applied on both faces. To specify the material faces the *setMaterialFace* method on the material state is provided. It can be set to *front*, which is the default, *back* or *both*. In order to keep the performance high the value *both* is only used when needed (e.g. on walls).

3.1.5 Specifying Interface Data-Types

In section 3.1.4 the data type for colors used for the interface between lower level graphic shapes and the other code was chosen to be *Color4f*. Whenever there are different classes used for the same purpose one has to be chosen to be used in the interfaces. Subsequently, conversion methods have to be provided for use in lower level classes like the Java3D and JMonkeyEngine specific code parts. This section gives an overview of the decisions made.

Transforms

Transforms are used for rotating, scaling and translating objects, or whole scene graph branches. As already mentioned in section 3.1.3, Java3D uses an own scene graph object of type *TransformGroup* only for the purpose of transforming the branch. The actual transform information is stored in its

own class; for three dimensional transforms it is the *Transform3D* class (see figure 12). The transforms can be set by different *set* methods and obtained

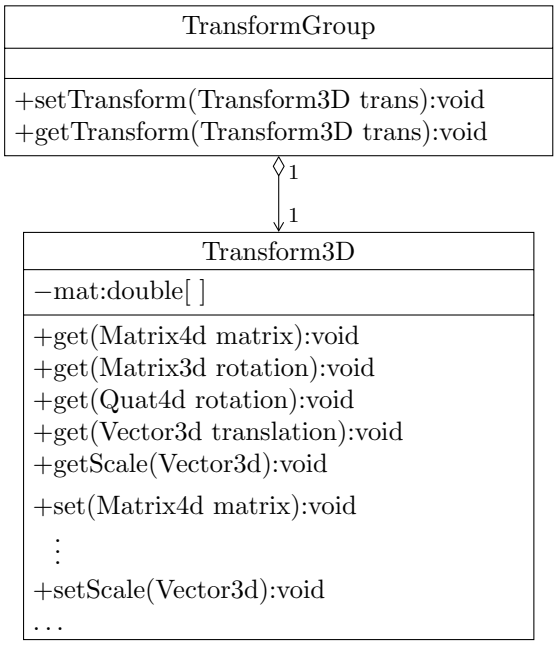

Figure 12: Class diagram of Java3D's Transforms

by *get* methods. To distinguish what kind of transform is addressed the parameter type is used. A *Matrix3d* is a 3×3 matrix and represents a rotation Matrix with double precision values. Since rotations can also be defined with quaternions the *get* and the *set* method with the *Quat4d* parameter also effects the rotation only. Translations can be defined with three dimensional vector, one dimension for translations of each of the three axis. In Java3D the class *Vector3d* is used for that. The `d` at the end of the class name stands for "double". It indicates the use of double values internally. There is also a *Vector3f* class using floats. The *Transform3D* class also defines *get* and *set* methods for float using classes. All those vector, matrix and quaternion classes are found in the "vecmath" library in package *javax.vecmath*.

Only if a type can be used for different types of transforms other methods are defined. This is the case with translation and scale. Both can be expressed by three dimensional vectors. Since the *set* method with the *Vector3d* was chosen to set the translation, the *setScale* method was defined. With a three dimensional vector different scales can be applied for every axis. Since all the transform operations can be expressed by a single 3×4 matrix the *Transform3D* class uses a matrix internally. With a *Matrix4d* parameter to the *set* and the *get* method the whole matrix can be set or obtained. The

transform class is also capable of other special operations like multiplying transforms with each others. Problems like matrix singularity are addressed as well. The transform is applied to the *TransformGroup* scene graph object with the *setTransform* method and can be obtained with the *getTransform* method.

With JMonkeyEngine the transforms are applied directly to any node of the scene graph. As shown in figure 13, the base class of all scene graph classes *Spatial* contains the methods for setting transforms. These methods are inherited to all derived classes and thus, to all scene graph objects. As opposed to Java3D the different transform types translation, rotation and scale are not stored in a single matrix, but in one member variable each. This way some problems like singular matrices are avoided. The additional layer of a container for the transforms (as it is with the *Transform3D* class in Java3D) is skipped. This has the advantage of less lines of code. If the scale vector should be obtained in Java3D from a scene graph node would look as follows:

```
public Vector3d getScale() {
  Vector3d s = new Vector3d();
  Transform3D trans = new Transform3D();
  mTransform.getTransform(trans);
  trans.getScale(s);
  return s;
}
```

In JMonkeyEngine the same method requires much less code:

```
public Vector3d getScale() {
  Vector3f scale = this.getLocalScale();
  return new Vector3d(scale.x, scale.y, scale.z);
}
```

The above code is from the two *Node3D* classes. In the implemented interface *TNode3D* it was decided to used the vecmath classes for the interface. Since JMonkeyEngine works with its own vector classes and entirely with floats, those classes have to be converted to the according vecmath class. In the case of a vector this is rather simple. The "return" line in the code above shows the conversion. A new object is created with the values in the constructor. This also works for quaternions, i.e. the classes *Quat4d* and *Quaternion*.

For backward compatibility reasons with the Java3D code some methods of the *TNode3D* interface are using the *Transform3D* as well:

```
public void setModelOffsetTransform(Transform3D t);
public Transform3D getModelOffsetTransform();
```

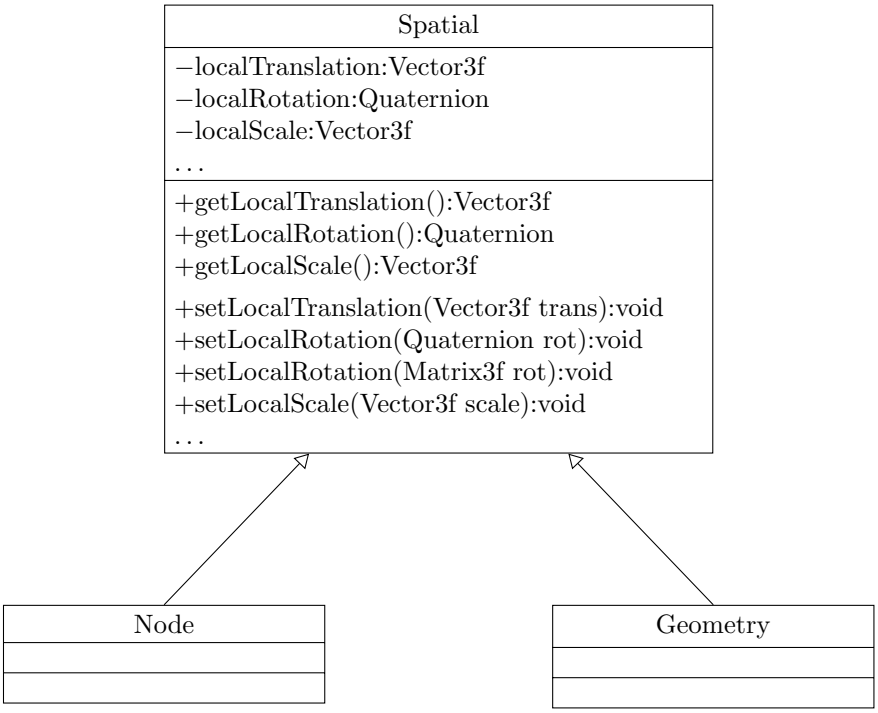

Figure 13: Class diagram of JMonkeyEngine's Transforms

With Java3D the transforms can be passed directly to the underlying scene graph object. With `JMonkeyEngine` the translation, rotation and scale has to be obtained from the *Transform3D* object. Then the obtained vecmath elements can be casted to JMonkeyEngine objects. Those are then applied to the scene graph object. For the *getModelOffsetTransform* method, the three values are to be obtained from the scene graph objects. Then a *Transform3D* object is created with those values and returned. For performance reasons the two methods should be removed in future versions of TealSim. This would also make the design cleaner, since interface parameter types should be rather simple classes or primitive types.

Bounding volumes

An objects bounding volume contains the object completely. Bounding volumes are always simple shapes. They are e.g. used to determine if the belonging object is still in the field of view. If not it does no need to be rendered. Bounding volumes usually build class hierarchies (see figure 14). The single specific bounding volumes are derived from an abstract class. With Java3D the specific bounding volume classes provided are

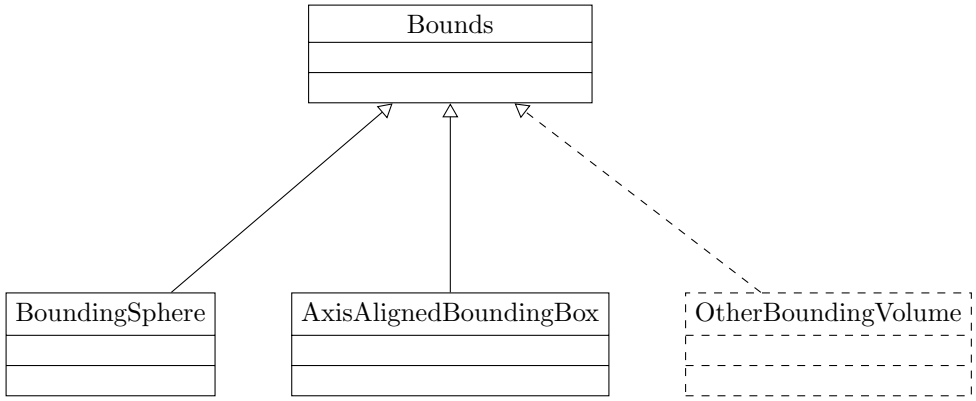

Figure 14: Class hierarchy of Bounding Volumes

- *BoundingSphere*
- *BoundingBox* and
- *BoundingPolytope*

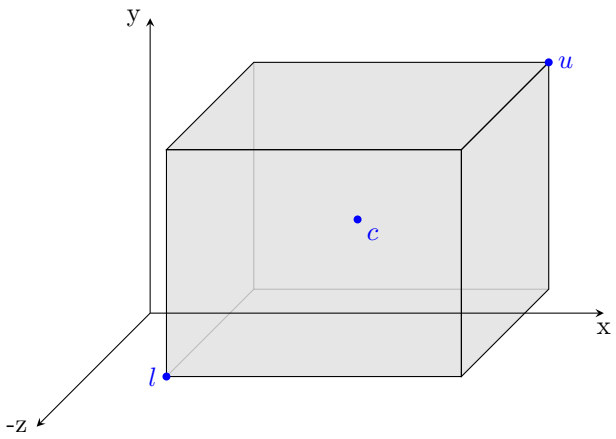

Figure 15: Bounding Box Coordinates

The sphere is defined by a center point and a radius, the box by two tree dimensional points. In figure 15, the point l marks the lower point and u the upper point. Since the box is axis aligned, those points are enough. With a bounding polytope a bounding volume can be build up with at least four half-spaces. This way potentially arbitrary shapes are supported. JMonkeyEngine defines the following bounding volumes:

- *BoundingSphere*

- *BoundingBox*
- *BoundingCapsule*
- *OrientedBoundingBox*

The sphere is defined the same as with Java3D. For axis aligned box the center point and the extends to each of the three axis define the volume. The capsule is defined by a line segment forming the capsules cylinder part, by a center point and by the caps radius. With the oriented bounding box, a box that is not axis aligned can be specified as a bounding volume. Additionally to the extends in each directions the three axis can be defined.

Fortunately there are only bounding spheres and boxes used in TealSim. Those are the two supported by both graphic engines. For the interface newly defined bounding volume classes were used. The *TNode3D* interface contains a *getBoundingArea* method. This requires the implementing code in the two *Node3D* classes to construct a proper bounding volume out of their native one. The newly implemented bounding volumes are defined similarly to the ones in Java3D; the bounding box is defined by a lower and an upper point. This way it is very simple to convert the volumes from Java3D, as the following code fragment shows:

```
// bounding sphere:
if(j3dBounds instanceof BoundingSphere) {
  double radius = ((BoundingSphere)j3dBounds).getRadius();
  Point3d center = new Point3d();
  ((BoundingSphere)j3dBounds).getCenter(center);
  returnValue = new teal.render.BoundingSphere(center,radius);

//bounding box:
} else if (j3dBounds instanceof BoundingBox) {
  Point3d lower = new Point3d();
  Point3d upper = new Point3d();
  ((BoundingBox)j3dBounds).getLower(lower);
  ((BoundingBox)j3dBounds).getUpper(upper);
  returnValue = new teal.render.BoundingBox(lower,upper);
}
```

To obtain if the bounding volume is a sphere or a axis aligned box the ***instanceof*** operator has to be used. This call is rather slow, but the *getBoundingArea* is not called very often.

Since the JMonkeyEngine bounding box is defined differently, the code is a bit more complicated:

```
Vector3f center = bounds.getCenter();
switch(bounds.getType()) {
```

```
    case Sphere: // bounding sphere
      javax.vecmath.Point3d vecMathCenter = new Point3d(center.x,
      center.y, center.z);
      float radius = ((BoundingSphere) bounds).radius;
      return_value = new teal.render.BoundingSphere(vecMathCenter,
                                                    radius);
      break;
    case AABB: // axis aligned bounding box
      return_value = new teal.render.BoundingBox();
      Vector3f extend = new Vector3f();
      ((BoundingBox)bounds).getExtent(extend);
      ((teal.render.BoundingBox)return_value).setLower(
                                new Point3d(center.x-extend.x,
                                            center.y-extend.y,
                                            center.z-extend.z));
      ((teal.render.BoundingBox)return_value).setUpper(
                                new Point3d(center.x+extend.x,
                                            center.y+extend.y,
                                            center.z+extend.z));
      break;
    case OBB: // oriented bounding box
      // this is casted to an enclosing j3d bounding sphere.
      OrientedBoundingBox obb = (OrientedBoundingBox) bounds;
      obb.computeCorners();
      BoundingSphere sphere = new BoundingSphere();
      sphere.averagePoints(obb.vectorStore);
      Vector3f sphereCenter = sphere.getCenter();
      Point3d vmCenter = new Point3d(sphereCenter.x,
                                     sphereCenter.y, sphereCenter.z);
      return_value = new teal.render.BoundingSphere(
                                     vmCenter, sphere.getRadius());
    default:
      TDebug.println(1,"Bounding volume not supported!");
  }
```

As opposed to Java3D the ***instanceof*** operator is not necessary here because the JMonkeyEngine bounding volumes define a *getType* method returning an *enum* value for the specific type. Because JMonkeyEngine defines all its bounding volumes with a center point the base class *BoundingVolume* already contains a *getCenter* method. The conversion of the bounding volume is straight forward because it is defined in the same way as with the interface bounding type. With the axis aligned bounding box the lower and the upper point have to be calculated with the center point and the extends:

$$\vec{l} = \vec{c} - \vec{e}$$
$$\vec{u} = \vec{c} + \vec{e}$$

The vector $\vec{e}$ indicates the extends to each axis. The JMonkeyEngine oriented bounding box volume is converted into a bounding sphere. This is currently not needed because the oriented bounding box is not used. However if this volume is necessary in the future the code is prepared for that.

3.1.6 3D Models

As mentioned in section 2.2, external 3D models can be used in simulations. Those are mostly 3DStudio-max[14] files with `.3DS` file endings. This is an open file format, but Java3D does not come with an importer. For that reason the class *teal.render.j3d.loaders.Loader3DS* was implemented previously by TealSim developers. It converts a 3DS model to a Java3D scene graph. In JMonkeyEngine such a functionality is already implemented in the subclasses of the *FormatConverter* class. In order to convert the data of a `.3DS` file into a JMonkeyEngine scene graph, only a few lines of code are needed:

```
ByteArrayOutputStream byteOutput = new ByteArrayOutputStream();
InputStream in = modelUrl.openStream();
FormatConverter converter = new MaxToJme();
converter.convert(in, byteOutput);
final byte[] out = byteOutput.toByteArray();
Node nd = (Node)BinaryImporter.getInstance().load(
                                    new ByteArrayInputStream(out));
```

The *FormatConverter*'s *convert* method reads a byte stream with 3DS data and writes the JMonkeyEngine's binary format into an output stream. This output stream is then converted to a scene graph using the *BinaryImporter* class. Textures are not stored directly inside 3DS files; links to images are used. To access the textures the texture file path has to be set prior to the importing code:

```
ResourceLocatorTool.addResourceLocator(
                ResourceLocatorTool.TYPE_TEXTURE,
                new SimpleResourceLocator(modelUrl));
```

Using JMonkeyEngine's native importer works well. Code for different file formats can be added quickly in the same way. However, the importer reads the 3DS file differently than the written importer for Java3D. The models have to be rotated by 270 degrees around the x-axis. There is also an offset to be applied to the y axis. The Java3D version also applies this offset.

There is a scene factory's *makeNode* method with a *TRendered* type as parameter (see also figure 3). This method is similar to the one with

[14] `http://www.3dstudio-max.com/`

the *NodeType* as parameter, but it obtains the type by a *getNodeType* call on the rendered object. As explained in section 2.2.1, the rendered objects represent the *Model* in the MVC pattern. If a rendered objects is represented by an external 3d model, the rendered's *getModel* method returns an object of class *Model*. This object contains all needed data to load the model:

- the path to the model,
- a position offset vector,
- a scaling vector,
- and optionally a path to the textures.

If no path to textures is given the model path is assumed to be the texture path as well. With the given data the model can be loaded and the required transforms (scale and translation) can be applied.

3.1.7 The Viewer

The GUI of TealSim consists of come control panels and a 3D panel. This 3D panel is filled by the viewer. A three dimensional viewer is embedded into the GUI which is a *javax.swing.JPanel* object. That viewer also holds the whole scene graph and the lights as well as the canvas drawn on. As it was done with the Java3D version of the viewer *ViewerJ3D*, the JMonkeyEngine version *ViewerJME* was also derived from the *AbstractViewer3D* class. To assure that the right type of viewer is created accordingly to the graphic output system (Java3D or JMonkeyEngine), a *makeViewer* method was added to the scene factories (see also figure 3).

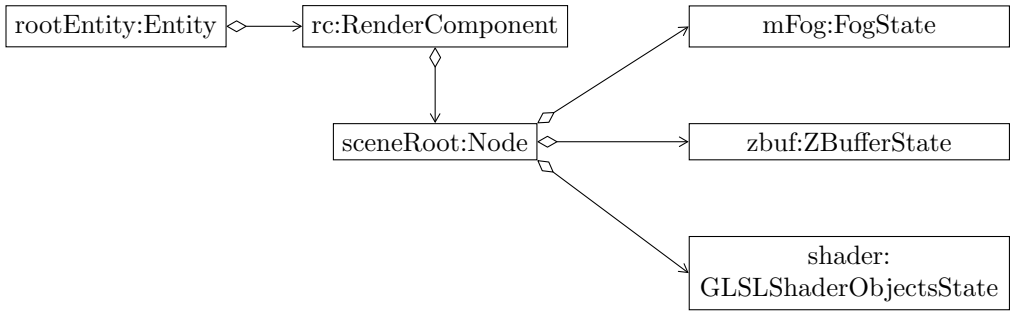

Figure 16: Object diagram of Viewer's Root Entity

In MTGame a scene is based on entities and their components. In our case we will need a root entity containing a render component and its scene

graph and a camera entity holding the camera scene graph and the required components for changing the camera perspective. The light can be directly added to the render manager. In figure 16 the entity *rootEntity* is shown. It contains a render component holding the top node of the scene graph. Global states like the z-buffer, fog and shade are added to this scene root node. The fog is disabled per default and can be influenced by the following methods:

```
public void setFogFrontDistance(double front);
public double getFogFrontDistance();
public void setFogBackDistance(double back);
public double getFogBackDistance();

public void setFogTransformFrontScale(double percent);
public double getFogTransformFrontScale();
public void setFogTransformBackScale(double percent);
public double getFogTransformBackScale();

public void setFogInfluencingBounds(Bounds bounds);
public Bounds getFogInfluencingBounds();

public void setFogEnabled(boolean enabled);
public boolean isFogEnabled();
```

All those methods are declared in the interface *TViewer3D* which has to be implemented. Most methods names are self describing. The influence bounds of the fog take a bounding volume in which the fog is applied. The z-buffer is used by JMonkeyEngine to get information about the depth of the three dimensional image. This way it is determined if some object has to be rendered or not (possibly because it is behind an other object). The shader is added to the scene root to add some shade. The states applied to the top node are inherited to all descending nodes. Because it is the top node every single node in the scene graph inherits the states. All the *Node3D* objects of a simulation are also added as a child to the root node. To do so the following methods had to be implemented:

```
public void addDrawable(TAbstractRendered draw);
public void addDrawableBulk(Collection<TAbstractRendered> bulk);

public void removeDrawable(TAbstractRendered draw);
public void removeDrawableBulk(
                              Collection<TAbstractRendered> bulk);
```

With the *getNode3D* method the scene graph object can be obtained from the rendered object. These methods can be called whenever an object should be added or removed from the scene graph. This happens mostly at the initialization phase of a simulation.

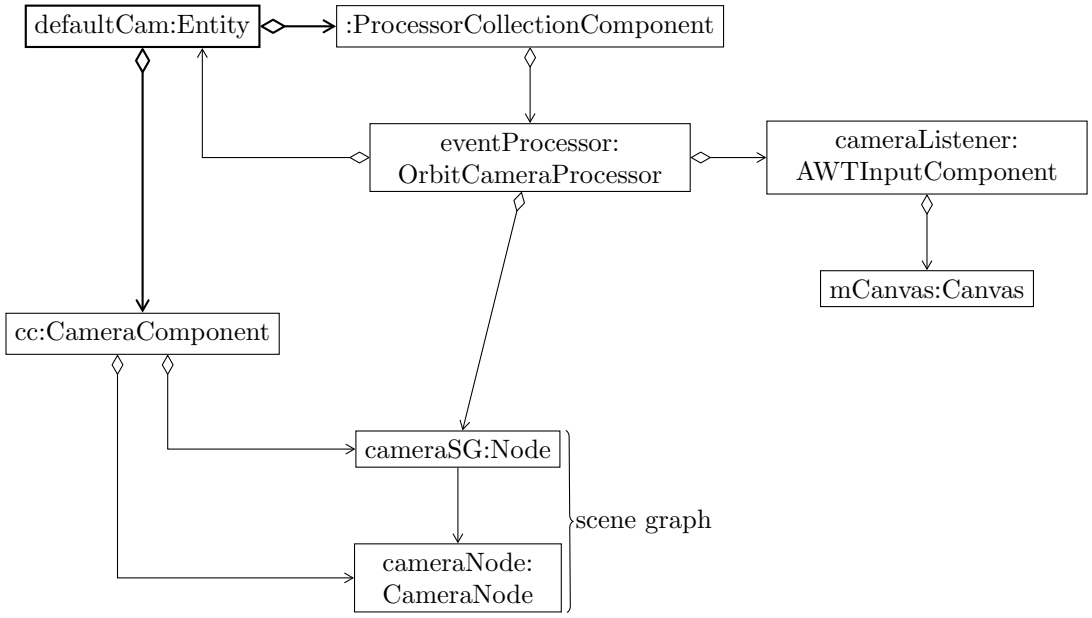

Figure 17: Relations of the Camera Entity

The second entity needed it the camera entity (see figure 17). First a camera component is added to the entity. It is created by MTGame's render manager which can be obtained from the world manager using the *getRenderManager* method. At creation time the camera component needs to know some data:

- the scene graph node *cameraSG* which is the top node of the camera scene graph,
- the actual camera node representing the camera in the scene graph,
- the resolution in both, x and y direction,
- the cameras field of view in degrees,
- the front an back clipping distance which specify the closest and the furthest distance captured by the camera,
- and if the camera is the main camera, which is ***true*** in our case.

The field of view and the clip distances can be changed later by the according methods declared in the *TViewer3D* interface. Those methods are used to put the elements of a simulation into the field of view. The sizes of the different simulations varies between values under 10 and some hundreds. In

TealSim's desktop version this is compensated by the camera angle and the camera distance. This way the viewed simulation sizes are roughly the same for all simulations. The algorithm applied to find the according field of view to a simulation was already implemented for Java3D and could be used in the same way for JMonkeyEngine.

In order to process the mouse input another component can be added to the camera entity. A *ProcessorCollectionComponent* which can hold a list of processors is used for that purpose. A MTGame processor is used to perform some tasks. It is armed with some arming condition. The the condition is fulfilled the processor event is done in two phases. First the processors *compute* method is called. It can do everything but changing live displayed object. The second phase is done with calling the processors *commit* method. The implementation of this method is allowed to change the scene graph, because it is executed in the renderer thread (see section 3.1.8). For now only one processor is added to the processor collection namely an *OrbitCameraProcessor*. This processor is implemented in the MTGame library and is used to orbit a camera around a specified point. It needs references to several objects:

- an *AWTInputComponent* object which provides the input data from mouse and/or keyboard. In our case we only process the mouse input
- the top node of the camera scene graph *cameraSG*
- the camera entity itself

In order to process the mouse input the camera listener needs a reference to the canvas. This canvas has to be created prior to the camera listener. Creating and setting up the canvas is done in a few steps:

1. A render buffer is created by the render manager. This implicitly creates the canvas.
2. The render buffer is added to the render manager.
3. The canvas is obtained from the render buffer using the *getCanvas* method.
4. The canvas' visibility is set to ***true***.
5. X and y coordinate bounds of the canvas' are set.

The canvas as well as the render buffer should never be created "by hand" using the constructor and not the render manager, because MTGame relies

on that. Since the render buffer is also needed at the camera creation process, the canvas is created and set up prior to camera creation. The camera component needs to be set on the render buffer.

The *TViewer3D* interface defines some more methods the *ViewerJME* class has to implement. E.g. the mouse and keyboard controls could be set, gizmos can be turned on and off, or zooming can be enabled and disabled. However, since the goal of the whole work is to let TealSim run in Wonderland this additional functionality was not implemented. It is left for future work (see also section 5.1). The viewer contains all the functionality needed to show the experiment and to be able to change the view. It reacts slightly differently to mouse input than the Java3D version.

3.1.8 Threading Issues

Because JMonkeyEngine comes without a threading model, MTGame adds such a model to JMonkeyEngine. The threads are build around so-called "entities". Basically one thread is responsible for one entity. However, for simplicity reasons only the two entities mentioned in section 3.1.7 are used. Those are one main entity holding a render component with the scene graph and the camera entity. Since there are only few objects in the simulations it does not make sense to have more entities. Of the components that could be held by an entity the render component is the most important one for TealSim.

MTGame works with a global *world manager*. In Open Wonderland this world manager can be accessed as a singleton. Because it makes it easy to use in Wonderland later this system is used with TealSim as well. The *TealWorldManager* singleton class is defined in the JMonkeyEngine/MTGame specific package *teal.render.jme*. The class provides a *getWorldManager* method returning the single world manager. There is also a *setWorldManager* method which will be used in conjunction with Wonderland to set Wonderland's world manager to TealSim's. The world manager is used for accessing the managers and has got some other globally effective methods described later in this section.

Because MTGame always wants to keep the frame rate constant, one thread is dedicated for rendering the scene, namely the "renderer thread". In order to avoid concurrency errors all changes on the scene graph and its objects have to be done in the renderer thread. To give a job to the renderer thread the world manager contains an *addRenderUpdater* method which has two parameters. The first one is of interface type *RenderUpdater*, the second is an arbitrary parameter of type *Object*. The *RenderUpdater* interface defines a call-back method *update* with an *Object* as single parameter. Adding

such an render updater to the queue triggers the renderer thread to call the *update* method with the second parameter as *update* method parameter. To give a task to the render updater code similar to the following can be used:

```
Node node1 = new Node("test node");
final Vector3f offset = new Vector3f(0.5f,2f,0);

// placing update into the renderer thread
TealWorldManager.getWorldManager().addRenderUpdater(
                                        new RenderUpdater() {
  public void update(Object obj) {
    // cast because I need it as Node
    Node elem = (Node) obj;
    elem.setLocalTranslation(offset);

    // alerting update
    TealWorldManager.getWorldManager().addToUpdateList(elem);
  }
}, node1);
```

All ***final*** variables can ne used in the call-back code as well. It must be remembered that the code inside the *update* method is executed later than the code around. The world manager's *addToUpdateList* method is to be called on every change of a scene graph node or the scene graph hierarchy itself. Because the renderer code has a lot to do with drawing the scene, all the code which can be put into other threads should never be given to the renderer thread. Otherwise the frame rate may break down.

3.2 Preparing TealSim for Client-Server Architecture

Since Open Wonderland consists of client side code and server side code, TealSim will have to run on Wonderland as client-server software. Although the Wonderland specific code will be put into the Wonderland module (see section 3.3), some code-parts of TealSim might have to change. This section describes the ideas how the splitting into client and server parts can be made and what parts of TealSim were actually changed because of this. The synchronization of client and server is also a big issue which is addressed in this section. The Open Wonderland module should eventually scale up well and use as little bandwidth as possible. On the other hand all the shared data should be perfectly synchronous among all clients.

3.2.1 Synchronizing the 3D Objects

A part of TealSim which has obviously to be synchronized among all clients the server are the three dimensional objects. There are several possible ap-

proaches which level the 3D object can be synchronized at. Those different approaches are explained and then discussed with their advantages and disadvantages. Finally the decision for one of the approaches is pointed out. All the discussed approaches assume at least some code running on the server. If the whole simulation is run on the client the 3D objects will be there at all levels naturally and no synchronization would be needed. See section 3.2.2 for more details.

Synchronizing at Scene Graph Level

The lowest possible level to synchronize the 3D scene objects is the scene graph level. This means the whole scene graph would be serialized and transferred to the clients every time the graph changes. This assumes that the greatest part of TealSim runs on the server. Having the scene graph objects there would imply to have the whole engine, the rendered objects and the *TNode3D* objects at the server side. This restricts the possibilities discussed in section 3.2.2. A positive effect would be that not much of the code would have to be adopted. The scene graph would be copied, sent to the clients and displayed there. Since the viewer has a reference to the top scene graph node, it can either do the job of sending the scene graph or provide functionality to obtain the scene graph from it.

JMonkeyEngine provides a way to serialize the scene graph elements in different formats. The scene graph objects implement the following methods specified in the *Savable* interface:

```
void write(JMEExporter ex) throws IOException;
void read(JMEImporter im) throws IOException;
Class getClassTag();
```

The last method stated above simply replies the class of the object in most cases. The subclass of *JMEExporter* or *JMEImporter* defines weather it is worked with XML data or with binary data. With the *BinaryExporter* class the *Savable* object can be exported to binary format on the server, transferred to the client and imported to the client side scene graph with the help of a *BinaryImporter*. There would also be no problem if a newer version of Wonderland uses another graphics system. As soon as the graphic library uses scene graphs (which is to be expected from a graphic library) and the nodes can be serialized they could be handled as the JMonkeyEngine nodes now.

Perhaps the biggest problem with this approach is the high bandwidth usage. With JMonkeyEngine's importer/exporter system the traffic *might* be kept lower because the *read* and *write* methods are implemented such that

the object is saved with as little data usage as possible. A sphere primitive e.g. only stores its position, its radius and how many samples should be created in axial and radial direction. The simplest way would be to store all the vertices and normals of the object which would require a lot of data and therefore a lot of bandwidth when is has to be transferred. However, this leads to the utilization of computational power on the client sides, because such objects need to be reconstructed there with every frame. This leads to another question, namely "How often does the scene change?". A closer look at the code shows how often the new positions are calculated, namely 20 times a second. This number makes the whole approach impossible as soon as the network bandwidth is not much bigger than today. There would simply be too much traffic.

A way to overcome this problem is to transfer only the changes on the scene graph objects. For this purpose a protocol would need to be specified. The method called could be used. E.g. "*setLocalTransform* with parameter vector 4,7,2 called on object xy" could be transferred. The objects would need some kind of identification. It would not be enough any more to be informed that *something* in the scene graph has changed, but *exactly what* has changed. This is quite tricky and requires changes throughout TealSim's JMonkeyEngine specific classes. The independency of the graphic library would be lost because the messages for the updates would have to contain library specific data. If the next Wonderland version would use another graphic system or even another version of JMonkeyEngine the whole module would have to be rewritten and the communication between server and client would be very different.

All in all this method fails because of the high bandwidth usage and/or with the mentioned possible improvements because of the lack of flexibility.

Synchronizing at TNode3D Interface Level

The idea with this method is to have the scene graph on the client side. On the server there are only place holders. They are proxies telling the client side if they have to adopt something. If something happens on the client side (due to user interaction), the information will have to flow in the opposite direction. A sequence diagram of the creation of a 3D object and about setting parameters while the simulation is running is shown in figure 18. The $\rightarrow$ arrow indicates method calls between the objects whereas the $\rightsquigarrow$ arrow indicates a message from the server to the client or vice versa. With the latter calls there must be some messaging objects in between, i.e. the objects shown in figure 18 do not send the message but tell another object to sent the data to the client. On the client side there is another object to

receive the message and to tell the objects about it. This objects could also buffer data and send bigger packets to lower the network traffic.

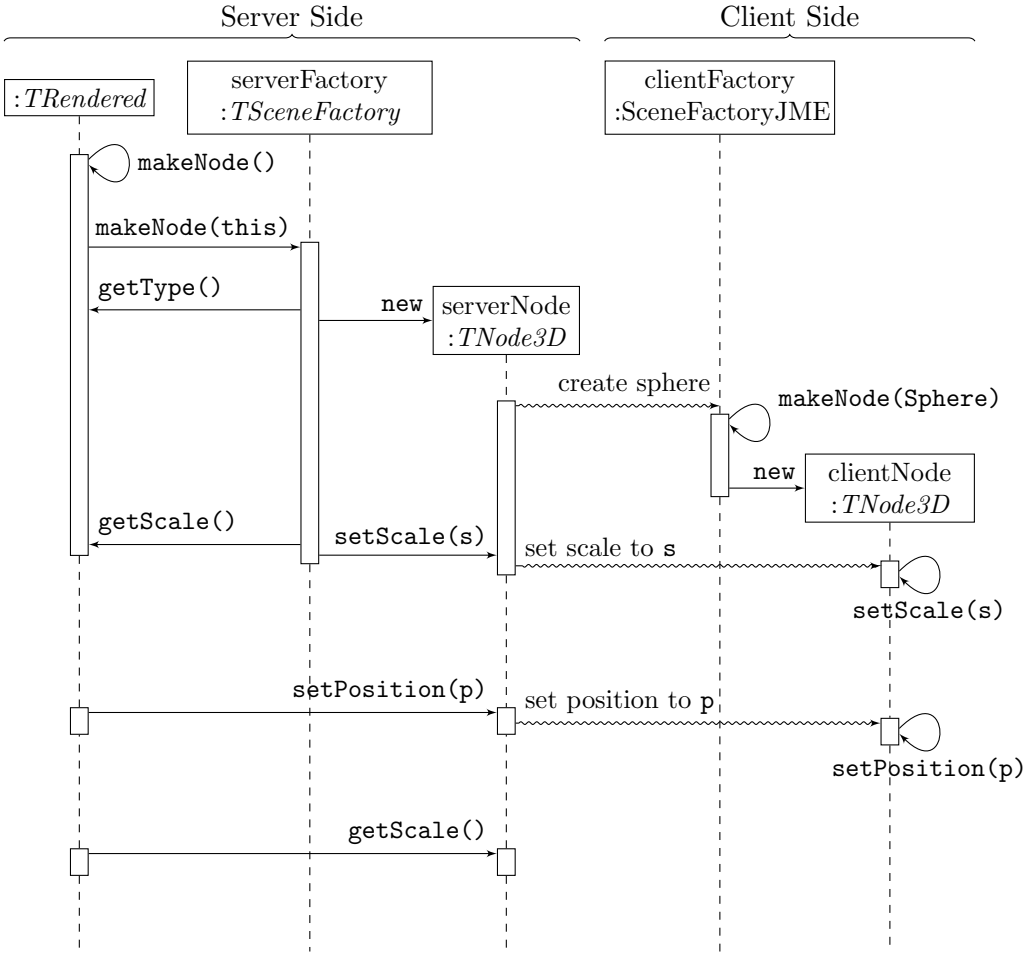

Figure 18: Sequence Diagram for creating 3D Objects and transferring changes at *TNode3D* synchronizing Level

Whenever a *TNode3D* object should be created the *makeNode* method of the rendered object is called. The *Rendered* class defines this method with ***protected*** scope. This method can be overwritten by the subclasses, but this is not necessary in most cases. The method makes a call to the *makeNode* method of the server side scene factory. There is one indirection step which for simplicity reasons is not shown in figure 18. As described in section 3.1.2, the specific factory is called by the general *SceneFactory* class. The server side factory would be defined inside the Wonderland module since it has nothing to do with the desktop version. After obtaining the type of the 3D object,this

factory creates the right proxy objects implementing the *TNode3D* interface as well as required additional ones (e.g. the *TFieldLineNode* for the field lines). The proxy objects are storing the current state of the objects, but they do not need to create any scene graph objects. Instead they (or some network communication object) send a message to the client saying that an object of the specified type should be created. On the client side the same factory as used in the desktop version, namely the *SceneFactoryJME* class could be used to create the scene graph object. The *makeNode* method with the specified type is called and a JMonkeyEngine scene graph node is created. Directly after creating the objects, the server side factory has to start to initialize them. The *TNode3D* object should exactly represent the rendered object. For that reason the server side factory has to obtain all the needed values. As an example, the scale value is obtained in figure 18. Several other values will usually be obtained in the same way. When the factory knows about the scale value, it calls the according method of the *TNode3D* interface on the proxy node. The proxy nodes stores the value internally and sends a message to the client. The scale is then applied to the actual JMonkeyEngine scene graph node on the client side.

After the creation face the setting of properties to the objects are performed in the same way. Since the server side node implements the sam interface as the client side node, the setter calls can be forwarded. The rest of the server side recognizes the server side node as if it was the actual node to be displayed. This would allow to keep the desktop version code pretty much as it is. As the synchronizing at scene graph objects level method the whole simulation would have to run on the server side. Only the graphic output is on the client.

The server-client communication can be reduced by storing informations about the scene graph node on the server side proxies. At the bottom of figure 18 the procession of a get method is shown. Since the proxy class stores the data of the client representation, the scale can be obtained from the proxy class directly. The rendered object does not need to know if it talks with a proxy class or with the "real" one. Since the getter methods are mostly processed on the server and the setter methods do not need any reply, the traffic will be mostly from server to client. Most clients are connected to the internet via asynchronous connection with higher download speed. Exactly this is what is required.

As soon as the system with the *TNode3D* interface is kept, synchronizing at that level would allow the change of other graphic libraries. The desktop version of TealSim would have to be provided with new implementations of the *Node3D* classes. Those could again be used with the Wonderland version on the client side. The viewer can be implemented as a proxy object in the

same way as the nodes.

The problem with the *TNode3D* synchronization approach is still the amount of data transferred. This is significant with only a view types of nodes. The node requiring perhaps the most traffic is the field line node. In most simulations a single field line consists of 200 data points and possibly as many color specifications. One point requires three floats. This sums up to 1600 bytes needed for a single field line. While a simulation is running, the field line data has to be calculated with every frame, i.e. 20 times a second. This leads to more than 31 kilobytes per second needed to be transferred during the simulation to each client for one single field line. Some simulations contain more than five field lines and if there are several users logged in the traffic on the server increases to an intractable amount. A field direction grid also produces a lot of data since the length and rotation have to be transferred for every single arrow. With the discussed approach only simulations which are not using field lines or field direction grids can be realized in Wonderland.

This problem could possibly be fixed by putting the *FieldLine* rendered objects to the client and let it calculate the field lines there. The same could be done with the field direction grid. However calculating fields is done with the objects of classes implementing the *GeneratesE* interface. All those objects will have to be on the client side if a field line or a field direction grid should be calculated there. For many simulations this leads to a situation where almost all of the rendered objects need to be on the client side as well as on the server side. It is also not a sign of good design if the synchronization is done on different abstraction levels for the same types of objects. This would be the case since the field lines and the field direction grid were to be synchronized at the rendered level whereas all the others were th be synchronized at the *TNode3D* level. The rendered objects needed for the field line calculation are to be synchronized at the rendered objects level as well. Subsequently, this leads to another approach.

Synchronization at Rendered Objects Level

With this approach no *TNode3D* objects are needed at all on the server side. Instead the rendered objects are kept synchronous on the server and on the client. Since the rendered objects represent the *Model* within the Model-View-Control (MVC) design pattern, the *View*, i.e. the *TNode3D* objects, can always be produced out of the model. The rendered objects in TealSim do not only contain data for the graphic representation but may also contain a lot of additional information. E.g. the *PointCharge* class contains a charge value and a radius additionally. Since it is a *PhysicalObject* it also inherits a mass, a velocity and many other properties. This makes the rendered

objects not only being the model for the graphic output, but representing an object as a whole. The simulation engine uses these objects as well for its calculations. Having all of the rendered objects on both the server and the client side would therefore also mean to be able to do parts of the calculations on the client and parts on the server. This requires synchronization between server and client. Since all the objects can be found on the client side, the field lines and the field direction grid can be calculated and displayed on the client side only. This overcomes the problem of the need to transfer all the data points of a field line. The needed computational power is also divided since the client and the server can do different tasks.

The approach would require to serialize the rendered objects in order to send them to the client. If calculations are to be on the server they have to be serializable anyway. Darkstar needs this behavior. However, transferring and distributing the serialized rendered objects with every change is not possible due to the big bandwidth utilization. Instead the objects can be transferred this way after they are created and be synchronized with update messages.

The dependence on the underlying graphic output primitive classes would completely vanish with this solution. The actual *TNode3D* objects only need to be created on the client side. If an other graphic library is used the *TNode3D* objects could be implemented for the desktop version of TealSim. Those objects could be used on the client side with the client-server version. However, there need to be much more code changes on TealSim than with the other approaches. Since the field line calculation have to be on the client side, TealSim's physics engine has to be split anyway (unless all the whole application is run on the client). The currently discussed solution seems to be the only one possible.

3.2.2 Splitting the Simulation Engine

This section describes the changes made on the simulation engine part of TealSim to prepare it for working as client-server version. First the engine as it was prior to the changes is described. After that the changes made are discussed. Alternative approaches are also stated and discussed.

Engine functionality

The simulation engine is capable and responsible for the simulation calculations. It is given a list with all *TSimRendered* elements from the simulation. When the simulation is running the internal data of the *TSimRendered* elements is calculated 20 times a second. Several types of forces are taken into account for that. The simulation runs in four steps with the according

methods implemented:

- *doReorder*,
- *doDynamic*,
- *update*,
- and *doRefresh*.

The *doReorder* step resolves all the collisions occurring between the objects at the beginning of the step. The objects rearranged are those who have implemented the *HasCollisionController* interface. With the help of the collision controller those elements are repeatedly reordered until there is no collision any more. Some of the elements need to be told if the positions have changed after the collision resolving step. Those can implement the *TUpdatable* interface. The only method declared in this interface is the *update* method, which will be called on each *TUpdatable* object at the end of the doReorder step. Usually the update call is used to set internal properties, e.g. the position, to a previously calculated shadow value. Further information about this values are given in the doDynamic step paragraph.

In the *doDynamic* step the actual integration of the objects occurs following the laws of physics. Only objects implementing the *Integratable* interface are touched during this step. The mentioned interface declares the following methods:

```
public void getDependentValues(double[] depValues, int offset);
public void getDependentDerivatives(double[] depDerivatives,
                                    int offset, double indepValue);
public int getNumberDependentValues();
public void setDependentValues(double[] depValues, int offset);
public boolean isIntegrating();
public void setIntegrating(boolean b);
```

All the descriptive data about an object needed in the integration step can be obtained by calling the *getDependentValues* method. Those values are e.g. x, y and z coordinate of the position, charge values, the velocity and so on. All of this values put into the *depValues* array. The *offset* parameter specifies where the integratable object should start to put the data into the array. The engine uses a single double array to store the dependent values of all the integratable objects. The *getDependentDerivatives* method returns the derivatives of the dependent values. They are needed in the engine for the integration. Since the engine stores all the dependent values into a single array it needs to know how much space to reserve within the array. This value can be obtained by using the *getNumberDependentValues* method. After each

loop within the integration the engine calls the *setDependentValues* method of the integratables. Finally, the object can be informed by the engine if it currently integrating using the *setIntegrating* method. This state can be obtained with the *isIntegrating* method. Internally the values obtained and written back to the integratable objects are not directly applied, but stored in so-called "shadow values". Before the integration step those values are the same as the actual values. Within the integratable class, shadow values are indicated with a *_d* or sometimes with a *_c* for shadow value prior to collision. As an example, the actual mass value of the class *PhysicalObject* is stored in the member variable *mass* whereas the shadow value is stored in *mass_d*. This way the whole integration process with several *setDependentValues* calls can happen without influencing the actual values.

To tell the integratables to apply the shadow values to the actual ones, the update phase calls the update method of each *TUpdatable* object. All the integratable objects should also implement that interface. The objects should always check if the dependent value is different from the actual before setting it. This can safe runtime.

The last step is the *doRefresh* step. In that phase the *nextSpatial* method of all objects implementing *IsSpatial* is called. The field line e.g. is such a spatial. On the *nextSpatial* call the field lines with all its data points is calculated. After that the rendering is triggered by calling the *render* method on the objects implementing the *TRenderEngine* interface. Currently this is only one object namely the viewer. It then causes the rendered objects to be rendered.

A part of the class hierarchy of TealSim prior to the changes is shown in figure 19. There is a class hierarchy of two engine types. The *SimEngine* is the base class and the *EMEngine* extends the functionality for electromagnetic simulations. One of those two engines is created when the simulation is loaded by the *SimPlayer* (see section 2.2.1). The simulation itself contains the information which engine it will need. The *getEngineType* method declared in the *TSimulation* interface returns an integer. The value responds to constant integers defined in the *TEngine* interface. If the integer indicates an electromagnetic engine (with value *TEngine.EM_ENGINE* an *EMEngine* class object is created; an *SimEngine* object otherwise. The created engine is set on the engine control. This engine control represents the user interface to the engine. It is a swing component showing buttons like "play", "stop" or "pause" on the user interface. When the user clicks on such a button the according method declared in the *TEngineControl* is called on the engine control which forwards the call to the engine. On the first call the simulation is started. Since the engine implements the *Runnable* interface, it can be run in its own thread. The engine control starts this thread and stores the thread

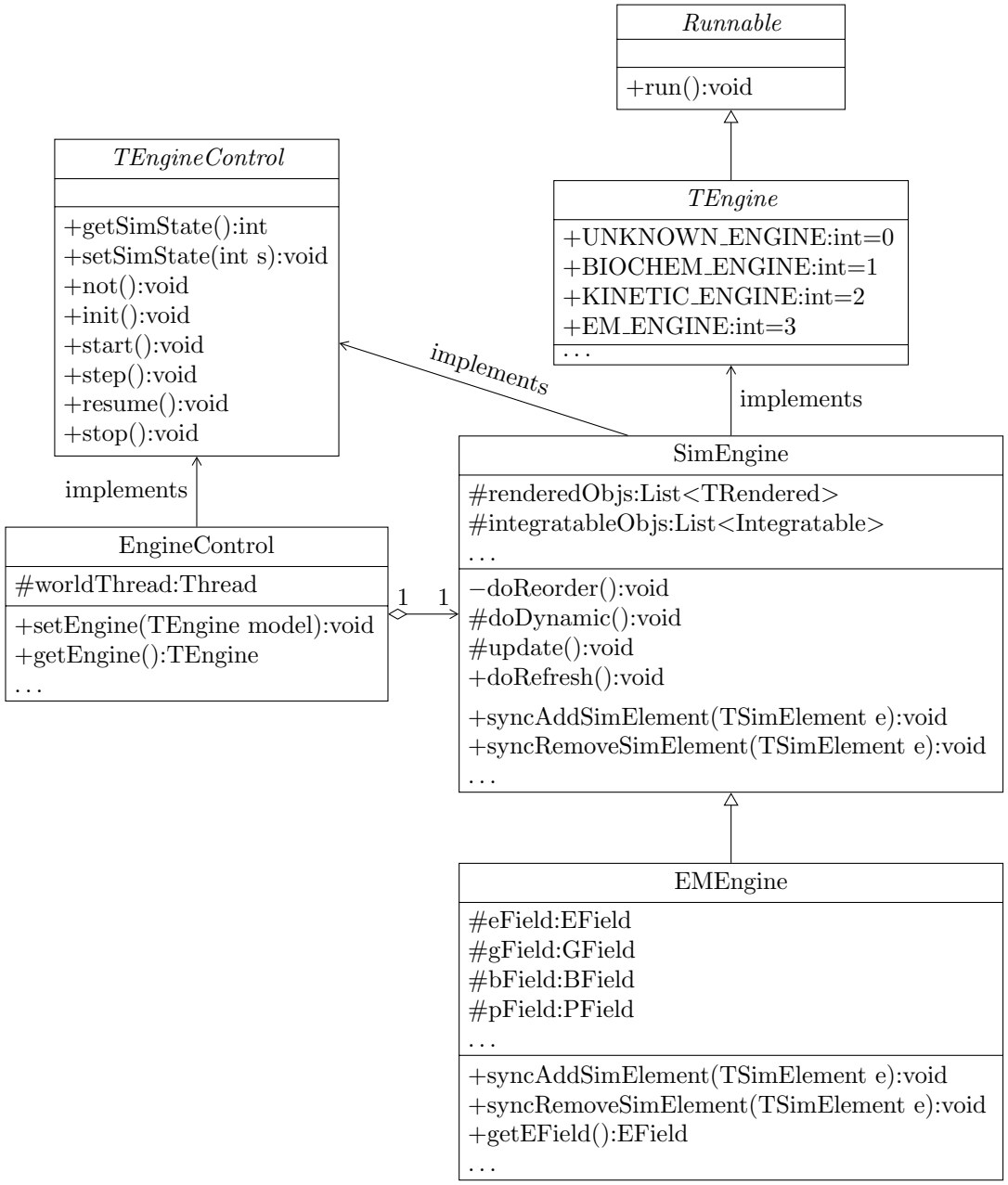

Figure 19: Previous Simulation Engine Class Diagram

object as the member *worldThread*. With the threads *isAlive* method the engine control can always check whether the engine is still running.

When the new thread is started the *run* method of the simulation engine is called. Within this method, there is a loop doing different things depending

on the simulation state. When the simulation should run the methods for the four steps are called one after the other. As shown in figure 19 the visibility for those four methods are different. The *doReorder* method is ***private***, the *doDynamic* and *update* methods are ***protected*** and the *doRefresh* method is ***public***. This indicates workaround changes on the code. With a clearly defined interface the modifiers would not be that different.

Every step should take around about the same amount of time. This time can be specified by the simulation. By default it is 50 milliseconds. It there is still time left after the four simulation steps the rest of the time is waited with a sleep call on the thread. The engine contains methods to add new elements. Those can be called while the simulation step is executed. In this case the elements are stored in a temporary list and are then added or removed after the calculations. This avoids concurrency problems with the element lists. If the engine is not in the running state only the *doReorder* step and the *doRefresh* step is executed. The different states are defined as constant integer flags in the *TEngineControl* interface.

The electromagnetic simulation engine adds functionality to the simulation engine top class. It stores some data in its members so they can be accessed with additional methods. E.g., there are four additional members for different fields:

- the *eField* for the electric field,
- the *gField* for the gravity field,
- the *bField* for the magnetic flux field and
- the *pField* for the Pauli field.

All those members have the according getter methods (e.g. *getGField* for getting the gravity field) which are called by the simulation elements if they need them. The field classes contain all elements generating the fields. Those are for example *GeneratesE* elements for the electric fields. The calls for adding and removing elements to the engine are overwritten in the electromagnetic engine. If the element is of interest for the electromagnetic engine, it is processed. E.g. if the element added is an object implementing *GeneratesB* it is added to the magnetic field. The add method has to call the implementation in the superclass so the general simulation engine can process it. When an element is added, it has to be put into the corresponding lists. A point charge e.g. is put into the list with the integratables, the rendered objects, the updatables and so on. Referencing the objects in different lists by type increases the performance of the engine. This is because with most operations all elements of a specific type are needed. If they are all in the same list

it can be stepped through this list and the operations can be performed with every single element of the list. If all elements were only stored in one single list the engine would have to step through all elements for each operation and check whether the element is of the required type. Instead the checking is done only once for each element when it is added.

Design of a new class hierarchy

The design allows to add new types or subtypes of engines by inheriting from the existing ones. However, if it should be used on a client-server system this becomes tricky. If some parts of the engine should run on the server it would run on darkstar. This requires some restrictions for the code (see also section 3.2.3). Synchronization among darkstar managed objects is done exclusively by darkstar. If the engine contains synchronization mechanisms, e.g. ***synchronized*** blocks, deadlocks will occur. Figure 20 shows a possible class hierarchy if the client-server functionality should be added

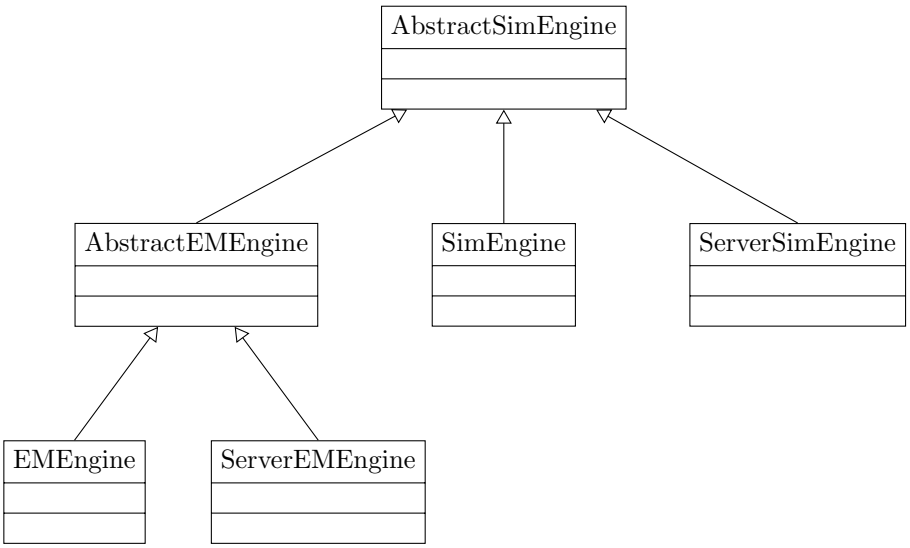

Figure 20: Extending Desktop Version by Inheritance

with additional inheritance classes. The *AbstractSimEngine* class represents the most abstract engine class. It contains the code needed in all the subclasses. The class must not contain any ***synchronized*** blocks since the server side engine classes which are inherited will run on darkstar. Similarly to the previously used model the *AbstractEMEngine* class is inherited from the *AbstractSimEngine*. Those two classes represent the simulation engine and the electromagnetic simulation engine. To extend the functionality the desktop versions and the server side versions of the engines are now

added. The *SimEngine* class is basically the same as the *AbstractSimEngine* but with the synchronization functionality. It is the simulation engine for the desktop version of TealSim. Similarly, the *ServerSimEngine* is inherited from the base class. It contains the code for the simulation engine on the Wonderland server. In order to add the electromagnetic versions of the engines the *EMEngine* and the *ServerEMEngine* class are inherited from the *AbstractEMEngine*. The two server engines can be implemented within the Wonderland module. This way the Wonderland part would remain inside the module and outside of TealSim.

However, this approach comes with many disadvantages. First of all it is not easily extendible. If a client engine would be needed two different classes would have to be added. One for the general simulation engine and one for the electromagnetic simulation engine. This code is very likely to be very similar if not identical. Thus, the code would be copied from one class to the other, which is very bad for the maintainability. If changes are to be made on that code the changes would have to be applied to both added classes. E.g. the *SimEngine* class and the *EMEngine* class in this model would likely already have duplicated code. If a new engine type, e.g. a kinetic engine should be added those problems occur again.

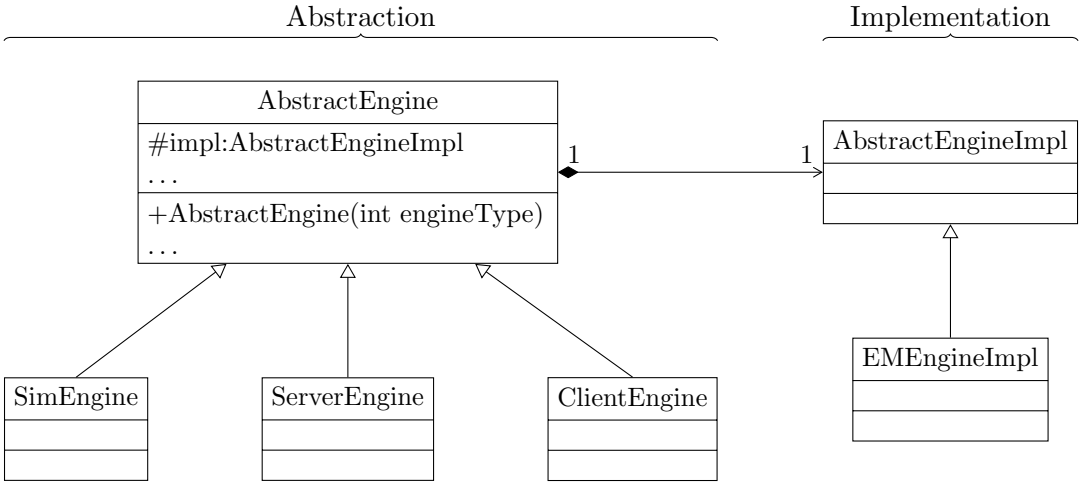

Figure 21: Engine class hierarchy using Bridge Pattern

Having a closer look on the problem it can be seen that there are two abstraction levels in the code hierarchy. The engine type (general and electromagnetic) and the application type (desktop version and Wonderland server side version). In [5] exactly this problem is addressed. The solution given is to make two hierarchies out of the single one and define one of them as the abstraction and one as implementation. Figure 21 shows the resulting

class hierarchy when the bridge design pattern is used. There are two class hierarchies now. On the left there are the different application types, on the right the engine types. Having two hierarchies makes extending easy. If a client engine is needed it is added to the left class tree. If a new engine type is needed the right class hierarchy is extended by inheriting from the *AbstractEngineImpl* class, or it is a specialized electromagnetic engine from the *EMEngineImpl* class. This does not only make the code extendible but also avoids duplicated code as with the hierarchy shown in figure 20. The Wonderland specific parts which are *ServerEngine* and *ClientEngine* can be put into the Wonderland module. The rest belongs to TealSim and will be implemented there.

From a logic point of view it makes more sense for the engine type being the abstraction and for the type of application being the implementation. This way there would be a desktop implementation, a server implementation and so on. In our case this is hardly possible because of the synchronizing behavior. Since the server parts of the code should not contain any ***synchronized*** blocks those blocks would only be possible in the implementation. A lot of the methods in the desktop version of the engine need to be synchronized. That's why there would not be much code in the engine specific hierarchy. For that reason the decision was made pro the solution shown in figure 21.

The *AbstractEngine* class provides a constructor with the engine type as parameter. According to this type the right implementation is instantiated. E.g. a desktop version of an electromagnetic engine can be created with:

```
TEngine engine = new SimEngine(TEngine.EM_ENGINE);
```

Of course the abstraction has to implement all the interfaces the previous version of the engine was implementing. *TEngine* is one of them. The only exception for this is the *Runnable* interface. The *TEngine* interface was derived from this interface (see also figure 19). This is rather desktop version specific. Is was used for running the engine's main loop inside a new thread. This can be very different with the server engine or the client engine. Inside the constructor the engine instantiates the implementation according to the type:

```
switch (engineType) {
  case TEngine.BIOCHEM_ENGINE:
  case TEngine.KINETIC_ENGINE:
  impl = new AbstractEngineImpl();
  break;
  case TEngine.EM_ENGINE:
  impl = new EMEngineImpl();
  break;
```

```
    default:
    impl = new AbstractEngineImpl();
    break;
}
```

This can be implemented in the base class of the abstraction. To assure the creation of the implementation the *impl* member must be instantiated in every defined constructor.

The implementation needs to have *one* interface which is used by the abstraction. This interface is defined by the *AbstractEngineImpl* class itself. Since there is only one specific implementation type only this type needs to be taken into account so far. If there should be more that the electromagnetic implementation supported in the future some code has possibly to be moved from the abstraction to the implementation. The original electromagnetic engine supports retrieving some additional objects compared to the simulation engine base class. This is the only functionality specific to the electromagnetic engine. There will have to be a method for adding and removing objects to the implementation. This methods are called by the abstraction when an object is added to or removed from the engine. The abstraction will handle the adding specifically to the application types and the forward the call to the implementation. In case of an electromagnetic engine implementation the field generating objects will be added to the according field. A list of objects implementing the *PhysicalObject* class also needs to be held by the *EMEngineImpl* class. The *AbstractEngineImpl* class does not need to do anything on the add and remove objects call since the implementation only handles the differences between an electromagnetic engine and an other engine.

The additional getter methods of the previous *EMEngine* are:

```
public CompositeField getEField();
public CompositeField getGField();
public CompositeField getBField();
public CompositeField getPField();
public Collection<PhysicalObject> getPhysicalObjs();
```

The *CompositeField* class is the common base class of the four field types. Since this methods are specific for the electromagnetic engine only they should not be provided any more by the abstraction but should be replaced by more general methods. In this case getting elements by their type is introduced. With such an approach the electromagnetic engine specific types can be obtained as well as many others. One way of defining such a method is to have the getter method with a Java *Class* as parameter. The method declaration in this case looks like this:

```
public <T> T getElementByType(Class<T> type);
```

This way Java's generics are used for returning an object of correct type. Here is an example how the call for the electromagnetic field *getEField*, previously located in the *EMEngine* class would be replaced:

```
EField field = theEngine.getElementByType(EField.class);
```

Since the call returns the same type as specified as parameter a cast to *EField* is not needed for this assignment. The implementation knows what element to return on a specific type request in this case. Whenever the abstraction asks the electromagnetic engine implementation for an element of type *EField* the *eField* member variable of the implementation will be returned. For all variables located in the abstraction, the *AbstractEngine* object can directly return the desired object. For object lists another interface method is defined for the engine abstraction:

```
public <T> Collection<T> getCollectionByType(Class<T> type);
```

This works similarly to the getter method for single objects. The method returns a collection with objects of the class type specified in the parameter. This way the last method specific to the electromagnetic engine, the getter method for the physical objects is covered. However, this approach has limitations. If there is more than one objects of the same type they can not be obtained this way. This is likely to occur with the primitive types and with others which can be used for several purposes. The *Vector3d* class as it is used for storing the gravity is such an example. To overcome that problem a method for getting elements per type with an *enum* as parameter can be used:

```
public Object getElementByType(EngineElementType type);
```

This has the additional advantage of being faster than the other approach. When choosing to return of the right object a ***switch*** statement can be used. One and the same method can be used for single objects and collections. The disadvantage of this method is that it can not make use of Java's generics. Thus, the return type has to be of class *Object* and the calling method needs to cast it to the actual object type. This makes the calling code longer and it is not as readable any more. Since both approaches have advantages and disadvantages, both of them were added to the *AbstractEngine* class.

The interface of the implementation is still to define. It could be defined with exactly the same method as the abstraction, namely *getElementByType*. First, the abstractions method would check whether it has got the requested

object on its own. If not, it would forward the call directly to the implementation:

```
public <T> T getElementByType(Class<T> type){
  if(type == Vector2f.class)
    return (T) gravity;
  else if (type == double.class)
    return (T) damping;
  // ...
  // several other else if's to be added here...
  // ...
  else
    return (T) impl.getElementByType(type);
}
```

The according implementation's method would look the same without the forwarding call of course. For simplicity reasons the code above contains no error handling e.g. if the type is unknown. A problem with this approach can be found when it comes to performance measures. This getter methods are called very often during a simulation step. For that reason they must be as efficient as possible. With the currently discussed approach the ***if*** - ***else if*** clauses are evaluated one after the other until the needed return value is found. If it is the last one it will take a while to evaluate all the if clauses. This can be improved by putting the if clauses questioning for more often used types first. However this method is limited because it often depends on the simulation how often a specific type is requested and the simulations should be supported as equally as possible. With *enum*'s the performance can be optimized better because the ***switch*** statement which is much faster can be used. However, there have to be two ***switch*** statements, one in the abstraction and one in the implementation.

Another approach is to use maps for getting the requested object by a key value. This key would be the class type or the enum value, respectively. Java provides a *HashMap* class returning elements in logarithmic ($O(\log n)$) time and an *EnumMap* class especially for enums returning elements in constant ($O(n)$) time. The implementation can now provide getter methods returning such maps. Those maps can be merged with the type maps of the abstraction. This way only one enum map and two hash maps with classes as key (for single elements and collections, respectively) would be stored within the abstraction. Every call for a type on the abstraction would be one lookup in the according map. The maps can only change whenever an object is removed or added to the engine. Such updates are processed prior to every step, which means that type map updates can not occur within the calculation step. This enables the engine abstraction to cache the implementation's list for the whole period of a calculation step. The implementation has to

provide a method for obtaining the maps which is then called by the abstraction, possibly at the first call of a step. However, due to lack of usage of the getting of typed objects specific to the abstraction so far, this speed improvement is not implemented yet. The abstraction provides the by-type getter functionality only with a few object. For that reason it still works with the ***if*** - ***else if*** clauses and the *select* statement, respectively. The engine implementations though provide the interface using maps. The complete interface of the implementation looks as follows:

```
// getter methods by class type
public HashMap<Class<? extends Object>,? extends Object>
        getTypedElements();
public HashMap<Class<? extends Object>,
                Collection<? extends Object>>
        getTypedCollections();

// getter methods by enum type
public EnumMap<AbstractEngine.EngineElementType, Object>
        getTypedElementsEnum();

// adding and removing elements
public void addSimElement(TSimElement obj);
public void removeSimElement(TSimElement obj);
```

With this simple interface all engine type specific functionality is covered and the design can be kept as it is. After checking if the type belongs to itself the engine abstraction can return the requested element by obtaining the according map from the implementation and the obtaining the element from the map. Here as an example the getter code by class type for collections:

```
public <T> Collection<T> getCollectionByType(Class<T> type) {
  if (type == TRenderEngine.class) {
    return (Collection<T>) renderEngines;
  } else {
    // ask implementation for the type
    return (Collection<T>) impl.getTypedCollections().get(type);
  }
}
```

To change the code towards using one merged map for the performance improvements mentioned earlier only minor changes in the code are necessary.

Separating Server and Client engine

The next question is how much of the functionality of the engine to put on the client and server side, respectively. If the whole simulation is done by the client, not much simulation data has to be transferred, because it is generated

on each client. But the more happens on the client, the harder it gets to synchronize among the clients. The engine state with all the objects needs to be the same on each client. The user interactions would have to go through the server to be applied on all clients synchronously. This would require some adoption to the engine code. If the whole simulation is calculated on the server the synchronization among the client is reduced to a minimum. Only the results have to be sent to the clients. As it is discussed in section 3.2.1 this is not possible because of the high network bandwidth utilization with the field lines and the field direction grid. Those elements have to be computed on the client side anyway. When they are computed on the client side they do not need to be perfectly synchronous among all clients. The field lines (as well as the field direction grid) is only a representation of the field. If that field is kept synchronous (this must be the case since all the charge values, positions and so on are) the field lines and the field direction grid will be synchronous as well automatically. Calculating things on the client also splits up the computational power between the client and the server. This saves CPU utilization on the server. All the code running on the server side has to run on darkstar and therefore be adopted as described in section 3.2.3. There is also the possibility of writing a Darkstar service which would overcome that problem. However this approach comes with many disadvantages. Running a service enables to run code "parallel" to the tasks in Darkstar. In this case the thread synchronization must be done by hand. The scalability provided by Darkstar is also in danger because the service may not be restricted in the use of computational power on the server. In case of Wonderland the whole virtual world could be slowed down. For that reason this approach was not followed, so all the engine code running on the server has to be made compatible with darkstar. One calculation step of the server side engine can be processed as a Darkstar task. If the engine is in the running state the task can be scheduled repeatedly. Darkstar provides a functionality to schedule tasks in such a way. The time between the executions as well as the waiting time until the task is executed for the first time can be specified. The server side engine class could implement the *Task* interface which similarly to Java's *Runnable* interface declares a single *run* method. This method is then run when a scheduled task is started.

To estimate the degree of how much engine code needs to be run on the server a closer look on some specific simulations is necessary. Unfortunately some of them are not deterministic. The "Capacitor" simulation (see also figure 11) for example starts with randomly distributed charges. During the simulation charges can be added or removed. The less deterministic simulations are the harder is it to synchronize them among clients. To avoid that problem as much of the engine as possible has to run on the server.

As mentioned above it is no problem to put the field line and direction grid calculation to the client. It is even possible to not calculate these field depending shapes synchronously to the other calculations if the client is too slow for such frequent calculations. This is because no other calculations are dependent on field lines of field direction grids since they are only a visualization of the field produced by the other elements of the simulation.

It turns out that the calculations which have to be done on the client happen inside the *doRefresh* step. This is the last phase of the engine calculation and calls the *nextStep* method of all the objects implementing *isSpatial* interface first. Those objects are mainly the field directions grid and the field lines. Secondly the doRefresh step calls the *render* method on the viewer indicating that the scene can be rendered. The whole doRefresh step has to be put into the client engine.

As mentioned earlier the *doDynamic* step does the integration which is the main calculation phase of a whole engine step. For synchronization reasons mentioned above, this will have to happen on the server. The *doReorder* step can be seen as a preparation step for the integration and will therefore happen where the doDynamic step happens; on the server. The *update* step is to write back the shadow values into the real value member variables. This has to happen prior to the doRefresh step since this phase may need the values. Because the doRefresh phase does not need to be executed on the server the update phase can also be skipped here. After the integration the clients have to receive the values and can then perform the update and doReorder step. Fortunately, the data needed to be sent to the client is already there and used in the doDynamics step. All the changed data is contained by the dependent values. A simple *getDependentValues* call on the engine can retrieve all the shadow values as a single array of doubles, this array will be sent to the clients. The client can then call *setDependentValues* on the client engine to set all the shadow values on the client side to the ones just calculated on the server. The order of the *Integratable* elements in the according list in the engine plays a role in this case. Since this order is the same as in the array with all the dependent values, the order on the clients *Integratable* list must be the same as on the server. This can be achieved by sending the order to the client. It is only necessary to synchronize that order whenever an object is added to the engine. After setting the shadow values of the client engine the *update* step can be performed to set the actual property values of the elements to the shadow values. Subsequently, the client engine is ready to perform the *doRefresh* step in order to calculate the field lines or perform other actions defined in the *nextStep* method. As a last step the client engine can trigger the rendering.

The whole process of a simulation step is shown in figure 22. The net-

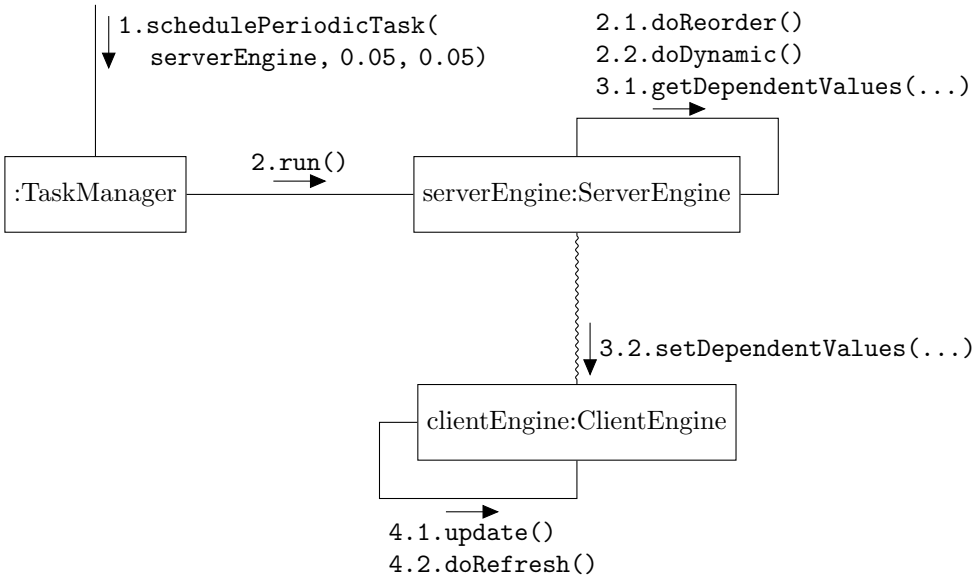

Figure 22: Communication Diagram for Engine execution step

work communication is simplified and replaced by a ~~ line. First the task manager is called and told about the *Task* object (which in this case is the server side engine), when to start the the first execution and the time between the executions. The task manager triggers the run method of the *Task* objects periodically. Within the server side engine a call on run triggers the *doReorder* and the *doDynamic* step. When those steps are finished the data is sent to the client by obtaining it with a *getDependentValues* call on the server engine and transferring this data to the client engine. This engine is now updated with the *setDependentValues* call. Subsequently, the *update* and the *doRefresh* phase are performed on the client by calling the according methods of the client side engine.

With the discussed method *all* dependent values are transferred to each client every simulation step. This can lead to a high bandwidth utilization at the server. E.g. for the capacitor simulation there are 24 point charges added per default. Each charge has got three dependent values for the position and the velocity, respectively, and one for the charge. Since all of these seven values are 64 bit doubles the transferred data is more than 10 kilobits per step per client. With 20 steps per second this value multiplies up to more than 200kbps, which is quite a lot. In order to reduce this amount some methods are possible:

- Using floats instead of doubles for the dependent values. This would reduce the amount of data needed to be transferred to 50%. If the

engine still uses doubles internally the double array would have to be converted into a float array which would take too much time. Another possibility is to use floats in the engine instead of double. This can possibly lead to rounding errors.

- Transferring only the changed data. Unfortunately it is not easy to determine the values which have changed. This can possibly only be done with comparing each value to the previous one. Another problem would be that additional information would be needed about what values are skipped for transfer. A simple *setDependentValues* on the client side would not ne possible any more since some of the values are in the array and some are not.

- Leaving out values which are not needed on the client side. There are several dependent values which are not needed on the client. With the point charges for example those are the velocities. The position is needed for rendering and the charges are needed if field lines have to be calculated. It is not easily to determine if a value is needed on the client. This is sometimes simulation specific and would therefore need specific adoptions for single simulations. As it is with the transferring of only changed data it would not be possible to simply call the *setDependentValues* method.

The three methods could be combined arbitrarily. Because of their drawbacks in terms of performance and the increased complexity none of these methods were implemented. A test with several clients and different applications to test the bandwidth was successful.

3.2.3 Preparing TealSim for the Project Darkstar Server

All the code running on the server needs to be ready for darkstar execution. In order to run on darkstar the code must fulfill some requirements. Darkstar's first priority is to make client-server applications scalable. For this purpose the execution is split up into tasks. Every task has got a certain amount of time to be executed. If it has not finished after that time it is interrupted and rescheduled. Splitting the execution into tasks avoids having a single task blocking the whole execution. A task is always run as an atomic state change. If it is interrupted the state prior to the task execution is reestablished. That functionality is implemented in darkstar using serialization. The objects are arranged to logic groups, each of them represented by a *ManagedObject* class object. *ManagedObject* is a marker interface. Classes implementing this interface are serialized and de-serialized at once

with all of their members. Darkstar holds all managed objects in their serialized form. If a task needs data of managed objects this objects will be de-serialized. Synchronization between the managed objects is done by darkstar. If a task tries to write to a managed object while another task accesses it the task is thrown away and rescheduled for later execution. Reading one managed object by more than one task is allowed of course. After a task has successfully processed to its end the used managed objects are stored and are replacing the previous versions of the objects. If a task can not be completed the previous versions of the managed objects are kept and can be de-serialized again if another task (or the same rescheduled task) require it. Code run on Darkstar has to follow several guidelines:

- All objects must implement the interface *Serializable*. Without that the mentioned atomicity of a task can not be provided. Darkstar throws an exception if an object not being serializable.
- A single managed object must not contain too much data. Otherwise the de-serialization and re-serialization process would take too much time and the task will be thrown away very often.
- All inner classes should be ***static*** since the time taken for the serialization increases significantly if they are not static.
- Synchronization blocks must not be used among managed objects and their members. Since Darkstar uses it's own locks those can conflict with the ones the user defined code uses. This can easily lead to a deadlock.
- Static fields which are not constant vanish on re-serialization. Although this problem can be solved with Java semantics another problem with this fields appear. Such fields are specific to a single Java virtual machine. This behavior undergoes the feature of Darkstar to run on more than one virtual machine.
- No *java.lang.Exception* should be caught. This is because Darkstar uses its own exceptions which would be caught by the user code. This is especially important for debugging and testing new functionality since the exception base class is often used together with such approaches.
- No objects except those implementing Darkstar's *ManagedObject* interface should be referenced by more than one managed object. After the first serialization process they will not be identical any more since a new object is created on re-serialization.

Most of the problems are relatively easy to overcome in TealSim. The Java's exception base class should be called very rarely in Java code anyway. Non-constant static fields are sometimes used for singleton objects. For such purposes darkstar provides functionality to bind an object to a name, i.e. a Java *String*. However, Darkstar code should not be put into TealSim. Such code parts could be replaced in Wonderland with classes defined in the Wonderland module. Fortunately this was not necessary since all the non-constant static fields used in the server side could be removed anyway.

To pull ***synchronized*** blocks out of the code is rather difficult with existing software. Most of the synchronized blocks needed on the server side occur in the original engine. With the new engine class design explained in section 3.2.2 those ***synchronized*** blocks needed for the desktop implementation are put into the desktop version of the engine. The base class which the server side engine is derived from does not contain any synchronized code.

Non-static inner classes can access the members of the parent class. This works because they hold a reference to them internally. They can be replaced by static inner classes where these reference to the parent object is set manually. This makes the code a little more complicated but makes the serialization which is needed on the Darkstar server much quicker. All the non-static inner classes on the server side were replaced by static ones for that reason.

Since darkstar must not be put into TealSim, the *ManagedObject* interface must not be used there. "Normal" objects may later be wrapped by Darkstar's *ManagedSerializable* class within the Wonderland module. As all the other objects on the server side those must implement the *Serializable* interface as well.

To make all the needed code serializable usually forces a lot of code changes. However, this is not too bad with TealSim because most of the classes were actually designed to be *Serializable* . A reason for that is that the functionality to save the current simulation state needs the elements to be serialized. The swing components for the user interface are *Serializable* anyway. One problem with serializability occurs with the bounding volumes. Those are not only needed in the low level graphics but also within the simulations and at several other parts of the code. Originally the Java3D bounding volumes were used, but those do not implement the *Serializable* interface. For that reason manually implemented bounding volumes are replacing the Java3D ones outside the low level graphic packages.

With another class of Java3D used in TealSim the same problem was addressed differently. The *Transform3D* class occurs only two times in TealSim outside the Java3D specific package. Therefore it was kept and the serialization was done by Java's object serialization customization. Normally Java

stores all non-transient member objects of an object recursively on serialization. This default behavior can be overwritten by implementing the following two methods in the class which should be serialized manually[6]:

```
private void writeObject(java.io.ObjectOutputStream s);
private void readObject(java.io.ObjectInputStream s);
```

Those methods can be made ***private*** because they are called by the virtual machine directly. Upon serialization the *writeObject* method is called. It usually calls the object stream's *defaultWriteObject* method first. This method does everything which would be done without defining a *writeObject* method. After that some extra information needed to restore the object properly is added to the output stream. On de-serialization the *readObject* method usually calls the input stream's *defaultReadObject* to restore all non-transient members. Subsequently the additional informations stored by the *writeObject* method are retrieved and used. With the classes holding a *Transform3D* object which is not serializable, this member is made ***transient*** first. This tells the default reader and writer methods to ignore that member. A Java3D transform can be represented by a transform matrix with 16 double values. The *writeObject* method of the containing class obtains these 16 values and writes them into the object stream. Those values can be read by the *readObject* method on de-serialization. With the 16 doubles a new *Transform3D* object can be created after that in order to restore the containing object *with* the transform. If the transform object can be ***null*** this information has to be stored within the stream additionally.

In some cases members of an object do not need to be serialized at all. This happens e.g. if a member is only used as cache. TealSim's *Route* class contains a target object as well as a string with the target property. With a Java *PropertyDescriptor* getter and setter methods can be obtained. If not stated otherwise the property descriptor assumes the setter method to be *setPropertyName* and the getter method to be *getPropertyName*. E.g. if the property is "mass" the getter method should be *getMass* and the setter method should be *setMass*. Using a property descriptor's *getWriteMethod* method the right *Method* object according to the target property and the target class is created. This *Method* object is saved into a member variable of the *Route* object. Upon request the method is invoked on the target object. Because the target object and the name of the property is enough of information the method does not need to be stored permanently. However it is cached for performance reasons as long as the *Route* object is not being serialized. On serialization it is thrown away by declaring the member variable ***transient***. When the *Method* object is needed for the first time after re-serialization it is recreated. At other parts of the code where a *Method*

object was used as a member variable to point to a method of its own class it was replaced by storing the method as a string.

3.3 Wonderland Module

After having the desktop version of TealSim run with JMonkeyEngine and MTGame a Wonderland module was implemented. As mentioned in section 2.1.1, pluggable modules can be constructed and implemented for Open Wonderland. The goal was to put all the code needed with a jar file including TealSim into a single Wonderland module. In general a module consists of the following components:

- Server side code put into a *server* package,
- client side code put into a *client* package,
- code to be distributed to both, client and server put into a *common* package
- and artwork put into the "art" directory.

The whole module is zipped into a single jar file consisting of a server side jar and a client side jar. The server side archive contains all classes in the *server* package and in the *common* package, whereas the client side archive contains the *client* and the *common* package as well as the artwork. For the TealSim module the needed TealSim classes and their dependencies are needed as well. To add them a new folder "lib" is added to the module. TealSim is added to this directory as a `jar` file including all needed classes. The subsequent sections explain the implementation of the module.

3.3.1 Preparing the Module's Environment

Wonderland modules are usually built with an ant[15] file called `build.xml`. In order to pack the TealSim `jar` file into the module this build file has to be adopted. First the path to the needed jar files is given using the *pathconvert* option:

```
<!-- Client side jars -->
<pathconvert property="module-client.classpath">
  <path location="${module.libdir}/${module.client.jar}"/>
  <path location=
    "${appbase.dir}/build/client/appbase-client.jar"/>
  <path location=
```

[15] `http://ant.apache.org/`

```
    "${appbase.dir}/build/client/appbase-client-cell.jar"/>
  <path location="${modules.dir}/tools/sharedstate/build/
client/sharedstate-client.jar"/>
</pathconvert>

<!-- common jars -->
<pathconvert property="module-common.classpath">
  <path location="${module.libdir}/${module.common.jar}"/>
  <path location=
    "${appbase.dir}/build/client/appbase-client.jar"/>
  <path location=
    "${appbase.dir}/build/client/appbase-client-cell.jar"/>
</pathconvert>

<!-- Server side jars -->
<pathconvert property="module-server.classpath">
  <path location="${module.libdir}/${module.server.jar}"/>
  <path location=
    "${appbase.dir}/build/server/appbase-server.jar"/>
  <path location="${modules.dir}/tools/sharedstate/build/
server/sharedstate-server.jar"/>
</pathconvert>
```

Each, the client, the server and the common code are told about their dependency jar files. The first entry for each of them specifies the TealSim jar. The environment variables *module.libdir* is specified directly in the properties file for the module and points to the "lib" directory. The names of the jar files for server, client and common are also specified within the module properties file. In this case all classes of TealSim with the exception of the *teal.render.j3d* package are specified for all three jars. The other needed dependency files for the module are also stated here. Those are the appbase module later used to show swing components in Wonderland and the stared state component used to maintain a shared state between the clients. In order to add the TealSim jar file to the client and server jar the "lib" directory has to be added to the *client* and the *server* tag, respectively:

```
<client dir="${current.dir}/${module.libdir}">
  <!-- skipped code -->
</client>
<server dir="${current.dir}/${module.libdir}">
  <!-- skipped code -->
</server>
```

In order to react more quickly on changes in the TealSim code a new target *pack_tealsim* was introduced to the `build.xml` file. It packs all the needed `.class` files of TealSim into a jar file called `tealsim.jar` whenever there were some changes in the TealSim code. The directory of the `.class` files is

set in the module's properties file. The jar file is then copied into the "lib" directory. By adding the newly introduced *pack_tealsim* target as dependency to the *build* target it is assured that the new version of TealSim is used if any code was changed in TealSim.

3.3.2 The Artwork

The artwork will obviously be TealSim's 3D-Studio Max models used in the simulations. They are copied into the module's "art" directory. When the module is uploaded to a Wonderland server the contents of the art directory are stored in the Wonderland file system. In order to access that data Wonderland's resource locator must be used:

```
AssetUtil.getAssetURL("wla://module-name/path-to-element");
```

The *module−name* is to be replaced by *tealsim−module* in our case and the *path−to−element* by the needed element which in most cases is a `.3ds` file. Within TealSim a path to such a model is resolved with TealSim's own resource locator. This call has to be replaced with the call stated above. The resolver is only used by the scene factory when creating a *TNode3D* element. TealSim's factory for creating JMonkeyEngine specific nodes uses it's ***protected*** *getURL* method which takes a name of a 3D Model element and returns the right URL class using TealSim's resolver *URLGenerator*. In order to override this behavior and use Wonderlands resource locator code a class *ClientSceneFactory* is created and derived from TealSim's *SceneFactoryJME* class. The *getURL* method is the only one which needs to be overridden. When a simulation is loaded the *SceneFactory*'s *setFactory* method has to be used to set the right factory version, namely a *ClientSceneFactory* object.

3.3.3 Simulation Selection Functionality

Since one module should be capable of showing different simulations (but only one at time) in world a mechanism was chosen to select simulations. The functionality can be supported by the module properties. The user can simply right-click on the module's graphics and select the according properties window. With a module such a functionality can be added by implementing the *PropertiesFactorySPI* interface with a class. The interface declares the following methods:

```
public String getDisplayName();
public void setCellPropertiesEditor(CellPropertiesEditor editor);
public JPanel getPropertiesJPanel();
public void open();
public void close();
```

```
public void restore();
public void apply();
```

The *getDisplayName* method simply return the name of the property which can then be selected by the user in the properties menu. Wonderland provides a way to work with different languages. For that purpose a bundle file can be added to the module. This file consist of keys with a string value:

```
TealSim_Cell=TealSim Cell
TealSim_Component=TealSim Component
Save=Save
%some more assignments...
```

A value can be obtained with the following code:

```
ResourceBundle.getBundle("path-to-bundle-file").getString("key");
```

The path to the bundle file must be the full path consisting of the Java package and the bundle filename. Since the bundle is needed on the client side, the path to be stated with the *getBundle* method will in out case be `org/jdesktop/wonderland/modules/tealsim/client/resources/Bundle`. Within the filesystem the bundle file with the default language is named `Bundle.properties`. If other language files should be supported this name will differ. E.g. the name for the file with the german translations is `Bundle_de.properties`. Wonderland uses the client system's language per default. If there is no bundle file for that language the default bundle file will be used. For the *getDisplayName* the bundle key `TealSim_cell` was used, i.e. on english systems "TealSim Cell" and on german systems "TealSim Zelle" is returned. This can be adopted later by using a key dedicated to the property.

With the *setCellPropertiesEditor* method a properties editor is set to the properties factory class. Such an editor can be used to obtain the server cell state or the client cell object. Changes on the server state can be made as well. They will be applied whenever the user clicks on the "Apply" button. The editors *getServerState* method is used to obtain the name of the current simulation which can then be changed within the properties dialog. When the user changes the simulation to another one the editor's *addToUpdateList* is used to set the new server state which will be applied later on the server side. Another editor's method used is the *setPanelDirty* method which influences whether the buttons as "Apply", "Cancel" and so on are enabled or not. If the simulation in the properties panel is set to the one currently used, the panel's dirty value is set to ***false***.

Wonderland's properties dialog is a swing panel. This user interface can be fully costumized. In order to obtain the GUI the property factory's

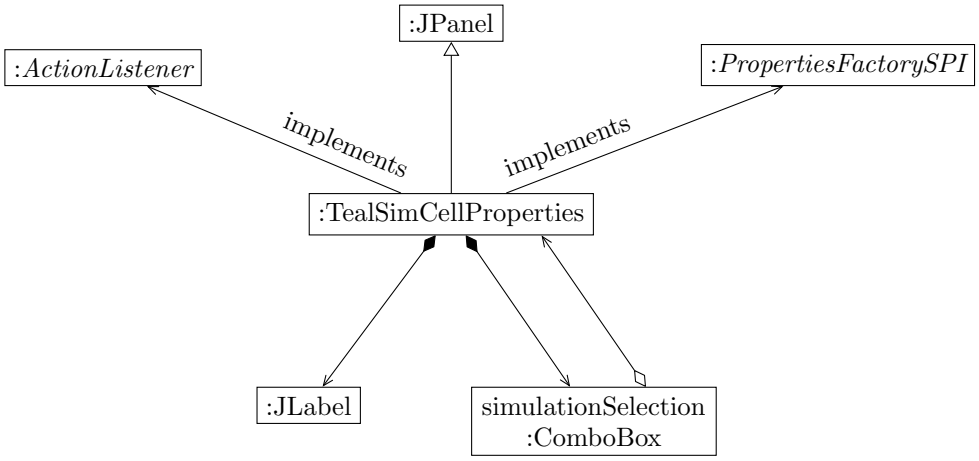

Figure 23: Object relations of the Properties Dialog

getPropertiesJPanel method is called. As shown in figure 23, the properties class itself represents the GUI as it is derived from the *JPanel* class. For that reason the *getPropertiesJPanel* method simply returns the properties class itself. The swing children of the panel are a label and a combo box. The label states the string "Choose Simulation" and next to it the combo box lists the implemented simulations. The *TealSimCellProperties* object is used as *ActionListener* for the combo box. Thus, it has to implement the *actionPerformed* method which the combo box object calls whenever the user chooses a simulation.

The additional methods within the *PropertiesFactorySPI* interface are called when the user opens or closes the dialog, or whenever the button according to the method is clicked. On *open* the default simulation shown in the combo box is set to the current one. On *apply* the simulation is set to the choosen one using the editors *addToUpdateList* method. The *close* and *restore* method are currently not in use. The properties dialog is very simple at the moment and does only fit the functional requirements. The possible simulations are stored within a *String []* member of the properties class. Whenever another simulation of TealSim is ready to run in Wonderland, it can simply be added to that array.

3.3.4 Creating a Simulation

As shown in figure 24, the process of loading the simulation is rather complicated. The according components of TealSim need to be informed about the other elements. Most of them hold references to each of the others. The *SimPlayerApp* class represents the whole user interface in the desktop

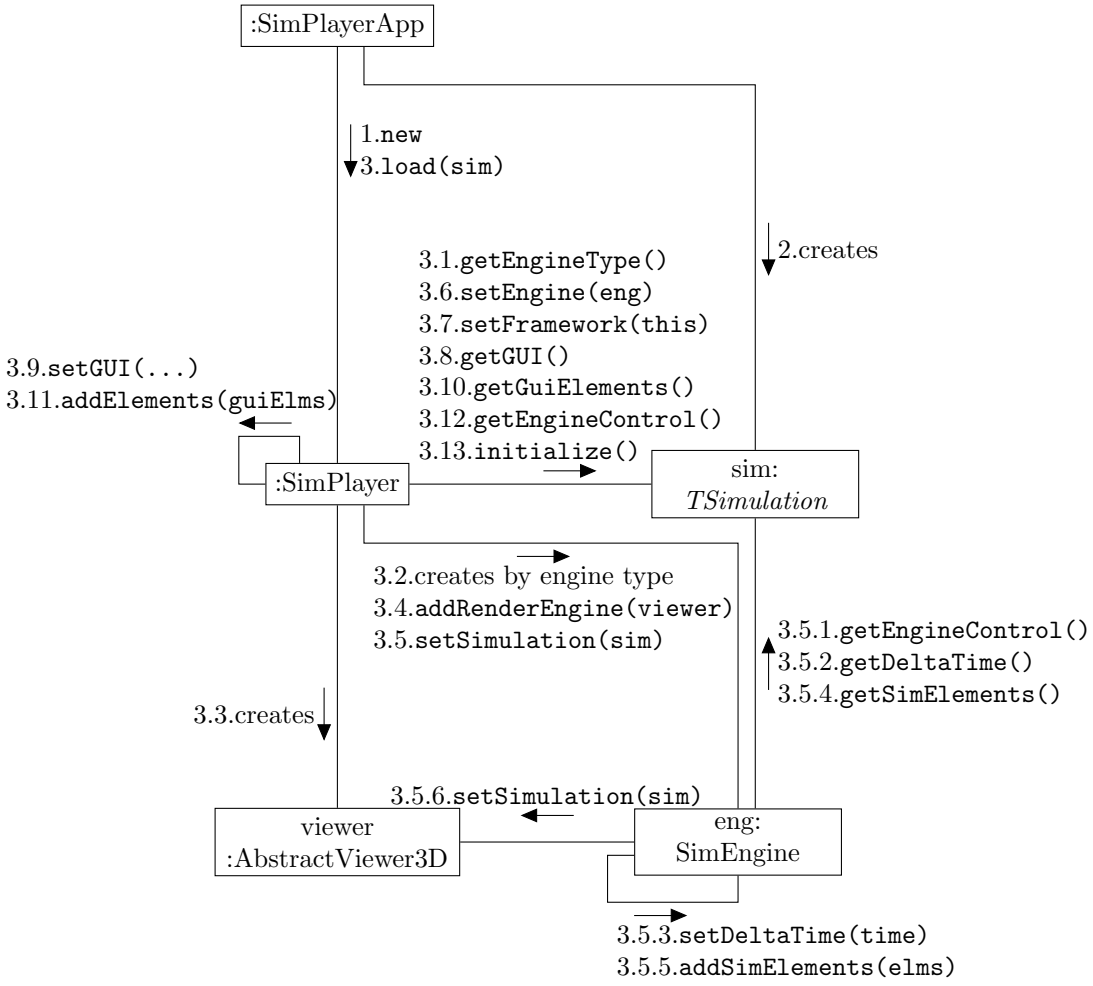

Figure 24: Communication Diagram of the creation of a Simulation with TealSim's Desktop version (slightly simplified)

version. First it creates the framework, in this case the *SimPlayer* class. Subsequently, the simulation is created. This happens whenever the user chooses a simulation. It is then given by a string which directly corresponds to the package and class name of the simulation. This way the simulation class can be loaded with Java's reflection functionality:

```
TSimulation sim =
        (TSimulation) Class.forName(nameString).newInstance();
```

The created simulation contains all the elements and parameters needed. It is passed to the player's *init* method which does the actual initialization. For the Wonderland version both, the client and the server side have their own

player. First the engine is created. For that purpose the necessary engine type is obtained from the simulation object. As stated in section 3.2.2, the engine is instantiated with the type as construction parameter. The last component needed is the viewer. With the desktop version, the proper scene factory class instantiates the viewer, on the client side a client specific viewer is created directly by the player. Since the server does not display anything all the steps including the viewer are skipped on Wonderland's server side. With the desktop version the viewer is added to the engine. With the Wonderland version this obviously happens only on the client side. In principle an engine supports more than one viewer, although this functionality is not needed at the moment.

The next step is to tell the engine about the simulation. The engine can obtain all the needed values for the simulation step from the simulation. The delta time for example is the time a step should require. Most importantly the engine obtains all the *TSimElement* object which are needed for the simulation. All this elements are added to the engine itself. After that step the engine sets the simulation on the viewer. Of course the server side engine for the Wonderland version skips this step since there is no server side viewer. At this step the viewer obtains all the information it needs from the simulation as well. This information is mainly the elements it needs to display. The fact that it is the engine which tells the viewer about the simulation unnecessarily makes the code more complicated. In the future the player should be responsible for that.

After the engine and the viewer are aware of the simulation this simulation is told about the engine and the player. This is necessary because some simulations are not only containers for all the elements, but do also contain functionality influencing the actual simualtion. On the desktop version the player then obtains the GUI object which specifies how the user interface looks like. With the desktop version of TealSim most simulations use the *SimGui* class. Figure 25 shows how the frames are arranged with this gui. To the left there is the view pane displaying the three dimensional output and the engine control buttons. On the right the control pane where the user influences parameters is displayed. This control pane is wrapped by a scroll pane in order to put a scroll bar to the left side of the pane if the frames are too high. Since the player is a *JPanel* it simply adds the GUI object as its own child. Because the 3D window on the client side is not part of the swing panel a *ClientGUI* class implementing the *TGUI* interface is defined within the client package of the Wonderland module. It works similar to the *SimGui* class in TealSim but instead of having a *viewPane* it has got an *ecPane* which holds the engine control. This panel is placed beneath the *controlPane*. With the next simulation initialization step the gui elements are

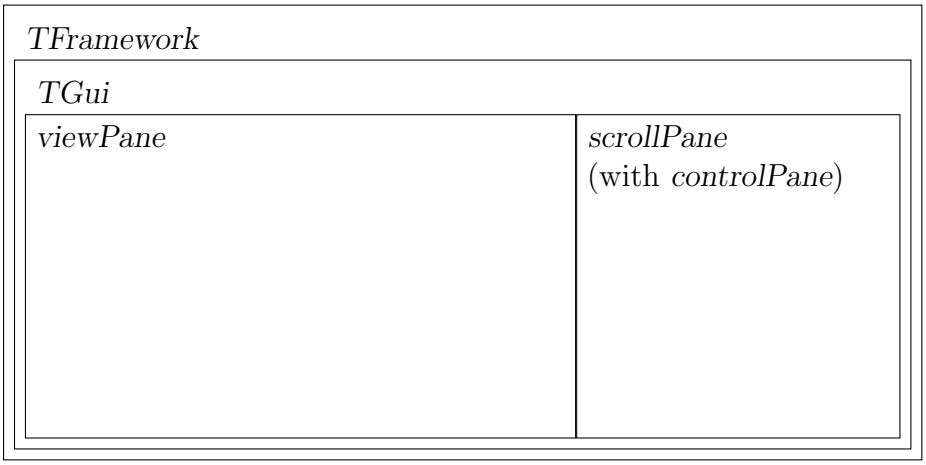

Figure 25: Frame parts of the Player (*TFramework*)

obtained from the simulation object by the player using the *getGuiElements* method. After that they are added to the GUI. The engine control is then set to the *init* state by obtaining it from the simulation and calling the *init* method on it. On the desktop version, this implicitly starts the engine thread and thus the main loop. However, there are no calculations made yet since the engine is in *init* state. The engine control basically represents the user interface with the "play", "resume", "pause" buttons and so on and also starts and stops the engine. On Wonderland's server side no engine control is needed since the engine's states are set mainly by the user interactions on the client side. Whenever an according message from the client arrives the engine's state is changed directly. On the client side an engine control is used similarly to the desktop version. However, it does not start the engine thread on initialization, but has additional responsibilities as sending informations about user clicks on its buttons to the server. The enabling and disabling of the buttons are also synchronized by the client's engine control (see section 3.3.6). Because the functionality is very similar but extended to the desktop versions engine control the *ClientEngineControl* is derived from TealSim's *EngineControl* class.

The last step of the simulation loading is to call the *initialize* method on the simulation. Some elements need the engine to be instantiated to be initialized properly. Previously the engine was created directly within the simulation's constructor. For flexibility reasons (this would have been a problem with different engines for the desktop version and the Wonderland version) the simulation does only contain information what type of engine (currently electromagnetic or other) is needed for the simulation and the

engine is created by the player. In order to keep most of the simulation code the parts which need the engine to be instantiated (e.g. flux field lines) are initialized by the newly introduced *initialize* method.

Instantiating a simulation can take a lot of time. Not only the main components have to be created but also all simulation objects and all the swing components. For the Wonderland module there are three possible ways to instantiate the simulation:

1. The simulation can be created on each client when it logs on. However, many simulations contain code where object are placed randomly into space. The clients must therefore synchronize the simulation objects after creation. Since there are many different simulations decisions would have to be made which simulation overrules the others. It seems to make more sense to create only one simulation object.

2. If the simulation is instantiated on the server the randomized parts of the simulation constructor do not matter any more. Once the simulation is created it can be distributed to the clients. Both, the server player and the client player can then do the initialization in their own way. The main problem with this approach is the long time it takes to create all the objects. Some tests made clear that for some simulations darkstar is not able to create the objects within the time a task can run. Increasing this time on the Wonderland server would break the requirement of being able to run the module on every Wonderland server without code changes.

3. To overcome the darkstar problem the simulation can be instantiated on a client and then be sent to the server. The server then distributes the simulation to all the clients. The server can not tell a single client to create a simulation because this single client possibly disconnects before sending the simulation to the server. For that reason the server tells all the clients to create the simulation as long as it does not receive a simulation from a client. The first simulation received is then used and sent to all clients.

In figure 26 the instantiation process of a simulation is shown. When the second client logs on the first one has not yet instantiated the simulation. For that reason the server tells the second client as well to create the simulation. This situation is similar to the one when the simulation is changed while more than one client is logged on. In that case the server will tell all the clients to create a simulation. When the server has received the simulation for the first time from a client it throws created simulations it receives afterwards

away, saves the simulation and distributes it to all clients. Whenever a new client logs on the current copy on the server is sent to it. After the client receives the simulation data its client player can initialize it as described previously. The described behavior requires some extra code and makes the module a bit more complicated. However, for previously mentioned reasons some simulations could not be created purely on the server. In future versions more of the creation code probably happens on the client (see also section 5.1). In this case the module is already prepared and there would be less changes to the code.

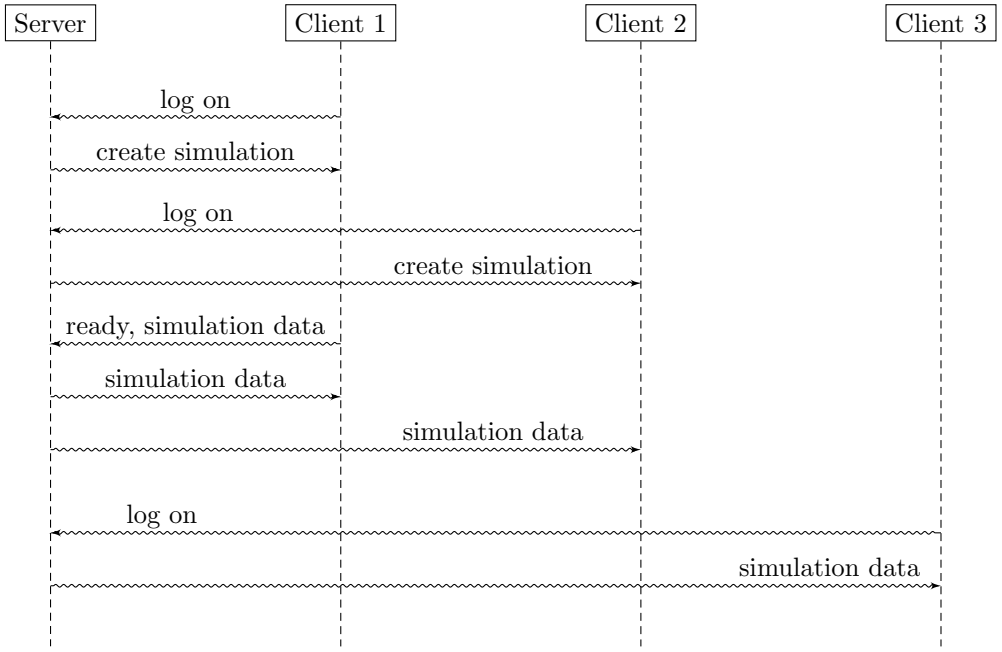

Figure 26: Sequence Diagram of Simulation instantiation and distribution

One issue with the client-server version is the creation of object ID's. Later such identification numbers will be used for synchronization. For that reason *all* the elements to be synchronized must have an unique identifier. It is assumed that only objects implementing the *HasID* interface will have to be synchronized since simulations should *never* use elements declared outside of TealSim. All the elements added to the simulation with the methods declared within the TealSim classes can simply be checked for an ID when they are added. With most swing elements in the simulations only the top container element in the swing hierarchy is added in such a way. The subcomponents implementing the *HasID* interface also have to be checked for an ID. For that reason all the elements derived from Java's *java.awt.Component* class

are asked for their children recursively using the *getComponents* method. If a *HasID* element has no id yet a random identifier is assigned:

```
((HasID) cmp).setID(UUID.randomUUID().toString());
```

Since the *HasID* interface works with Java strings the ID created with the *UUID* class has to be converted to such a string. As the awt components are parsed through all the single *HasID* elements are added to a list which is later used for synchronization.

After last step which has to occur in addition to the initialization as with the desktop version is to remove the references to all the server-specific elements. In the first place this is the simulation object which defines the whole simulation. This object implements the *ActionListener* interface. Some simulations use that behavior to add some execution code within the simulation. This code often has to be only executed once for the whole simulation, i.e. on the server side. For that reason the simulation is removed as action listener and the client side simulation is added instead to only transfer informations about the user interaction to the server side where the changes can be applied. Since the simulation's *actionPerformed* method is very heterogeneous among the different simulations the simulation as *ActionListener* is not supported yet. The simulation is only removed from all the elements it listens to. For that purpose the swing component tree has to be parsed again in order to remove the simulation as action listener.

3.3.5 The Control Panel

Most of the user interaction in TealSim happens with the swing user interface. Parameters within a simulation can be changed, the view can be changed (e.g. whether field lines show up or not) or the simulation can be started or stopped using this interface. Wonderland provides functionality as a module called *App Base* for that purpose. It can render light weight swing components onto a two-dimensional pane in world. User interactions are also possible quite as simply as with actual swing programming. However there is no synchronizing among clients implemented in the *App Base* module. This would break the modularity of Wonderland since synchronization should always be customized with knowledge of the actual application and therefore has to be implemented specifically.

Figure 27 shows the class hierarchy of a cell using the *App Base* module. Normally the implemented client cell class is directly inherited from Wonderland's *Cell* class, but when using the swing components it will be derived from the *App2DCell*. Similar things happen with the renderer and with the server side cell. The latter is usually derived from the *CellMO* class. This changes

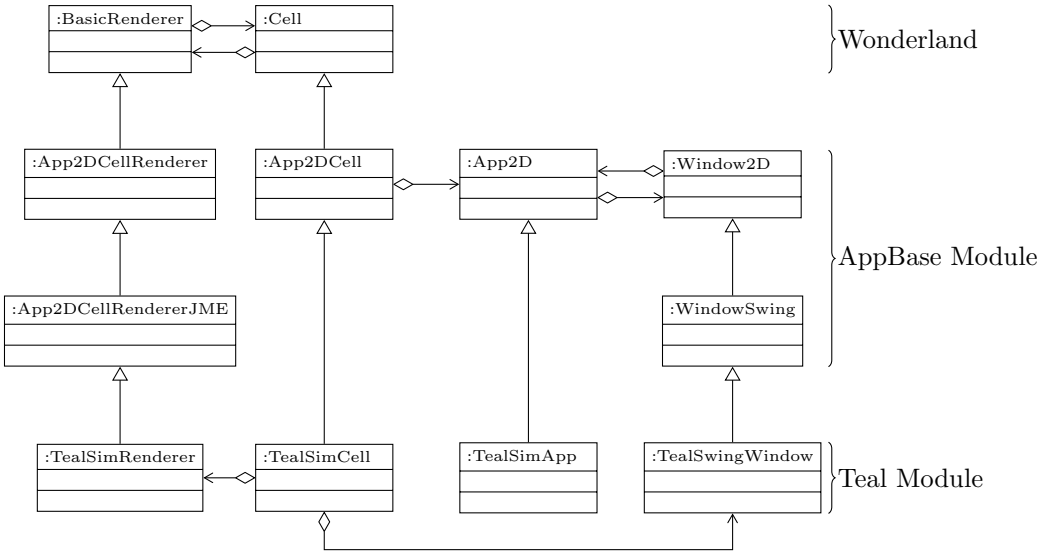

Figure 27: Simplified client side swing Components class hierarchy of the Module

to the *App2DCellMO* class (this is not shown in the class diagram). The 2D-App's renderer is usually not overwritten. But to extend the functionality of the 2D application to additionally render 3D graphics a *TealSimRenderer* derived from the normal JMonkeyEngine 2D application renderer is implemented. In order to instantiate this renderer the cell's *createCellRenderer* method which is called by the *Cell* base class itself is overwritten. It instantiates the TealSim version of a renderer. Within this renderer an additional branch of the scenegraph is added which holds the 3D output of TealSim. The viewer has to be given a reference in order to tell the renderer what elements to add to or remove from the scene graph.

A 2D application as well as the window holding the swing elements need to be created. This is done whenever the cell turns into the "active" state, meaning it is within a client's field of view. To react on such cell state changes the *Cell*'s *setStatus* method is overwritten and for the case of becoming "active" the following code is being added to the method:

```
// creating and setting up the 2D application
TealSimApp stApp = new TealSimApp("Test",this.getPixelScale());
setApp(stApp);
stApp.addDisplayer(this);

// creating the swing window
window = new TealSwingWindow(stApp, 500, 600, true,
                             this.getPixelScale());
```

```
// adding the client player as a panel
if(thePlayer != null) {
  window.addPanel(thePlayer);
  window.setVisibleApp(true);
}
```

When construction the application the name and the pixel scale has to be given to the constructor of the *TealSimApp* class. The pixel scale states the with of a single pixel in meters. Within the constructor of *TealSimApp* the superclass' constructor is called with an additional parameter: the control arbiter. Wonderland provides a class hierarchy of such arbiters with the base class *ControlArb*. The arbiter specifies the user interaction policy of the panel. In case of the *TealSimApp* the *ControlArbMulti* class is used as arbiter. This class specifies that many users can manipulate the swing window concurrently and that they need to "take control" first in order to do so. The arbiter instantiation and setup is done within the *TealSimApp* constructor. After the 2D application is created the cell has to be told about it by calling the *setApp* method located in the *App2DCell* class.

The next step is to create the swing window of a specified width and height. Both values are given in pixels. The pixel scale has again to be given. The fourth constructor parameter states whether the window is the main window which is ***true*** in our case. As with the desktop version's player, the client player is a swing panel as well. For that reason it can be added to the swing window using the implemented *addPanel* method. When the graphics of the module are not needed any more in Wonderland (e.g. if no client is logged in) the swing window is made invisible. As with the creation this can be done within the *setStatus* method of the cell when the "disk" status is reached. When the cell becomes "active" again the visibility only has to be set to ***true*** instead of again creating the 2D application and the swing window. In order to use the *App Base* module the according jar files of this module have to be added to the TealSim module's `build.xml` file (see section 3.3.1):

3.3.6 Starting the Simulation and synchronizing Engine States

With the previously explained functionality the cell is capable of showing up a simulation with the 3D objects and the swing elements on the clients. The next functionality to implement is to actually run a simulation. To achieve that a lot of synchronizing among the clients and the server is necessary. The user will start the simulation by clicking on the engine control's play button. After a simulation shows up some buttons are not enabled, i.e. they are not

clickable, because this would not make any sense. The pause button e.g. can not be pressen when the simulation is not started yet. However, as soon as the user clicks on "play" the pause button has to be enabled. In order to synchronize the button's ability to be clicked this information has to be sent to the server and from there to all the other clients whenever a click event occurs on the engine control's button list. The server does not need to know anything about which buttons are enabled and which are not. Some synchronization code would have to be added to the server side, though. Fortunately, a Wonderland module exists to maintain states among the clients without adding additional server side code, the *Shared State* module. It encapsulates the server side code. Whenever an additional module is used additional parameters have to be added to the TealSim module's `build.xml` file. This is shown in section 3.3.1. The *Shared State* module is implemented as a so-called "Component". From a logic point of view a Wonderland component is something to be attached to a cell. Usually components are added to cells using Wonderland's client user interface. In our case the *Shared State* component needs to be added to the cell permanently and as soon as it is needed. To achieve this behavior the following code lines are added to the server side cell class:

```
@SuppressWarnings("unused")
@UsesCellComponentMO(SharedStateComponentMO.class)
private ManagedReference<SharedStateComponentMO> sharedStateRef;
```

This is only a declaration telling Wonderland that a shared state component has to be attached to the cell. The member variable itself is never used within the class.

The *Shared State* consists mainly of a map *SharedMapCli* mapping Java strings to *SharedData* objects. This objects are the ones shared among all client. Currently there are three shared data classes implemented:

- *SharedBoolean*
- *SharedInteger*
- *SharedString*

For the enabled buttons in the TealSim module a *SharedInteger* is used to represent the buttons as bitmask. With a specific string key this value can be obtained from the map. Additionally a listener can be defined and added to the map. This listener is implemented as an inner class of the client engine control. It has to implement the *SharedMapListenerCli* interface and therefore have a callback method *propertyChanged*. This method is called whenever the map is changed on one of the clients. This way it can react

and change the enable mask of the buttons on the current client. The client cell has to tell the client engine control about the map immediately after it is created. Whenever a new client logs on and an existing simulation is already in world the enabled state of the engine control can be obtained from the shared map. The different keys of the map can be used for different values. Within the TealSim module the map is not only used for the engine control buttons, but also for another swing component explained in section 3.3.7.

Clicking a button at the client's engine control user interface must not only effect the button enabled mask, but obviously also the state of the engine. If the user clicks on the "step" button one simulation step should be executed. For that purpose the call is forwarded to the client engine which fires a property change event. The client cell which is capable of communicating with the server is a property change listener to the engine and sends the message about the pressed button to the server. For this purpose the *EngineMessage* class is implemented. It holds engine state changes as well as all the other communication regarding the engine. Since it is derived from Wonderland's *CellMessage* class every arbitrary objects can be placed inside the class and can be transferred by Wonderland. On the server side a message receiver to such message classes can be defined. For the engine messages the message receiver is a static inner class of the engine. This is helpful because it is within the same scope as the server side engine itself and can therefore access even the ***private*** members of the *ServerEngine* class. As the receiver class is derived from Wonderland's *AbstractComponentMessageReceiver* class it can be used with Wonderland's messaging mechanism. When the simulation is initialzed the engine's message receiver is instantiated and registered:

```
cell.getComponent(ChannelComponentMO.class)
        .addMessageReceiver(EngineMessage.class,
                new ServerEngine.EngineMessageReceiver(cell));
```

Registering the engine message receiver as a listener to the *EngineMessage* class causes a callback method (*messageReceived*) to be called whenever the according message is received.

The server side engine defines a method called *handleClientStateChange* responsible for handling state changes triggered by a user on the client side. This method can be called by the engine message receiver. The single parameter of the method is the new state which can be obtained from the engine message. The *handleClientStateChange* method does some pre-actions as stopping a running simulation if the simulation is in progress and forwards the state to the according methods like *start* for the "running" state. So far the engine states defined within the *TEngine* interface are used. The prob-

lem with that is that these states describe states and not state changes. E.g. there is no state saying that the "step" button was just pressed. Whenever this happens the pause state is transferred. For the server side it does not make a difference whether the "pause" button or the "step" button is pressed. In both cases a simulation already running is stopped and one additional step is calculated before the server side calculations stop. On the client side it is not possible to click the "pause" button twice in a row since the button will be disabled after the first call. With this approach the pause state have to be transferred multiple times if the user clicks on the according button. This behavior lacks intuitiveness since declaring "pause" several times in a row usually do not have any effect after the first call. Although not recognized by the user the additional step after the click on "pause" is also not intuitive. To solve that states saying "the user clicked on the pause button" and so on should be defined. With the "reset" button the same problem occurs. The *init* method is called on such a user interaction.

It is very important to not cache the engine within the simulation object. If it contains a reference to a *ServerEngine* class the object will be part of the managed object accessing the message receiver. Because of the serialization process in Darkstar a cached object would be re-serialized as an own object in its state previous to serialization. The managed object in the TealSim module holding the engine as well would implicitly have its own object and those two engine objects would not be synchronized.

The *ServerEngine* defines a *doStep* method performing a server-side engine step. After the calculation of the step the dependent values are sent to the client. Since the engine's *getDependentValues* method asks all the simulation elements for their values this call should not be called too often. Instead the server side engine caches the dependent values within a member variable which can be obtained in order to send it to the client. The value array is sent using the *EngineMessage* class. The clients receive this message and can then update their elements and do the client specific calculations as explained in section 3.2.2. Whenever the "start" button is clicked the *doStep* method has to be called repeatedly. Since the server-side engine implements Darkstar's *Task* interface, the engine calculation can be scheduled repeatedly. On scheduling such a task a *PeriodicTaskHandle* object is returned implementing a *cancel* callback method. The handle is stored as a member within the engine and is used to cancel the running task whenever necessary. The last functionality regarding the engine control buttons on the server is to reset the simulation. Every simulation knows what to do on reset and implements a *reset* method. Since the simulation is on the server it has to be called there. After calling the dependent values are sent to the client in the same way as after a simulation step.

3.3.7 Synchronizing the Swing User Interface

Similarly to the simulation elements some of the swing elements have to be synchronized among the clients as well. This is a rather complicated task since those elements are heterogeneous. Elements not influencing the server engine can be synchronized using the *Shared State* module. However, some of them are connected with TealSim *Route*'s to simulation elements. The server has to be told about everything influencing the simulation. Some of the swing elements do not have to be synchronized among clients at all. Those are e.g. check boxes indicating whether field lines are shown or not. Every client can select whether the field lines are shown or not on its own.

Visualization Control

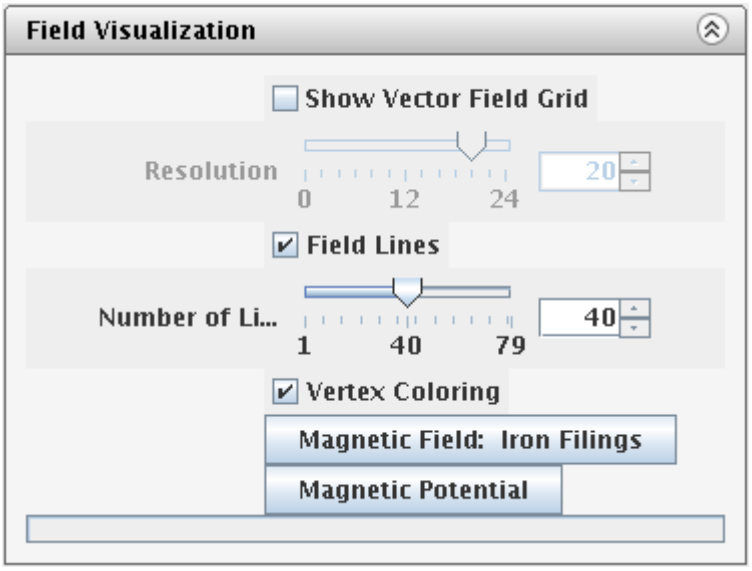

Figure 28: Visualization Control in GUI

TealSim's *VisualizationControl* class is responsible for any visualization of fields. It is a swing panel and added to the user interface in most of the electromagnetic simulations. In figure 28 the panel is shown. The simulation chooses which of the elements are needed within the simulation. Those not needed are set to be invisible. The first three entries concern the field direction grid and the field lines. The user can choose whether those are shown and how many of them are shown. For the field lines the vertex coloring can be enabled showing different colors for different strengths of the field along a field line. All those elements do not need to be synchronized since every user

chooses this part of the view independently. The mechanisms to influence the view (e.g. for hiding the field lines when the user disables the check box) on the client side work exactly the same way as with the desktop version of TealSim. Therefore they do not need to be adopted for the Wonderland version at all.

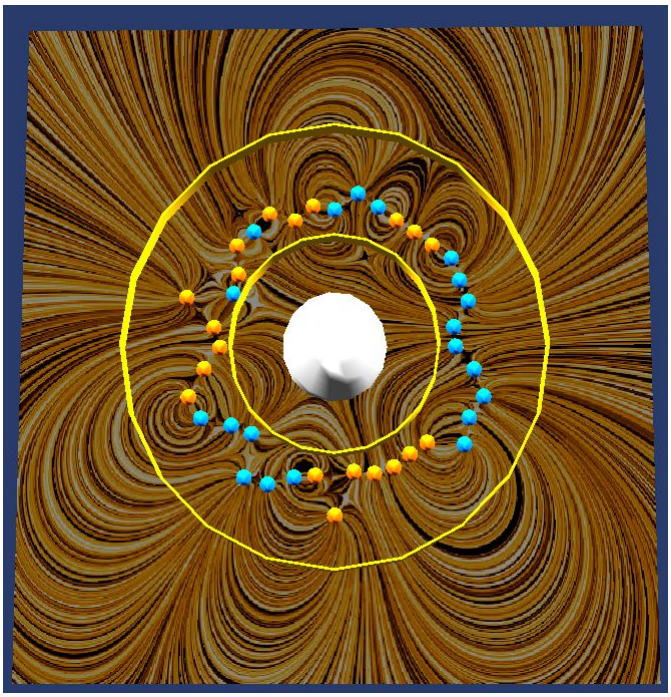

Figure 29: Display of electric potential in the "Charge by Induction" Simulation

If any of the buttons below are clicked the simulation is paused and the requested representation of the field is calculated. During the calculation the progress bar in the swing window indicates the current progress. When the calculation is completed a 2D panel as shown in figure 29 with the requested visual representation shows up. This process needs communication to the server because the engine needs to be stopped whenever a user clicks on such a button. The other clients need to know they should calculate and display the requested view. With the desktop version the engine's state is changed after a click on one of the buttons. In the Wonderland version the *ClientEngine* object replaces the engine in the desktop version. The client-side cell class *TealSimCell* is registered as a property change listener to the client-side engine listening for a change of the simulation state. With this mechanism the server is informed whenever the engine state should change.

This causes the server to stop the simulation when the user clicks on one of the buttons on the visualization control.

In order to tell the other clients that they should show up the panel with the field representation the *Shared State* component is used. For every *VisualizationControl* object an entry is put into the shared map. The value is an integer with the flag indicating what button was pressed. The integer keys for the buttons are already defined in TealSim's *DLIC* class. On every client a listener class called *VisualizationControlSynchronizer* is registered to the map entry. This class is also used for the rest of the functionality regarding button clicks on a visualization control. The user clicks are emulated on the other clients by calling the visualization controls *actionPerformed* method with the same parameter object as it would be called on the event of a user click. This parameter object is an *ActionEvent* object triggered by a *TealAction* with the value of the button mask as command.

The last issue with the synchronization of the visualization control is how to get the information within the module when the user has clicked a button. For that purpose the *VisualizationControl* is changed slightly. As stated above a *TealAction* is triggered on every user click. The *VisualizationControl* object itself is the listener to all those actions. Its code is adapted to store a list of all these actions where the visualization control is an action listener. A new method to add an external listener to all of those actions is introduced as well. This way the *VisualizationControlSynchronizer* in the Wonderland module is able to listen to all events the *VisualizationControl* is listening to. If the user clicks on the button the *VisualizationControlSynchronizer* can now trigger the event on all the other clients by changing the integer value within the shared map to the integer belonging to the clicked button.

Routes to simulation elements

Most of the other swing components influencing parameters of the simulation are connected to the simulation elements by routes. Most of TealSim's classes implement the *TElement* interface which declares some methods for establishing such routes:

```
public void addRoute(Route r);
public void addRoute(String attribute,
                     TElement listener, String targetName);
public void removeRoute(String attribute,
                     TElement listener, String targetName);
```

Whenever a route should be added to a swing component within the user interface the simulation calls the component's *addRoute* method. The *attribute* parameter is the name of the parameter to be addressed. With the *listener*

parameter the route target object is given. The *targetName* is the property the route effects. If e.g. a slider value is coupled with the charge value of a *PointCharge* object the target name will be "charge". This causes the *setCharge* method of the point charge to be called whenever the slider value is changed by the user. The call is performed by a *Route* object which contains the three parameters of the *addRoute* method. It is added to the element (in this case the slider) as a property change listener. A route can also be directly added to a listener using the appropriate *addRoute* method with the *Route* object as parameter. Routes can also be removed from an *TElement* object by calling the *removeRoute* method.

With the desktop version of TealSim the according target method of the simulation element can be called directly by the *Route* object. With the slider example this happens whenever the user moves the slider. The changed values of the charge are recognized by the engine immediately and taken into account while running an engine step. With the Wonderland version of TealSim the slider values will have to be transferred to the server because the charges effect the server side engine calculations. The sliders also need to be synchronized among the clients. This is done by sending the values back to the clients. The server also has to store the new values in order to be able to send them to newly logged on clients. The slider values of the simulation stored on the server side must be kept up to date. After creation of the simulation the server only keeps a copy of the simulation in order to distribute it to new clients.

In order to detect user interactions on swing components a property change listener called *GuiPropertyChangeListener* is introduced to the client side of the Wonderland module. It is an inner class of the client side player. On initialization of the simulation it is instantiated and registered to all needed swing components. If a processed swing component has subcomponents they are also parsed recursively. Since all swing components are derived from the *java.awt.Component* class the *GuiPropertyChangeListener* can call the *addPropertyChangeListener* method on the components with ***this*** as parameter.

Whenever a user clicks on a GUI component or changes any value the *GuiPropertyChangeListener*'s *propertyChange* method will be called with the fired event. In this case an *EventContainer* is constructed and packed into an *EventMessage* object which will later be sent to the server. The event container contains the property change event and the ID of the source object which will be a simulation element. Within the *PropertyChangeEvent* class the source object is ***transient*** and will therefore not be transferred to the server. The ID is used instead and the server will resolve this identifier to the corresponding object on the server side. This is done by the server player's

handleClientPCE method which is called by a message receiver. With a *propertyChange* call on the server side swing object the server is updated. After that the message previously received by the server is sent to all the clients. They process the message in a simlar way to the server. The identifier will be resolved to an object and the property change event will be applied to this object.

A problem with this approach is that the network between client and server is not synchronized. If serveral property changes happen with the same object within a short time they can become mixed up. This is very likely to happen with the sliders. When a user drags on a slider the value changes very frequently. What can happen is that one value is transmitted to the server and then sent back to the clients. In the mean time the client value has changed because the user drags the slider further. While the new value is about to reach the server the old one comes back to the client (where the slider already shows the new value). The old value is applied again which causes the slider to jump back to the old value. Every value change triggers a new sending event. This is usually stopped by the server since it does neither process nor forward a message to the client if the property value has not changed. However, this mechanism does not work when the values change quickly since the property value will alternate between two or more values. This causes a slider to continously jump between different values. The effect is even worse if two different clients change the same GUI element. To reduce the effect the number how often one and the same value is sent to the server is limited. The less frequently a client is allowed to send such data to the server the more unlikely are the synchronization problems to occur. However, if the network speed to a user is lower and therefore the latency is higher the problem occurs again. The transmission frequency of the synchronization messages can not be made arbitrarily long since the user interaction would be delayed too long. Additional synchronization mechanisms are still to be implemented.

3.4 Implementation of a Multi-player Simulation

In order to make use of the virtual world a new simulation was implemented. Games are an interesting way to learn physics. In TealSim several games are already implemented for that reason. One of those was extended to be played by three clients at the same time. The players of the game have to communicate with each other during the game with respect to the physics.

Figure 30 shows the implemented game. There are three charges. Their charge values can be changed using the three sliders in the swing window. According to the charge values the three charges attract or repulse each

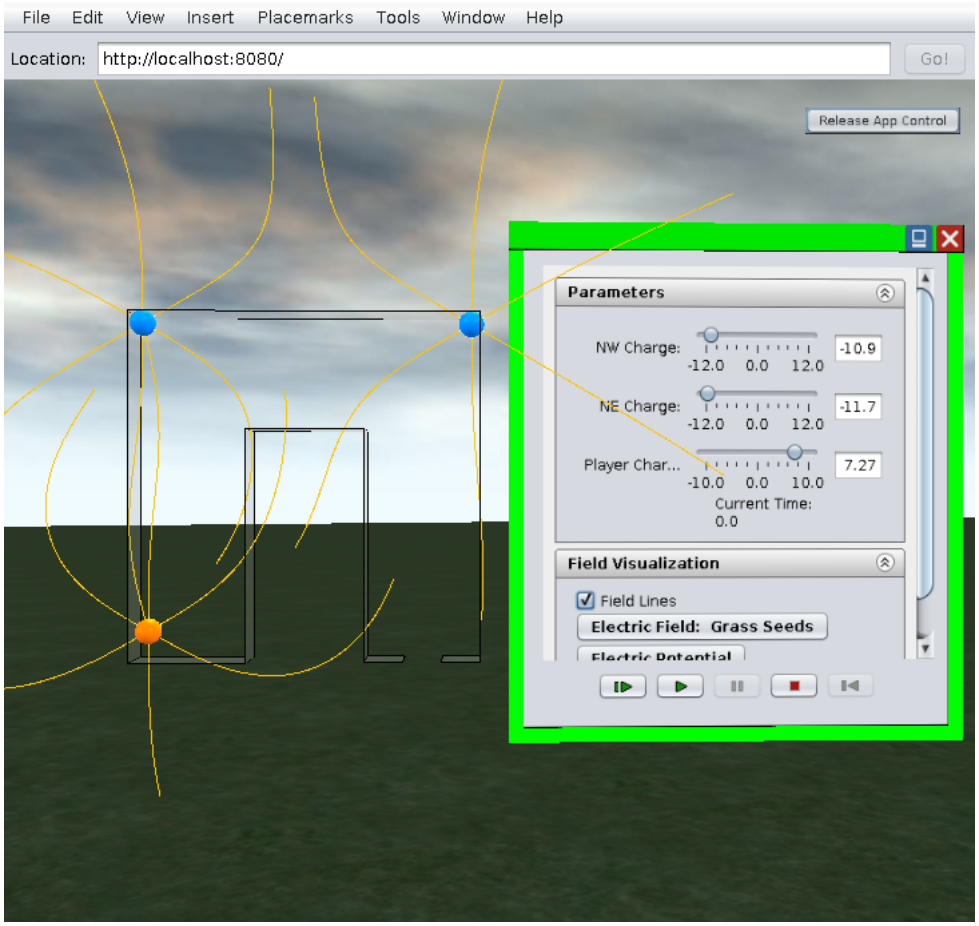

Figure 30: 3-Player Video Game in Wonderland

others. The two blue charges above can not change their position but the third charge can. The goal of the game is to navigate the movable charge through the horseshoe shaped body until it reaches the exit on the bottom right hand side. This can only be done by changing the charge values properly while the charge is moving. The colors of the charges change with their charge value. Every user can switch on or switch of the field lines.

The simulation was added to TealSim and can therefore also be used with the desktop version. In that case it is not easy to play though since one user will have to change the values of the three sliders during the game. The implementation of the team game was rather simple since an other game used and adopted for three players. In the original game there was only one slider which was used to change the charge value of the moving charge. The values of the other two charges remained the same. To adopt the simulation

to a three player game two sliders were added. Their values were coupled with the charge values of the other charges using routes.

4 Installation and Usage of the Module

This section explains how to use the implemented module together with a wonderland server. For better understanding several screen shots are added. All the explained details require a running Wonderland server with access to the server administration. The module will be compiled using the *ant* command. The resulting `jar` file represents the module and has to be uploaded to the wonderland server. Wonderland comes with a Glassfish application server and a corresponding web interface. This interface can be started with a web browser. On the main page a user can click on the "Server Admin" button in order to administrate the server. Figure 31 shows how to upload a module. On the top image the server administration page as it appears first is shown. To the right the different servers running are shown. They can also be started or stopped here. On the left hand side of the page the page menu is shown. In order to upload the TealSim module the "Manage Modules" section has to be selected (marked orange in the image). The "Manage Modules" page is shown at the bottom of figure 31. All the installed modules are shown here and can be removed as well. On top of the page the module `jar` file can be uploaded. After the upload the Darkstar has to be restarted by clicking the "restart" link in the "Manage Modules" page shown at the top of image 31.

Now the module can be loaded in world. The wonderland client is started by a click on the "Launch" button at the home page of the web interface. The login window appears where a username can be chosen. A TealSim cell can be loaded by selecting "TealSim Cell" at the dialog shown after clicking the "Insert" and "Object..." item at the task bar. Figure 32 shows the cell after it is loaded. The "Capacitor" simulation is loaded by default. In Wonderland TealSim looks very much like it's desktop version. On the right hand side a control panel is shown to the user and the simulation can be started or stopped. The main visual difference is the 3D part of the simulation which is constructed in world and not within a window next to the control panel.

To prevent from unintended clicks the user has to take control of the panel before accessing the panel. This is done by right-clicking on the panel. The menu shown at the right bottom of figure 32 having a "Take Control" item appears. This menu can also be used to open the properties of the cell. As shown in figure 33 some "Capabilities" of the selected cell can be changed within the object editor showing up after clicking on "Properties...". The frame on the right hand side changes according to the selected capability. In order to switch to another simulation the "TealSim Cell" capability has to be selected. The "Properties" frame now shows a drop-down menu where the desired simulation can be selected. After selection the "Apply" button

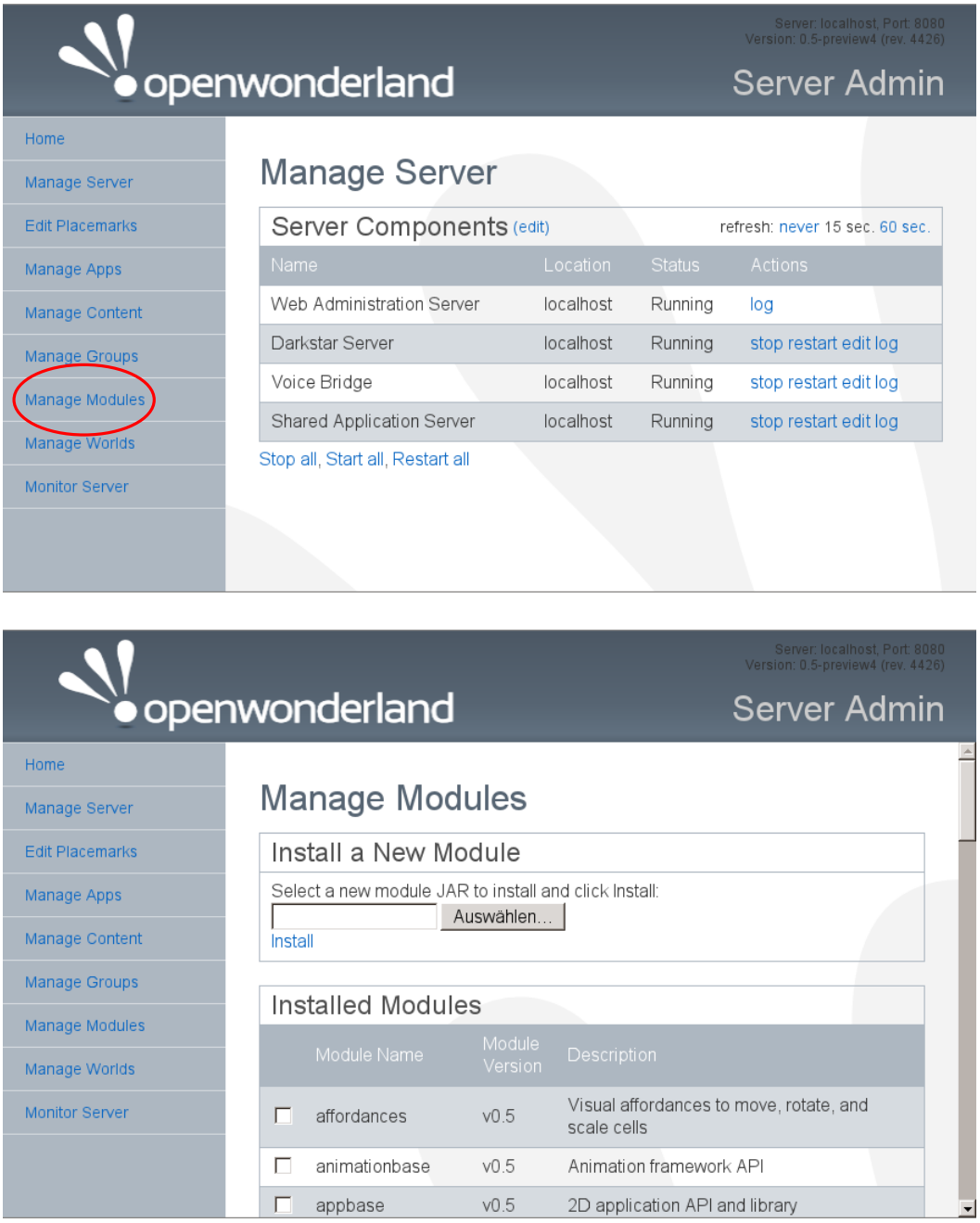

Figure 31: Uploading a Module

can be pressed and the selected simulation will show up in world.

Of course more than one simulation can be placed in-world at once. The user can add an arbitrary number of TealSim cells and change every instance

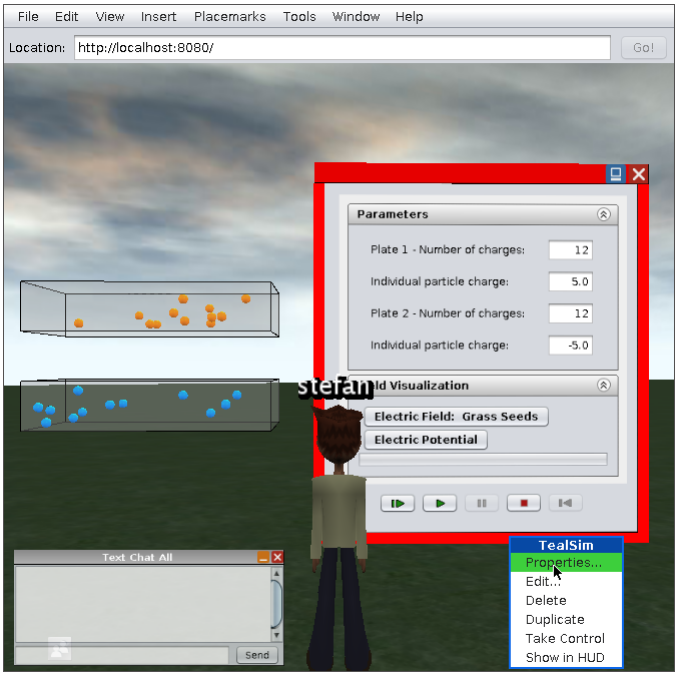

Figure 32: Cell after it is loaded

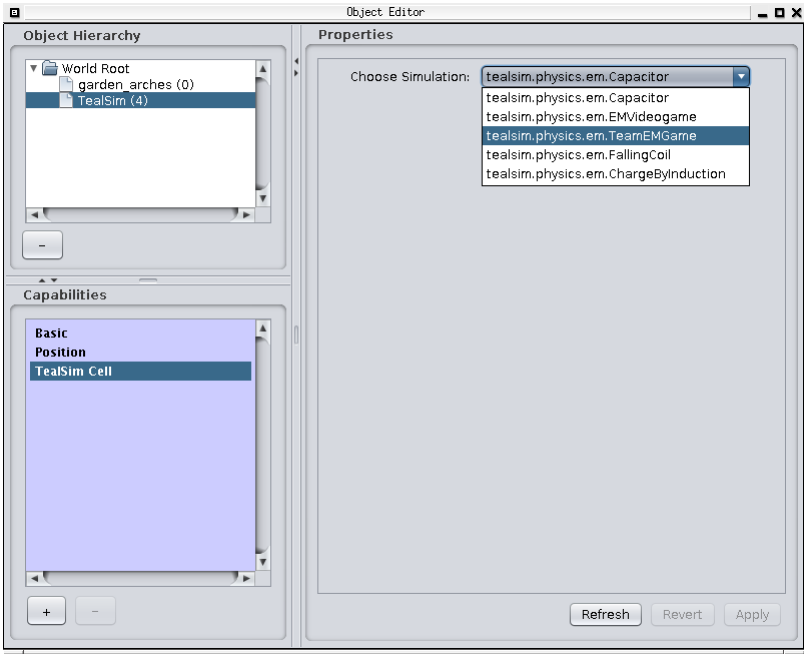

Figure 33: Properties Window of the TealSim Cell

to a different simulation. The quickes way to do so is to right-click on the GUI and select the “Duplicate” item. This way the simulation object is duplicated and can be placed somewhere else within the virtual world.

5 Conclusion and Outlook

This work aimed at providing several physics simulations within Open Wonderland. In order to enable as many different simulations as possible an existing simulation software was adopted to run in the virtual 3D world. With the implemented functionality many users can meet in-world and study the physics behind the experiments supported by the simulation software. Built-in features of the virtual world enable communication methods as stereo sound or blackboards to increase the learning effect.

In order to point out the advantage of the simulation software within a 3D virtual world compared to the desktop version a multi-player e-learning game was implemented. Tests were pointing out the software's capability to manage several avatars playing at the same time. The work showed that the two environments (TealSim and Open Wonderland) can be coupled in order to enable the benefits of both environments. Since some stableness issues with both environments were detected a lot of effort was taken to overcome such shortcomings. The fact that both used environments are open source software decreased this effort. Although there are documentation shortcomings with the two environments a powerful learning environment was created.

5.1 Future Work suggestions

Since the focus lied on providing the functionality little attention was payed to the world around the users. Open Wonderland provides many features to build a world around the simulations that supports learning. There is also the positive social effect when groups of people meet [3] which can be increased by a virtual world closer to real world. There is also a potential in enhancing the usability. In figure 34 a simulation where the two dimensional swing windows are replaced by three dimensional knobs is shown. Compared to the current implementation with the swing windows this approach seems to be closer to real world.

Another issue occuring because of the focus on the functionality is the lack of usability in some cases. The following minor changes regarding that topic are recommended:

- The distance between the 2D swing window and the three dimensional simulation can not be changed by the user at the moment. Especially when the simulation should be placed within small 3D-spaces as buildings this functionality is essential. For better usability the mechanism to change the distance should be similar to the position change mechanisms built in in Open Wonderland.

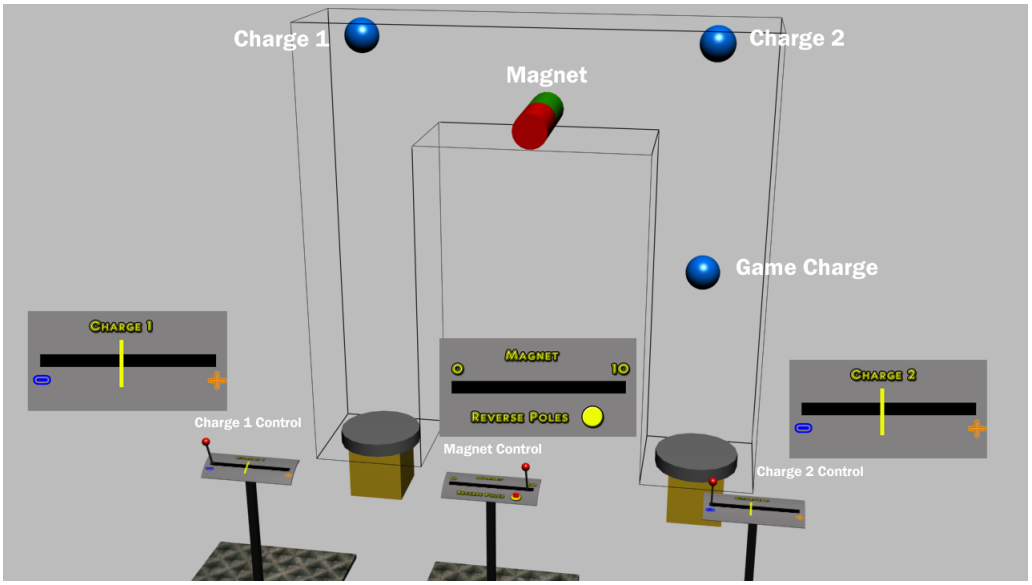

Figure 34: Simulation with 3D User Interface

- At the moment the 3D elements of the simulations are scaled to a diagonal width of three meters. This size can not be changed without editing the Java source code. An additional dialog for changing the size of the 3D and the 2D parts seperately is needed. Again, the interface for that should be similar to the ones already used in Open Wonderland.

- The simulation change dialog is just a simple drop-down menu at the moment. Since well-designed user interfaces increase the acceptance and satisfaction of the users [11] this user interface should be refactored.

In order to achieve the functionality for the first two points the cell could be split into two cells, one for the 3D components and one for the swing components. This way Open Wonderland's mechanism to resize, move and scale cells could be applied to both parts separately.

One major issue with the implementation is performance. The implemented module scales up well for more than a dozen users depending on the simulation. However, some speed improvements can still be made by pulling elements out of the server and by optimizing the server to client network traffic.

Although many different simulations are supported not all of them work out of the box within the Open Wonderland module. A lot of the simulations are not implemented fulfilling the specifications. In order to use them with the module they have to be refactored. So far only electromagnetic simula-

tions were tested. With the other simulations in TealSim a broad spectrum of different physics simulations could be used within Open Wonderland.

References

[1] John W. Belcher, *Studio Physics at MIT*, MIT Physics Annual 2001, `http://web.mit.edu/physics/news/physicsatmit/physicsatmit_01_teal.pdf`

[2] Jonathan Bishop, *Enhancing the understanding of genres of web-based communities: the role of the ecological cognition framework*, Int. J. Web Based Communities, Vol. 5, No. 1, Pages 4-17.

[3] Callaghan, M.J.; McCusker, K.; Losada, J.L.; Harkin, J.G.; Wilson, S.; , *Teaching Engineering Education Using Virtual Worlds and Virtual Learning Environments*, Advances in Computing, Control, & Telecommunication Technologies, 2009. ACT '09. International Conference on, Pages 295-299, 28-29 Dec. 2009 DOI: 10.1109/ACT.2009.80

[4] Yehudit Judy Dori, John Belcher, *Effect of Visualizations and Active Learning on Students Understanding of Electromagnetism Concepts*, Proceedings of the Annual Meeting of the National Association for Research in Science Teaching (NARST 2003), `http://icampus.mit.edu/projects/Publications/TEAL/EffectOfVisualizations.pdf`

[5] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional; 1 edition, November 10, 1994

[6] Todd Greanier, *Discover the secrets of the Java Serialization API*, Java tutorial, `http://java.sun.com/developer/technicalArticles/Programming/serialization/`

[7] Bernard Horan, Michael Gardner, and John Scott. *MiRTLE: A Mixed Reality Teaching & Learning Environment. Technical Report.* Sun Microsystems, Inc., Mountain View, CA, USA

[8] Masound, F.A.; Halabi, D.H.; *ASP.NET and JSP Frameworks in Model View Controller Implementation*, Information and Communication Technologies, 2006. ICTTA '06. 2nd , vol.2, Pages 3593-3598, DOI: 10.1109/ICTTA.2006.1684998

[9] Paul R. Messinger, Eleni Stroulia, Kelly Lyons, Michael Bone, Run H. Niu, Kristen Smirnov, Stephen Perelgut, *Virtual worlds – past, present, and future: New directions in social computing*, Decision Support Systems, Volume 47, Issue 3, Online Communities and Social Network, June 2009, Pages 204-228, ISSN 0167-9236, DOI: 10.1016/j.dss.2009.02.014.

[10] Norhayati Abd. Mukti, Dayana Razali, Mohd. Fadzil Ramli, Halimah Badioze Zaman, Azlina Ahmad, *Hybrid Learning and Online Collaborative Enhance Students Performance*, Advanced Learning Technologies, IEEE International Conference on, Pages 481-483, Fifth IEEE International Conference on Advanced Learning Technologies (ICALT'05), 2005

[11] Rodriguez, N.J.; Borges, J.A.; Murillo, V.; Ortiz, J.; Sands, D.Z.; , *A study of physicians' interaction with text-based and graphical-based electronic patient record systems*, Computer-Based Medical Systems, 2002. (CBMS 2002). Proceedings of the 15th IEEE Symposium on, Pages 357-360, 2002 DOI: 10.1109/CBMS.2002.1011405

[12] Scheucher, B., Bailey, P., Gütl, C., Harward, V. *Collaborative virtual 3d environment for internet-accessible physics experiments* Pages 65-71, International Association of Online Engineering Vol5, 2009

[13] Jordan Slott, *Project Wonderland (v0.5): Importing 3D Models*, Java Tutorial, 2010 `http://wiki.java.net/bin/view/Javadesktop/ProjectWonderlandArtImport05`

[14] Jordan Slott, *Project Wonderland v0.5: Working with Modules*, Java Tutorial, 2010 `http://wiki.java.net/bin/view/Javadesktop/ProjectWonderlandWorkingWithModules05`

[15] V. Vani and S. Mohan. *Interactive 3D class room: a framework for Web3D using J3D and JMF*, In Proceedings of the 1st Amrita ACM-W Celebration on Women in Computing in India (A2CWiC '10). ACM, New York, NY, USA, , Article 24 , 7 pages. DOI:10.1145/1858378.1858402

[16] Zhenlong Li and Xiaoming Zhao. *The Design of Web-Based Personal Collaborative Learning System (WBPCLS) for Computer Science Courses*, In Proceedings of the 7th international conference on Advances in Web Based Learning (ICWL '08), Springer-Verlag, Berlin, Heidelberg, Pages 434-445. DOI: 10.1007/978-3-540-85033-5_43