JMonkey Engine User's Guide

Compiled by Tomás Lázaro

July 4, 2008

Contents

Introduction v

I jME Wiki 1

1 Getting Started 2
1.1 Prologue . . . 2
1.1.1 Installation Guide . . . 2
1.1.2 Steps and Components . . . 3
1.1.3 Finally . . . 11
1.2 Introduction . . . 11
1.3 The Main Game Loop . . . 12
1.4 Application: AbstractGame and subclasses . . . 13
1.4.1 PropertiesDialog . . . 14
1.4.2 PropertiesIO . . . 16
1.4.3 Game Loop . . . 16
1.4.4 Game Types . . . 17
1.5 DisplaySystem . . . 23
1.5.1 Example 1 - Creating a DisplaySystem . . . 25
1.5.2 Example 2 - Getting Screen Coordinates . . . 25
1.5.3 LWJGLDisplaySystem . . . 26
1.6 Logging System . . . 26
1.6.1 Example 1 - Using the Logger class . . . 26
1.7 Renderer . . . 26
1.7.1 Drawing . . . 27
1.7.2 Example 1 - Creating and Obtaining the Renderer . . . 27
1.7.3 Example 2 - Clear, Draw and Display . . . 28
1.7.4 Example 3 - Obtain the Rendering Statistics . . . 28
1.7.5 LWJGLRenderer . . . 29
1.8 Texture Renderer . . . 29
1.8.1 Creating . . . 29
1.8.2 Example 1 - Create a TextureRenderer, Set the Camera and Get the Texture . . . 29
1.8.3 Applying . . . 30

1.8.4 Example 2 - Render to the Texture and Display it on a Quad 30
1.8.5 Rendering Other Buffers 31
1.8.6 Example 3 - Rendering the Depth Buffer to a Texture . . 31
1.9 RenderQueue 31
1.10 Multiple Rendering Passes 32
1.10.1 Usage 33
1.10.2 RenderPass 33
1.10.3 Simple Shadows Example 33
1.11 Writing a simple jME enabled Java Applet 35
1.11.1 Before you begin 35
1.11.2 Step One 36
1.11.3 Step Two 36
1.11.4 Signing 39
1.11.5 Issues and Gotchas 39
1.11.6 Cleaning up on exit 39
1.11.7 Threads and OpenGL 40
1.11.8 Memory Usage 40
1.11.9 Next Steps 40
1.12 Fundamental Shapes 40
1.12.1 Arrow 41
1.12.2 Axis Rods 42
1.12.3 Box 43
1.12.4 Rounded Box 44
1.12.5 Capsule 45
1.12.6 Cone 46
1.12.7 Cylinder 47
1.12.8 Disk 48
1.12.9 Dodecahedron 49
1.12.10 Dome 49
1.12.11 Hexagon 50
1.12.12 Icosahedron 51
1.12.13 Octahedron 51
1.12.14 Pyramid 52
1.12.15 Quad 53
1.12.16 Sphere 54
1.12.17 GeoSphere 54
1.12.18 Teapot 58
1.12.19 Torus 58
1.13 Summary 59

Glossary **62**

Preface

This is an attempt to produce printable documentation to reach more and more people. I'm using LaTeX so it can be easly modified to suit everyone's need. I'm actually learning how to use it and I chose this as an interesting project to learn it as well jME. I'm not responsible for any information here, I'm just a copy—paste—format grunt. Feedback, typos alerts and suggestions are welcome: tlazaro18@gmail.com.

Introduction

This guide is designed to aid the developer in writing their applications using the jME API. The primary application for the jME API is gaming software and graphically intensive applications. This API is designed to allow for the most efficient graphics system and gaming systems.

jME was very much influenced by the work of David H. Eberly and his book 3D Game Engine Design. While not completely taken from this source and evolved well beyond it, much of jME's functionality is derived from his teachings. I highly recommend you pick up a copy of this wonderful resource.

The primary design paradigm used by jME is the concept of the Scene Graph. The scene graph provides a hierarchically grouped tree where each Node is ordered based on Spatial location. Every node of the tree can have one parent and multiple children. This allows for efficient scene rendering as well as an easy method for the user to build their game's scene.

This guide will cover all concepts, classes and usage defined in the jME API. First, the basic system window management concepts will be covered. This will explain how a DisplaySystem is created and how to use the Renderer as well as what Renderer contains and allows. Next, the geometrical concepts of jME will be covered. This will give an understanding of the mathematical concepts including Matrix, Vector3f, Vector2f, Quaternion and others. Then, the Camera Model will be explained, as well as how Frustum Culling is determined. Next, the basic concepts of the scene graph will be covered, explaining the usage of all scene graph classes including Spatial, Node and Geometry. This will lead into the discussion of RenderState and how they can be used to modify the look of a node. Next, picking and collision detection will be explained. Which will lead into jME's usage of curves and surfaces. Once surfaces are defined, Model Types can be explained as well as how these can be used for Animation. Chapter 10 - Level of detail will be explained and then Chapter 11 - Terrain. Next, spatial sorting will be explained including Quad-Trees and Oct-Trees, Portal Systems and Binary Space Partitions. Lastly, different special effects will be explained.

Please, provide any feedback to the API and this document. This is a team effort and open in every way.

Regards, Mark Powell jMonkey Engine

Part I

jME Wiki

Chapter 1

Getting Started

The beginning is the most important part of the work. *Plato*

1.1 Prologue

Before diving in, you might want to see the following links to understand what jME is and what it can do:

- Demos
- Screen Shots

Note that jME is a Java Game Engine, not a Game Creator. If you are new to programming, or don't know what Java (as opposed to Javascript) is, then this is not the ideal place to find out. I would suggest you follow some basic Java tutorials and come back when you feel confident.

1.1.1 Installation Guide

Getting jME up and running in your development environment is relatively straight forward. However, there are several components you need to have in order to really get going. This guide will outline all the necessary steps from beginning to end. **Important: There is a much simpler guide to using jME if you are using an IDE such as eclipse or netbeans. They are located here: Eclipse Netbeans**

Note: This is geared toward a Windows XP installation. Other versions of Windows will have a similar procedure. Linux and other OS' will be similar as well, but the specifics are not covered here.

1.1.2 Steps and Components

You need several things in order to develop with jME:

- Java Compiler and Runtime Environment
- CVS Source Code Control Utility
- Ant Build System
- LWJGL OpenGL/OpenAL Game Library
- jME Source Code

Note: When extracting files from archives, make sure you do not simply drag the files out of the archive, use the actual decompression function. In some programs, the former method will not create subdirectories which will cause errors.

Don't panic if this is sounding too complicated! It's easy. Here are the basic steps we're going to follow:

- Download and Install Java SDK
- Download and Install CVS Tool
- Download and Install Ant
- Download the jME Source Code, Plus LWJGL
- Compile jME
- Verify jME is Working

Ready to get jME up and running? Let's go!

Step 1: Java Compiler and Runtime Environment

You need to have the Java Compiler and Java Runtime Environment installed in order to use jME.

If you already have the Java SDK, you can just skip this step. If you don't, you can download the Java SDK from Sun from java.sun.com.

There are usually links to download the SDK's on the right hand side of the main page. It's a large download (around 50Mb), so it may take a few minutes, even on a high speed connection. It's probably easiest to just go ahead and download it to your desktop. You can get rid of the installation file later.

Once the SDK has been downloaded, double-click on the icon to begin the installation. Follow the steps and complete the installation.

After the installation has completed, it would be a good idea to verify that your installation was successful. Open a command prompt and type:

```
java -version
```

You should see something like the following:

```
java version "1.6.0_04"
Java(TM) SE Runtime Environment (build 1.6.0_04-b12)
Java HotSpot(TM) Client VM (build 10.0-b19, mixed mode, sharing)
```

If you get an error, java may not be set up correctly. After you have the Java SDK installed, you can go on to the next step.

Step 2: CVS Source Code Control Utility

To obtain the source code for jME, you need to get access to the source code repository on a remote server. To do this, you use CVS.

Note: Getting and compiling the source code for jME is what this guide is all about. If you just want the pre-compiled jar files of the latest release, you can simply download them from the website. Note that you will also need to download and install LWJGL manually!

Download WinCVS from: `www.wincvs.org` Note that they have a windows client as well as one for the Mac and GTK. Again, this guide covers the Windows version. The others are probably very similar, though.

The download should be a zip file containing a 'setup.exe' executable file. After downloading this file (again, your desktop would be a good choice), use WinZip (or whatever unzip tool you prefer) to extract the the setup.exe file to your desktop.

Double click on the 'setup' file to begin the installation of WinCVS, and follow through until it's finished.

That's it for installing WinCVS. You can set that aside for the moment, though. Before you can really do anything with the source, you'll need a way to compile it. That's done using a tool named 'Ant'.

Step 3: Ant Build System

Ant is a powerful, free, open source, build system written in Java. (It is in essence a replacement for 'make', if that helps clarify anything for you.) In order to use it, you need to download and install it. You can get ant from ant.apache.org. Simply go to the site and follow the instructions to download the zip file. (Again, your desktop is an easy choice.) Now that it's downloaded, you need to get it installed so you can use it. Ant does not have a nice installer like other programs, but it's not too difficult to set up.

First thing: Select a location for the installation. Following the Windows paradigm: C:Program Files/Apache would be a good choice. You can pick whatever you want, though (some people just use: C:/ant for example.)

Next, extract ant from the zip file you downloaded into the directory you choose.

Now that you have ant unzipped, you need to be able to access it from the command prompt, which means altering your Windows setup to include it in your path. For Windows XP (or Windows 2000, and possibly NT), to alter your environment, do the following:

- Right click on 'My Computer', go to 'Properties'
- Click the 'Advanced' tab
- Find the 'System Variables' section at the bottom.

Now that you found where you need to be, you need to add and change a few things:

First, add these variables:

```
ANT_HOME - C:\Program Files\Apache\ant (or wherever you installed it)
JAVA_HOME - C:\j2sdk1.4.2_05 (or wherever your SDK is)
```

Then, update/edit the 'Path' variable to look like:

```
;%JAVA_HOME%\bin;%ANT_HOME%\bin
```

Make sure you don't remove the variables that are already there. Now, just click 'Ok', and your system should be updated.

Older Versions of Windows You may need to update your config.sys file instead. This can be found in: C:\config.sys

To do this, just add the following lines to it (Note that you should replace the directories with the ones that you actually installed Ant and Java in):

```
set ANT_HOME=c:\Program Files\Apacheant
set JAVA_HOME=c:\j2sdk1.4.2_05
set PATH=%PATH%;%ANT_HOME%bin
```

After updating config.sys, just go ahead and re-boot. (You won't need to do that for WinXP/2000)

Finally Now, Ant should be installed and ready to go! To test it, open a new command prompt window. (Note that you can't use one that was up before you altered your environment, though! Just open a new one.) At the prompt, in any directory, type:

```
ant -version
```

It should print out a message that is clearly from ant, and is the version you just downloaded and installed. If you get an error message, check your environment to make sure you specified everything correctly.

That's it for installing Ant. You're now finally ready to download jME!

Step 4: jME Source Code

Summary of steps:

- Setup WinCVS for connecting
- Create a destination directory
- Connect to CVS server
- Download jME code with LWJGL
- Disconnect from CVS server

The first step is to download the jME source. In order to do this, you will use your CVS client. Basically, the command-line CVS commands you need to execute are as follows: (make sure to replace [java.net username] with your java.net username)

```
cvs -d :pserver:[java.net username]@cvs.dev.java.net:/cvs login
cvs -d :pserver:[java.net username]@cvs.dev.java.net:/cvs checkout -P jme
```

If you have a command-line CVS client, you probably already know what you're doing, and you just needed the commands. If you're new to this, or are using WinCVS, read on.

WinCVS First, if you haven't done it already, start WinCVS. A 'Preferences' dialog may appear, which you will need to add some things to. If it doesn't appear, go to: Admin → Preferences

In the 'General' dialog that appears, enter the following:

```
Authentication: pserver
Path: /cvs
Host Address: cvs.dev.java.net
Username: [java.net username]
```

Using Windows Explorer (or a command prompt), create a place to put the jME code you're about to download. Something like: C:/projects might be a reasonable choice. Don't worry that it's not jME specific. That will be handled for you later.

Next, you need to login to the remote CVS server that stores all the code. To do this, go to: Admin →Login.

Your previously entered data should be there under the 'General' tab. Also, you should probably check the box at the bottom of the 'Local Settings' tab that says 'Force Using the CVSROOT (-d)' so you can get the '-d' option. Click 'Ok' to start the login. WinCVS should connect to the host, then prompt you for a password. Enter your java.net password here.

You should now be logged into the CVS server, so you're ready to get the source code! This is called performing a 'Checkout' of the code. In order to do this, go to: Remote → Checkout module

That should bring up a dialog asking for a module and a destination directory. There are many different projects on on this CVS server. A 'module' specifies which project you're interested in. In this case, you want the source for jME.

Enter 'jme' (without the quotes) for the module name. This specifies that you want the code for the jME project.

Next, either navigate to or specify your newly created folder as your destination directory (ie, C:projects). Again, don't worry that it's not 'jME' specific.

After specifying this, click 'Ok'. Now, the jME source code should begin downloading to your local machine. It will end up under a 'jme' directory in your destination directory. Ie, in this case: 'C:\projects\jme'

After it finishes downloading, it'd probably be nice of you to go ahead and log out of the CVS server. Use Admin → Logout to disconnect.

That's all there is to getting the source code! Easy, right?

There's an added benefit of downloading the source code instead of the jar files. You get LWJGL with it! LWJGL is a lower-level API that communicates to your video card via OpenGL, and is needed for jME to function. LWJGL will end up being under (in our example): c:\projects\jme\libs

Now that you have the source code, it needs to be compiled. You will use the ant installation that you did eariler.

Step 5: Compiling jME

Compiling jME is easy after you've done all the setup.

First, open a command prompt, then change to the directory where you downloaded ant to. In the example here, this can be done with:

cd c:\projects\jme

Note: There may be a small issue with your 'jme' directory: There might be 2 of them, one inside the other! The jme directory you're interested in is the one that has the 'build.xml' file in it. If you have this issue, you should be able to just move the directories around to eliminate the duplication. This appears to be a CVS organization issue.

Now, you're ready to compile. It only takes a couple commands to do that, neither of which will take very long. The first command compiles the main jME source code:

```
ant dist-all
```

You should now see ant going through the various build targets and performing the compilation. There should be no 'javac' error messages. The second command is to compile all the testing code and demos:

```
ant dist-test
```

Ant should now quickly compile the test files. Again, there should be no 'javac' error messages.

That's it. It's done. Now the only thing left to do is test your installation.

Step 6: Testing jME

To test your jME installation, you can just run some of the demo programs you compiled in the previous step.

First, open a command prompt (or use one you already have open). Then, navigate to your main 'jme' directory. In our examples, this would be: C:\projects\jme (Or, from the note in the previous section: C:\projects\jme\jme You're after the one with the 'lib', and 'target' directories in it.)

The next real step is to run a demo. However, before you do that, it would be beneficial for you to actually understand how things work.

Here's the command you ultimately want to execute:

```
java -Djava.library.path=./lib -cp
./lib/lwjgl.jar;./lib/jogg-0.0.5.jar;./lib/jorbis-0.0.12.jar;
./target/jme.jar;./target/jme-effects.jar;./target/jme-model.jar;
./target/jme-sound.jar;./target/jme-terrain.jar;./target/jmetest.jar;
./target/jmetest-data.jar;./target/jme-awt.jar
```

Important Be aware that the sample command-line here generally will not work as-is, because it contains versioned jar files, and these versions are updated regularly. You must look in the lib and target subdirectories to see the current versions. You probably just want to update those version numbers in the CLASSPATH, not pollute your CLASSPATH by including all of those jars.

UNIX-specific Most of the CLASSPATH setup samples on this page won't work for UNIX. However, UNIX/Linux/MacOS/Cygwin users can get a head start with this shell script. Just download it to your JME root directory, make it executable, and run with no args to see syntax usage. It's much easier to accommodate library changes this way. FIXME: Seems that this Wiki does not accommodate attachment uploads other than images. If you know how to do it, please upload the script to the Wiki for me and change the link to an Internal link.

If you understand everything above, you're good to go. If you don't, a little explanation is in order.

Java Command Line

```
java
```

This is what runs all java applications. It launches the Java 'virtual machine' and executes a compiled java program inside of it. You don't really need to know how that all works, but it would probably be helpful to you in the long run to learn more about it. But, this isn't a java tutorial, so we'll just leave it at that.

```
-Djava.library.path=./lib
```

This portion of the command defines where the java VM can find external libraries. In this case, it tells it where to find the .dll files for LWJGL (they're in the lib directory go have a look if you want). This needs to be there because LWJGL uses those libraries to talk to the graphics card in your computer, and jME needs to use LWJGL.

```
-cp ./lib/lwjgl.jar;./lib/jogg-0.0.5.jar;./lib/jorbis-0.0.12.jar;
./target/jme.jar;./target/jmetest.jar;./target/jmetest-data.jar
```

This portion tells the java VM what the 'classpath' is. The classpath of an application defines where needed compiled files can be found. Without the classpath being set, the java VM wouldn't know to look in the jar files for the various classes, nor would it know where the compiled demos or jME source code is. (jar files are collections of files, usually compiled classes)

- lwjgl.jar is the jar file for LWJGL. It contains all the files it needs to do what it does. Since jME uses LWJGL extensively, nothing will work without this.
- jogg-0.0.5.jar and jorbis-0.0.12.jar are needed for sound support, specifically ogg and vorbis. If you don't use any sounds, you shouldn't need these files.
- jme.jar is the compiled jME source code, packaged up in a jar file. Naturally, jME won't work without its own code.

The 'jmetest.jar' and 'jmetest-data.jar' files contain all of the code for the test applications (demos), as well as the extra data needed to run them (images, models, sounds, etc..). You can't run the tests without the applications themselves!

Note: One final thing to mention about the classpath: '-classpath' could be used instead of '-cp'. They mean the exact same thing. '-cp' is just a very commonly used shorthand.

Note: One more final thing to mention about the classpath :-). If you're only working with JME in your command shell, it is usually much more convenient to set your CLASSPATH variable to the exact same value as as for the -cp/-classpath param. Your java command lines become much shorter this way because you don't need the classpath stuff any more. Samples for Windows and UNIX Bourne-compatible (incl. Linux and Cygwin):

```
set CLASSPATH=lib/lwjgl.jar;lib/jogg-0.0.7.jar;lib/jorbis-0.0.15.jar;
target/jme.jar;target/jme-effects.jar;target/jme-model.jar;
target/jme-audio.jar;target/jme-terrain.jar;target/jmetest.jar;
target/jmetest-data-model.jar;target/jmetest-data-cursor.jar;
target/jmetest-data-sound.jar;target/jmetest-data-skybox1.jar;
target/jmetest-data-texture.jar;target/jmetest-data-images.jar;
target/jme-awt.jar
```

```
CLASSPATH=lib/lwjgl.jar:lib/jogg-0.0.7.jar:lib/jorbis-0.0.15.jar:
target/jme.jar:target/jme-effects.jar:target/jme-model.jar:
target/jme-audio.jar:target/jme-terrain.jar:target/jmetest.jar:
target/jmetest-data-model.jar:target/jmetest-data-cursor.jar:
target/jmetest-data-sound.jar:target/jmetest-data-skybox1.jar:
target/jmetest-data-texture.jar:target/jmetest-data-images.jar:
target/jme-awt.jar
export CLASSPATH
```

Running a Test Now you need to tell java what application to run. You can't just give it the name of the application file, though. You have to enter the full package name of it. By convention, the package name of a file corresponds to what directory it's in, hence calling it a path. Again, this isn't a java tutorial. If you don't fully understand packages in Java, it would probably be in your benefit to learn how they work.

Now that we've covered the command, what are the demos that you can run? Well, there are lots of them. You can go look through the testbuildjmetest directories, looking for class files that begin with the word 'Test'. The full package name will name will be the directory names separated by a '.' (See the examples below).

There are lots of demo programs. The ones listed below are some of the more interesting ones. One final note before we get to them, though. The demos will all start with the jME splash screen. To run the actual demo, pick your options and click 'Ok'.

Now, for a partial list of demos. Again, you can find more in the testublidjmetest directories. These are listed in alphabetical order. Run any or all of them in any order you want:

- jmetest.effects.TestDynamicSmoker
- jmetest.effects.TestLensFlare
- jmetest.effects.TestParticleSystem
- jmetest.intersection.TestCollision
- jmetest.renderer.TestBoxColor
- jmetest.renderer.TestDisk
- jmetest.renderer.TestEnvMap
- jmetest.renderer.TestScenegraph
- jmetest.renderer.TestFireMilk
- jmetest.sound.TestSoundGraph FIXME I believe this class is obsolete. Perhaps replaced by jmetest.audio.TestJmexAudio
- jmetest.terrain.TestTerrainLighting

Just in case you're still having trouble understanding what to do, here is the exact command to run the first demo on the list, the 'TestDynamicSmoker' demo. Again, this assumes you're in the base 'jme' directory (the one where you ran ant from):

```
java -Djava.library.path=./lib -cp./lib/lwjgl.jar;
./lib/jogg-0.0.5.jar;./lib/jorbis-0.0.12.jar;
./target/jme.jar;./target/jmetest.jar;
./target/jmetest-data.jar jmetest.effects.TestDynamicSmoker
```

If you have set your CLASSPATH variable as explained previously, your java command becomes

```
java -Djava.library.path=./lib  jmetest.effects.TestDynamicSmoker
```

That's it! Try out the demos. They give you a nice overview of some of the really great features jME has. They should help you see why jME is an excellent choice for developing your future game.

1.1.3 Finally

Still having trouble? Visit the jME Forums. There are lots of helpful people there, ready to answer your questions.

1.2 Introduction

jMonkey Engine (jME) was designed to be a high-speed real-time graphics engine. With the advances made with computer graphics hardware as well as that of the Java programming language, the need for a Java based game library became apparent. jME fills many of those needs by providing a high performance scene graph based rendering system. jME leverages the latest in OpenGL capabilities to push the capacity of the most modern graphics cards while providing ease of use for the game designer. jME was designed with ease of use in mind, while maintaining the power of modern graphics programming. The modular design insures that as computer graphics hardware improves and OpenGL improves to accommodate, jME will be able to quickly make use of the newest features. This same modular design allows the programmer to make use of as little or as much of the jME system as they deem fit.

At the core of the jME system is the scene graph. The scene graph is the data structure that maintains data of the world. The relationships between game data (geometric, sound, physical, etc) are maintained in a tree structure, with leaf nodes representing the core game elements. These core elements typically are the ones rendered to the scene, or played through the sound card. Organization of the scene graph is very important and typically dependent on the application. However, typically, the scene graph is representing a large amount

of data that has been broken up into smaller easily maintainable pieces. Typically, these pieces are grouped in some sort of relationship, very commonly by spatial locality. This grouping allows large sections of the game data to be removed from processing if they are not needed to display the current scene. By quickly determining that a section of the world is not needed to be processed, less CPU and GPU time is spent dealing with the data and the game's speed is therefore improved.

While the scene graph is the core of the graphics elements of jME, the application tools provides the means to quickly create the graphics context and start a main game loop. Creation of the window, input system, camera system and game system are no more than one or two method calls each. This allows the programmer to stop wasting time dealing with the system and more time working on their own application. The display system and renderer provide abstractions to communicate to the graphics card. This is done to allow for a multitude of rendering system that are transparent to the end application. The main game loops are provided to allow for easy extension of the looping class and easy application development.

Actual rendering of the scene graph is abstracted away from the user. This has the benefit of allowing the swapping in and out of different rendering systems without breaking a single line of application code. For instance, if you run using LWJGL from Puppy Games, you could as easily switch to JOGL from Sun. There should be very little noticeable difference on the end system and no rebuilding of your project. However, to create a better API, the focus so far has been spent on adding features rather than maintaining more then one rendering system. Therefore, LWJGL is currently the only supported renderer. (A JOGL renderer has been created but not maintained, contact a developer if you are interested in this).

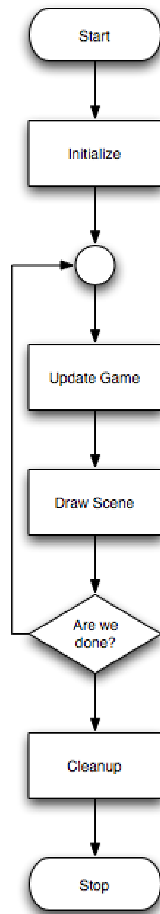
Each tool of the jME system was built using the basic building blocks. So all graphical elements will originate from the spatial class, all inputs and associated actions will come from the InputAction class, and so on. This will insure that consistency is maintained, and once the initial learning curve is overcome, using more advanced features will be substantially easier.

jME provides you, the programmer, with the tools you need to build your application quickly and efficiently. It will give you the building blocks you need to achieve your goals without getting in your way.

1.3 The Main Game Loop

At the core of every game is a loop - a portion of code that gets executed time and again throughout the life of the game. This loop, is responsible for coordinating a basic Update/Draw cycle.

All AbstractGame subclasses are responsible for setting up this loop. While each subclass varies this loop slightly they all follow the same basic style:



With the advent of multi-core machines, this game loop may vary dramatically. All discussion of game loop mechanics assumes a single threaded loop. That is, the initialization, update and render phase of the game occurs in the same thread. Further discussion can be found in Chapter 14 - Multiple Threads.

1.4 Application: AbstractGame and subclasses

When creating your graphics application, you will need a main class thats contains your main() method. jME provides the AbstractGame class and its subclasses. The AbstractGame classes provide the main game loop. This game loop is the backbone of the application and drives the entirety of the application. The game loop is called repeatedly until the application ends, where one iteration through the loop is called per frame. How the game loop is processed is dependent on which subclass of AbstractGame you choose. AbstractGame and all its subclasses are abstract classes and therefore cannot be directly in-

stantiated. The intent is that the user's application will extend one of the provided game types implementing the abstract methods. It is also perfectly accepted and expected that there will be times when the provided subclasses do not meet your game's needs and you must build your own subclass of AbstractGame. There is nothing stopping you from this; while the provided game types cover most application types, it does not cover everything.

NOTE: First, the getVersion method will provide you with a string detailing the version number of the jME library. The version string will be in the form:

```
jME version 1.0.
```

This string is a useful tool for system information.

1.4.1 PropertiesDialog

Overview

PropertiesDialog provides a Swing GUI for user's to select properties about the running application. This includes setting the screen resolution, bit depth, monitor frequency, fullscreen flag and select the rendering API. This is a useful tool for quickly managing your application and testing various settings, but it is recommended that you create your own in game system for resolution control.

PropertiesDialog reads and writes to a properties file using PropertiesIO. This properties file saves the settings between runs.

Setting

You may also set the behavior and appearance of the PropertiesDialog using the setDialogBehaviour methods. The PropertiesDialog is a dialog that appears (if so set) upon application start up to request information on the display settings. The dialog provides the means to select the resolution, color depth, full screen mode and refresh rate of the display. It also provides a means to select the valid renderers. These settings are then saved in a properties configuration file. Depending on the constant passed into the setDialogBehaviour method, the dialog will display always, will display if there is not properties configuration file set, or will never display.

Constants

NEVER_SHOW_PROPS_DIALOG Will never display the options screen. This must be used with care, for if there is not properties file set, the application will

likely fail on systems that are not able to display a default 640x480x60Hzx16bits fullscreen. However, if you can guarantee that the properties file will be present, you will not bother the user of the application during start up.

FIRSTRUN_OR_NOCONFIGFILE_SHOW_PROPS_DIALOG Will display the options screen if no configuration file is present. By default this setting is used. This will allow the user to set his display once and never be bothered again. However, if the user wants to change his settings later, he'll have to delete his properties file (or alter it manually), or you as the application developer will have to provide an in-game way of selecting new settings.

ALWAYS_SHOW_PROPS_DIALOG Will always display the options screen. This is what is used on all of jME's test applications. While very robust in letting you quickly and easily change your display settings, it may become obtrusive to the user if he must see this dialog each run.

You may also display a different image in the options screen using the setDialogBehaviour method. By default the jME logo is displayed, but if your application has it's own logo, using this method you can change the options screen to display it.

AbstractGame will create the dialog for you, and you typically access it through this class.

1.4.2 PropertiesIO

PropertiesIO maintains a properties file for saving the display settings for a jME application. Typically, you do not need to directly interact with this class, as AbstractGame defines a PropertiesIO class. AbstractGame saves a properties file as properties.cfg in the running directory.

PropertiesDialog also makes use of PropertiesIO to manipulate the saved settings.

Gotcha

- Do not worry about seeing: WARNING: Could not load properties. Creating a new one.. This will happen during the first run (it doesn't exist yet).

1.4.3 Game Loop

The game loop begins when the start method is called. The start method contains the main game loop and is implemented by the individual subclasses of AbstractGame. While the implementation may vary slightly across different game types, the basic game loop consists of the following stages:

1. Initialization of the System.

2. Initialization of the Game.
3. Begin Loop.
4. Update Game Data.
5. Render Game Data.
6. If time to exit continue otherwise go to 4.
7. Clean up.

Depending on which game type you are using, you will have to implement the various stages of the loop. We will delve into each game type individually, to see the variation and learn how you would properly implement each game stage, but first, there are some AbstractGame methods that provide some utility.

1.4.4 Game Types

There are five game types provided by the jME library.

BaseGame

BaseGame provides the most basic of implementations of the AbstractGame class. It defines the main game loop and nothing more. The loop is a pure high speed loop, each iteration is processing as fast as the CPU/GPU can handle it. The user is required to fill out each loop method:

The initSystem method provides the location where the developer is required to set up the display system. These are the non-game elements, such as the window, camera, input system, etc. Basically, all game systems should be initialized here. We will go into more detail later when we discuss the various system classes.

The initGame method is where all game or scene data is set up. This is purely application dependent and different for most every game. The basic elements usually consist of setting up the scene graph and loading game data.

The update method provides the location to update any game data. This may be the game timer, input methods, animations, collision checks, etc. Anything that changes over a period of time should be processed here. What occurs here is purely application dependent, however, there are some common processes that occur here and we will discuss them as they become relevant.

The render method handles displaying of the game scene to the window. While this too is application dependent, it typically involves two method calls: clearing the display and rendering the scene graph. We will go into this in more detail as we discuss the Renderer.

The reinit method is what is called when the system requires rebuilding. Such times as when the display settings change (resolution change for instance), will require a call to the reinit method to rebuild the display settings.

The cleanup method handles closing any resources that you might have open to allow the system to recover properly.

SimpleGame

The next game type and the one designed for quick usage and easy prototyping is SimpleGame. SimpleGame is a subclass of BaseSimpleGame and implements all the abstract methods of BaseSimpleGame. Instead, the user only must implement simpleInitGame. In the simpleInitGame, the user must only worry about his own game data. There is even a default root scene node (rootNode) that all game data can be attached to. This root node is automatically rendered. Everything else is initialized for the user, including the input system (FirstPersonHandler), the timer is set up, as well as render state for: lighting, zbuffer, and WireFrameState (switched off by default). This provides a basic platform for the user to experiment with different jME features without having to worry about building a full featured application.

Optionally, you can override simpleUpdate to add additional update requirements. This will only add to SimpleGame's updating capabilities, not override any. You can also override simpleRender to add rendering needs. This also will not override existing rendering functionality of SimpleGame.

Most jME demos use the SimpleGame type.

Examples in this guide will typically use the SimpleGame class, to conserve screen space. However, some examples may require a more in depth look and a different example may be used.

SimpleGame extra

SimpleGame provides an implementation of BaseGame. Its intention is to provide a framework to put together quick examples without having to worry about any of system initialization. SimpleGame provides a single abstract method, simpleInitGame. This method is for the user to create the game data to display. Any created Spatial objects should be attached to the rootNode that SimpleGame provides.

SimpleGame creates an input system that detects the following keys:

Access to the InputHandler can be obtained with the input variable that SimpleGame provides.

SimpleGame creates a standard Camera whose position is (0,0,25), left Vector is (-1,0,0), up (0,1,0) and direction is (0,0,-1). This Camera, referenced in SimpleGame as cam, is controlled by input.

SimpleGame provides a single Light. This is a white PointLight at position (100,100,100). It is attached to a LightState named lightState and attached to the root of the scene. This light will affect every element added to the scene unless L is pressed (see above), or lightState.setEnabled(false) is called in the client code.

Next, SimpleGame creates and maintains a Timer for tracking frame rate and the time between frames. This time between frames, tpf, is passed into

Table 1.1: SimpleGame's Input System

Key	Action
W	Move Forward
A	Strafe Left
S	Move Backward
D	Strafe Right
Up	Look Up
Down	Look Down
Left	Look Left
Right	Look Right
T	Wireframe Mode on/off
P	Pause On/Off
L	Lights On/Off
C	Print Camera Position
B	Bounding Volumes On/Off
N	Normals On/Off
F1	Take Screenshot

the different update and render methods to notify systems of the speed of the computer. Maintaining the timer is not required for the client.

When running SimpleGame, you will always have the frame rate and triangle/vertex information displayed at the bottom of the screen. This is controlled within the fpsNode Node and fps Text objects. The fps text is automatically updated in SimpleGame's update method.

Perhaps the most important object created by SimpleGame is rootNode. This rootNode is a Spatial Node that defines the root of the scene to render. By adding other Spatial's to rootNode SimpleGame will handle rendering and updating them for the client.

NOTE: All variables (input, cam, rootNode, etc) are set to protected allowing the client to modify them as they feel neccessary.

SimpleGame provides methods that can be overridden to allow for tighter control of the application.

simpleInitGame is the only manditory method to be overridden. This is where all game related data is to be placed. This includes building the scene graph, setting textures, etc. All scene elements should be attached to rootNode. Overriding this method is sufficient to having 3D objects displayed on the screen and being able to move around them.

simpleUpdate provides a way to add other tasks to the update phase of the loop. For instance, you might rotate a Sphere about an axis, or detect collisions. This is called after update but before updating the geometric data of the scene.

simpleRender provides the means to render data that is not contained in the standard rootNode tree. For example, if you are doing any rendering to a

texture you would place these calls in simpleRender. This is called after render but before rendering the statistics (to insure accuracy). Example - Full Demonstration Rendering a Rotating Sphere

Note - This is from jmetest.renderer.TestSphere.

```
//required import statements
import java.io.File;
import java.net.URISyntaxException;
import java.net.URL;
import java.util.logging.Logger;

import com.jme.app.SimpleGame;
import com.jme.bounding.BoundingBox;
import com.jme.image.Texture;
import com.jme.math.Quaternion;
import com.jme.math.Vector3f;
import com.jme.scene.shape.Sphere;
import com.jme.scene.state.AlphaState;
import com.jme.scene.state.TextureState;
import com.jme.util.TextureManager;
import com.jme.util.resource.MultiFormatResourceLocator;
import com.jme.util.resource.ResourceLocatorTool;
import com.jme.util.resource.SimpleResourceLocator;

/**
 * TestSphere extends SimpleGame giving us a basic framework.
 */
public class TestSphere extends SimpleGame {
    private static final Logger logger =
     Logger.getLogger(TestSphere.class.getName());

  //required values for rotating the sphere
  private Quaternion rotQuat = new Quaternion();
  private float angle = 0;
  private Vector3f axis = new Vector3f(1, 1, 0);

  //the Sphere to render
  private Sphere s;

  /**
   * Entry point for the test,
   * @param args
   */
  public static void main(String[] args) {
    TestSphere app = new TestSphere();
    app.setDialogBehaviour(ALWAYS_SHOW_PROPS_DIALOG);
```

```
    app.start();
  }

  /**
   * updates an angle and applies it to a quaternion
   * to rotate the Sphere.
   */
  protected void simpleUpdate() {
    if (tpf < 1) {
      angle = angle + (tpf * 1);
      if (angle > 360) {
        angle = 0;
      }
    }
    rotQuat.fromAngleAxis(angle, axis);
    s.setLocalRotation(rotQuat);
  }

  /**
   * builds the Sphere and applies the Monkey texture.
   */
  protected void simpleInitGame() {
    display.setTitle("jME - Sphere");

    s = new Sphere("Sphere", 63, 50, 25);
    s.setLocalTranslation(new Vector3f(0,0,-40));
    s.setModelBound(new BoundingBox());
    s.updateModelBound();
    rootNode.attachChild(s);

    try {
        MultiFormatResourceLocator loc2 = new MultiFormatResourceLocator
         (new File("c:/").toURI(), ".jpg", ".png", ".tga");
        ResourceLocatorTool.addResourceLocator(
         ResourceLocatorTool.TYPE_TEXTURE,loc2);
    } catch (Exception e) {
        e.printStackTrace();
    }

    URL u = ResourceLocatorTool.locateResource(
     ResourceLocatorTool.TYPE_TEXTURE, "/model/grass.gif");
    System.err.println("FOUND URL: "+u);

    TextureState ts = display.getRenderer().createTextureState();
    ts.setEnabled(true);
    ts.setTexture(
```

```
        TextureManager.loadTexture(u,
        Texture.MM_LINEAR_LINEAR,
        Texture.FM_LINEAR));

    rootNode.setRenderState(ts);

    AlphaState alpha = display.getRenderer().createAlphaState();
    alpha.setBlendEnabled(true);
    alpha.setSrcFunction(AlphaState.SB_SRC_ALPHA);
    alpha.setDstFunction(AlphaState.DB_ONE_MINUS_SRC_ALPHA);
    alpha.setTestEnabled(true);
    alpha.setTestFunction(AlphaState.TF_GREATER);
    alpha.setEnabled(true);
    rootNode.setRenderState(alpha);
  }
}
```

SimplePassGame & SimpleHeadlessGame

Additionally, there are SimplePassGame and SimpleHeadlessGame that provide the means to quickly build multi-pass applications and headless renderering applications (useful for embedding in Swing/AWT). Similar to SimpleGame these classes handle all the needed set-up for creating a quick scene in which to render objects.

The next three game types provide a variation on the BaseGame type that will provide ways to control the speed at which events occur in the game world.

FixedFramerateGame

FixedFramerateGame allows the user to set the maximum framerate of the game. The game loop is guaranteed never to run faster than the limit, but nothing guarantees that it will not run slower. Limiting the framerate of the game is useful for playing nice with the underlying operating system. That is, if the game loop is left unchecked, it tends to use 100

FixedLogicrateGame

FixedLogicrateGame allows the renderer to render as fast as possible but only giving a defined amount of time to the update method. This ensures that the game state will be updated at a specified rate, while the rendering occurs as fast as the computer can handle it. This gives tighter control on how the game state is processed, including such things as AI and physics.

VariableTimestepGame

Last, VariableTimestepGame is very similar to BaseGame. However, instead of just calling the basic update/render methods, they are supplied with the time between frames. This saves the user from having to create and maintain a Timer object.

StandardGame

Excerpted from StandardGame, GameStates, and Multithreading (A New Way of Thinking):

StandardGame extends AbstractGame and intends to implement all necessary functionality that your Game will need. Support includes client/server division without a change in any code, GameSettings as an alternative to the PropertiesIO system that the other games provide, inherent (forced) multithreading in your game as StandardGame manages the OpenGL thread for you, ability to directly interject necessary invocations into the OpenGL thread using the GameTaskQueueManager, shadow support, reinitialization of the graphical context (if settings change for example), and everything else a typical Game provides that all games need. However, with all of the aspects that StandardGame does provide, it does not force anything extra on you as the non-necessary items should be put into your GameStates and managed there instead of the Game itself. This process helps to abstract into aspects of your game and get the game process started ASAP to kill the long-standing problem in jME of the lagged startup because theres just so much junk youre trying to load that you sit with a black screen for a while until everything is ready to show. Resources

- StandardGame Java Docs
- StandardGame, GameStates, and Multithreading (A New Way of Thinking) - A paradigm shift by darkfrog
- SimpleGame to StandardGame - An effort to gain mindshare by darkfrog
- What calls must be made from the OpenGL Thread?
- Some StandardGame frequently asked questions - Collected from the user forums

1.5 DisplaySystem

The DisplaySystem provides a means to communicate with the windowing system. The DisplaySystem class provides the user with an abstraction that hides the actual windowing implementation. This allows for the user to write

Table 1.2: Graphics Adapter Info

Method	Information	Example Output
getAdapter	The name of the graphics adapter.	ati2dvag
getDisplayAPIVersion	The OpenGL API Version supported.	2.0.6174 WinXP Release
getDisplayRenderer	The name of the renderer. Specific to a particular configuration of a hardware platform.	Radeon X1800 Series x86/SSE2
getDisplayVendor	Company responsible for the GL implementation.	ATI Technologies Inc.
getDriverVersion	The version or release number.	6.14.10.6648

the application a single time and switching rendering systems without changing a line of code. Because DisplaySystem is a factory class, a call to getDisplaySystem with a string identifying the preferred Renderer is all that is required. Using PropertiesIO, the configuration file can change this value, preserving the original application source code.

DisplaySystem has two main jobs. To build the window and to build the Renderer. Both occur using the createWindow method. The method takes the width and height of the window (resolution), the frequency of the monitor and a boolean determining if it is fullscreen or not. These values are used to create the window, and then create the Renderer which in turn provides the OpenGL context. At this point the system is prepared for rendering.

DisplaySystem also provides utility methods for setting various attributes of the windowing and rendering system. First, it provides multiple getter methods for obtaining the current window and render state. getDisplay() without the string parameter will return the current DisplaySystem object. getWidth and getHeight return the resolution of the window. getRenderer will return the Renderer object, while getRendererType returns the type safe constant of which renderer is being used (i.e. LWJGL, JOGL, etc).

Information about the user's graphics adapter can be obtained from DisplaySystem. getAdapter, getDisplayAPIVersion, getDisplayRenderer, getDisplayVendor and getDriverVersion provide all information available about the adapter.

If you require non-default graphic settings. That is, depth and alpha bits less than what your card can handle (by default, the bits are set to the cards maximum), you may call the get/set methods for AlphaBits and DepthBits.

DisplaySystem also provides the means for rendering to a texture. A TextureRenderer can be obtained by calling the createTextureRenderer method. This special renderer displays its output to a texture rather than the window.

DisplaySystem allows the user to create a Headless window. Which is a rendering target that is never shown on screen. This allows you to store a ren-

dering to an offline buffer and apply it elsewhere (save to a file, attach to a Swing/AWT context, etc). You can create both a headless window as well as a standard window from the same DisplaySystem factory. To create a headless window, calling createHeadlessWindow with the height and width of the display and the color depth is required.

DisplaySystem gives the user the ability to calculate the screen coordinate of a world location and vice versa. Using the getScreenCoordinates method, the user may supply a world position and obtain the approximate screen location. This is used for such things as displaying text above an in-game character, or displaying a lens flare. The reverse may be accomplished by calling getWorldCoordinates.

The SimpleGame application type will take care of building the DisplaySystem for you, so you will not see DisplaySystem code in future examples. However, to insure a firm understanding of the system, the first example will show a complete application that extends BaseClass. This will give a good guideline on how to use the DisplaySystem.

1.5.1 Example 1 - Creating a DisplaySystem

```
//Creating a display can fail, in such a case a JmeException is thrown.
//We must therefore, catch it.
try {
      //we will create an LWJGL display, typically you would obtain
      //the values from the properties dialog.
      DisplaySystem display = DisplaySystem.getDisplaySystem("LWJGL");

      //We'll create a 640x480x32 fullscreen display with a 60Hz monitor
      //again these values will be typically obtained from the properties
      display.createWindow(640,480,32,60,true);
    }
    catch (JmeException e) {
      e.printStackTrace();
      System.exit(1);
    }
```

1.5.2 Example 2 - Getting Screen Coordinates

```
//We wish to obtain the screen coordinate (screen) using a
//world coordinate (world)
DisplaySystem.getDisplaySystem().getScreenCoordinate(world, screen);
//We can now use screen however we wish.
//We can convert back if desired (they should be the same as the
//original world)
world = DisplaySystem.getDisplaySystem().getWorldCoordinate(screen, 1);
```

1.5.3 LWJGLDisplaySystem

LWJGLDisplaySystem is a concrete implementation of the DisplaySystem abstract class using the LWJGL API.

1.6 Logging System

Note: The LoggingSystem class in jME has been removed, instead now the java class Logger is used

The Logger class is used internally by jME to write log messages to standard error output. Here's an example message written by the Logger:

```
Nov 5, 2007 3:01:30 PM com.jme.system.lwjgl.LWJGLDisplaySystem <init>
INFO: LWJGL Display System created.
```

1.6.1 Example 1 - Using the Logger class

```
// somewhere in your class
static final Logger logger = Logger.getLogger(MyClass.class.getName());

//...
public void someMethod(String aString){
   if (aString == null)
       logger.log(Level.WARNING, "Invalid input!");
}
```

1.7 Renderer

The Renderer defines exactly how geometrical elements will be displayed to the screen. It's an abstract class that does not particularly care about how the implementing class is displaying the information. This also means, that a jME application need not care how the actual rendering is being handled. It could be making use of the LWJGL API, JOGL API, a software renderer, or any other rendering API that might be supported.

The concrete Renderer object will be created by the DisplaySystem class during the creation of the window. This is the object that all jME applications should use for displaying graphics.

Renderer also provides the means to obtain RenderState objects. This method of obtaining the RenderState objects ensures that the properly configured state is provided to you. This also means, that you can, at any time, change the Renderer type without affecting the jME application.

Renderer provides the means for the client application to interact with the display buffer. This includes rendering geometry, clearing the screen, setting the mode (Ortho vs. Perspective), etc. Typically, the display method of the jME

application will consist of two calls to the Renderer, clearing the buffers and rendering the data. Creation

As stated above, creation of the Renderer occurs during the creation of the DisplaySystem. The DisplaySystem creates the type of Renderer requested by the user. For instance, if an LWJGLDisplaySystem is created (DisplaySystem.getDisplaySystem(LWJGL)) then it will create an instance of LWJGLRenderer.

1.7.1 Drawing

You may then obtain the Renderer via DisplaySystem's getRenderer method. You can then draw (actually send data for display) any Geometry (display. getRenderer(). draw(myGeometry)), and swap the buffers to show the data on the screen.

The order of which the geometry is rendered is dependant on its place in the RenderQueue and can be found explained in detail there. However, Renderer maintains constants: **QUEUE_INHERIT, QUEUE_SKIP, QUEUE_TRANSPARENT, QUEUE_OPAQUE, QUEUE_ORTHO**. The standard steps for drawing using the renderer are:

1. Clear the current buffer
2. Call the draw method on the Geometry to render
3. Display the back buffer (the new screen)

Statistics to the scene (Number of vertices and Number of triangles rendered) can be obtained via the Renderer. This is the data that is commonly shown on the SimpleGame demos. This is done via clearStatistics (to reset the count for the frame) and getStatistics (to get the count for the frame). Statistics can be enabled or disabled via a call to enableStatistics.

Additionally, Renderer is capable of taking a screenshot of the current display. This basically involves obtaining the data from the last rendered frame and writing it out to a PNG image. A call to takeScreenShot with the filename (the filename should not contain the file extension, as it's always going to be .png). This PNG image will be written out to the running directory.

1.7.2 Example 1 - Creating and Obtaining the Renderer

```
//Create a display system using the values from the properties dialog
DisplaySystem display = DisplaySystem.getDisplaySystem(
properties.getRenderer());
/** Create a window with the startup box's information. */
display.createWindow(
          properties.getWidth(),
          properties.getHeight(),
          properties.getDepth(),
```

```
          properties.getFreq(),
          properties.getFullscreen());

//Now that the display system is created, we can obtain
//the appropriate Renderer
Renderer r = display.getRenderer();
```

In the above example PropertiesIO was used to obtain: Renderer type, width, height, depth, frequency and fullscreen mode.

1.7.3 Example 2 - Clear, Draw and Display

```
//A main game loop that continually renders a piece of Geometry (myGeom)
//We are assuming the display has been created above
while (!finished) {

    //clear the buffer
    display.getRenderer().clearBuffers();
    //draw the geometry
    display.getRenderer().draw(myGeom);
    //swap the buffers
    display.getRenderer().displayBackBuffer();
}
```

1.7.4 Example 3 - Obtain the Rendering Statistics

```
{}
//This code displays some geometry and prints the statistics to the
//console.

display.getRenderer().enableStatistics(true);

while (!finished) {

    //clear the buffer
    display.getRenderer().clearBuffers();
    //clear the stats
    display.getRenderer().clearStatistics();
    //draw the geometry
    display.getRenderer().draw(myGeom);
    //Print out the stats
    System.out.println(display.getRenderer().getStatistics());
    //swap the buffers
    display.getRenderer().displayBackBuffer();
}
```

1.7.5 LWJGLRenderer

LWJGLRenderer is a concrete implementation of the Renderer abstract class using the LWJGL API.

1.8 Texture Renderer

TextureRenderer provides the means to render entire scenes to a texture rather than the screen itself. This texture can then be applied to geometry like any other texture. TextureRenderer can use Frame Buffer Object if it is supported or PBuffer and will make use of Direct Render to Texture if the card supports it, otherwise, the OpenGL function glCopyTexImage2D is used. Safe guards are in place to insure that TextureRenderer renders closest to the parameters given and supported by the user's graphics card. Usage

Use of the TextureRenderer is very similar to Renderer. It maintains its own Camera allowing the view point to be different than that of the main scene. Just like Renderer creating the TextureRenderer is done (in most cases) via the DisplaySystem class. Scenes can then be rendered to a supplied texture with a call to the render method.

TextureRenderer is very useful to various techniques: Imposters, Cube Mapping, miscellaneous effects such as mirrors, video camera monitors, etc.

1.8.1 Creating

The createTextureRenderer has a few parameters. First, is the width and height of the texture to render. In the case of the example, this is a 512512 texture. Any size is usable (the larger the better the quality), but the size should be reasonable to avoid speed issues. Next, you set the target type, this is a constant defined in TextureRenderer and can be:

- RENDER_TEXTURE_1D - Use for creating a one dimensional texture.
- RENDER_TEXTURE_2D - Use for creating a two dimensional texture.
- RENDER_TEXTURE_RECTANGLE - Use for creating a rectangular texture.
- RENDER_TEXTURE_CUBE_MAP - Use for creating a cube map (six sided texture).

1.8.2 Example 1 - Create a TextureRenderer, Set the Camera and Get the Texture

```
//display and camera are created elsewhere
//Create the texture renderer
TextureRenderer tRenderer = display.createTextureRenderer(512, 512,
```

```
TextureRenderer.RENDER_TEXTURE_2D);
//set the texture renderer's camera to the left and back.
tRenderer.getCamera().setLocation(new Vector3f(-10, 0, 15f));
tRenderer.updateCamera();

//create and setup the texture so we can use it in the main scene.
Texture trTexture = new Texture();
tRenderer.setupTexture(trTexture);
```

1.8.3 Applying

First, we are making a couple assumptions in this example. All the objects have been defined else where and the main scene (containing the Quad) are rendered elsewhere. In the example, we create a Sphere and a Quad. Only the Quad is used in the main scene, while the Sphere is being rendered to a texture. This texture (fakeTex) is applied to the quad. Therefore, whenever we render to the texture, this new rendering is applied to the texture, and shown on the Quad.

1.8.4 Example 2 - Render to the Texture and Display it on a Quad

```
protected void drawTextureStuff() {
      tRenderer.render(sphere, fakeTex);
  }

  protected void initializeStuff() {
    tRenderer = display.createTextureRenderer(512, 512,
     TextureRenderer.RENDER_TEXTURE_2D);

    sphere = new Sphere("Sphere", 25, 25, 5);

    q = new Quad("Quad", 15, 13f);
    rootNode.attachChild(q);

    tRenderer.setBackgroundColor(new ColorRGBA(0f, 0f, 0f, 1f));
    fakeTex = new Texture();
    tRenderer.setupTexture(fakeTex);
    TextureState screen = display.getRenderer().createTextureState();
    screen.setTexture(fakeTex);

    tRenderer.getCamera().setLocation(new Vector3f(-10, 0, 15f));
    tRenderer.updateCamera();
    q.setRenderState(screen);
  }
```

1.8.5 Rendering Other Buffers

TextureRenderer also allows you to render other buffers to a texture other than just the color buffer. You can render RGB, RGBA, Depth, Luminance, Luminance with Alpha, alpha, and intensity. This option is set up in Texture using its setRTTSource method. By setting this value, use TextureRenderer as normal, the resulting texture will contain the values specified.

1.8.6 Example 3 - Rendering the Depth Buffer to a Texture

```
protected void drawTextureStuff() {
    tRenderer.render(sphere, fakeTex);
}

protected void initializeStuff() {
  tRenderer = display.createTextureRenderer(512, 512,
   TextureRenderer.RENDER_TEXTURE_2D);

  sphere = new Sphere("Sphere", 25, 25, 5);

  q = new Quad("Quad", 15, 13f);
  rootNode.attachChild(q);

  tRenderer.setBackgroundColor(new ColorRGBA(0f, 0f, 0f, 1f));
  Texture fakeTex = new Texture();
  fakeTex.setRTTSource(Texture.RTT_SOURCE_DEPTH);
  tRenderer.setupTexture(fakeTex);
  TextureState screen = display.getRenderer().createTextureState();
  screen.setTexture(fakeTex);

  tRenderer.getCamera().setLocation(new Vector3f(-10, 0, 15f));
  tRenderer.updateCamera();
  q.setRenderState(screen);
}
```

1.9 RenderQueue

The render queue provides the means to organize scene elements prior to drawing them. In simple scenes, this is not required and can be ignored. However, it doesn't take long before proper Geometry sorting is required to prevent any rendering artifacts. For example, by default Geometry is rendered by its position in the Scene Graph. If there are transparent objects in the graph, and they are drawn before any opaque objects that are supposed to be behind it, those objects will not appear visible through the transparent object. This is where RenderQueue comes into play. RenderQueue makes use of a number of dif-

ferent buckets: opaque, transparent and ortho. Renderer defines the constants and to set a queue you use the method setRenderQueueMode in Spatial. For example, to place some Geometry into the transparent queue:

```
geom.setRenderQueueMode(Renderer.QUEUE_TRANSPARENT);
```

Once objects are in the queues, they are preprocessed and ordered before being drawn. The ordering is as follows:

1. Draw the opaque objects first (front to back)1)

2. Draw transparent objects (back to front) drawing the inside of the object.

3. Draw transparent objects (back to front) drawing the outside of the object.

4. Draw Ortho objects.

The order is important for the following reasons:

- Drawing opaque first insures that any opaque objects will be visible from behind transparent objects. Drawing front to back allows OpenGL to optimize by throwing out occluded pixels.

- Drawing transparent objects from back to front insures that all objects are visible from behind a transparent object, even other transparent objects.

- Drawing the transparent objects twice (once rendering the inside, again the outside), allows complex objects to appear truly transparent.2)

- Ortho objects are drawn last because these are screen oriented (UI, HUD, etc) graphics and should always be on top of the rest of the scene.

RenderQueue modes are inherited from a parent, so special care needs to be taken if a branch of the scene graph has different queue elements contained within it.

RenderQueue handles all processing and requires no work from the user. Sorting of the Geometry occurs dynamically as needed. The user must only set the RenderQueue mode.

1) front/back means the distance from the camera, front being close, back being far 2) Transparent objects can be only rendered once by calling RenderQueue.setTwoPassTransparency(false).

1.10 Multiple Rendering Passes

Multiple render passes are used to create certain effects which are impossible with single-pass rendering, they are also used to apply certain effects to a scene as it is being renderered.

1.10.1 Usage

In the jME framework, the class BasicPassManager is used to manage the passes and render them. Pass is an abstract class which is extended to provide specialized rendering, while RenderPass is a subclass which renderes the scene as usual. All jME's passes are stored in the com.jme.renderer.pass, com.jmex.effects.water and com.jmex.effects.glsl packages.

1.10.2 RenderPass

The order in which RenderPass objects are added to the BasicPassManager via BasicPassManager.add() determines the z-order of the scene graphs. For example, the following code, placed at the end of SimplePassGame.simpleInitGame(), results in the frames-per-second text always appearing in front of the root node hierarchy's geometry:

```
ShadowedRenderPass sPass = new ShadowedRenderPass();
sPass.add(rootNode);
sPass.setRenderShadows(true);
sPass.setLightingMethod(ShadowedRenderPass.ADDITIVE);
pManager.add(sPass);

RenderPass rPass = new RenderPass();
rPass.add(fpsNode);
pManager.add(rPass);
```

1.10.3 Simple Shadows Example

I did this test because I coudn't find a simple example of shadows casting.

```
import com.jme.app.SimplePassGame;
import com.jme.light.DirectionalLight;
import com.jme.light.PointLight;
import com.jme.math.FastMath;
import com.jme.math.Quaternion;
import com.jme.math.Vector3f;
import com.jme.renderer.ColorRGBA;
import com.jme.renderer.Renderer;
import com.jme.renderer.pass.RenderPass;
import com.jme.renderer.pass.ShadowedRenderPass;
import com.jme.scene.Node;
import com.jme.scene.shape.Box;
import com.jme.scene.shape.Quad;
import com.jme.scene.shape.Sphere;

/**
 *
```

```
 *
 */
public class TestSimpleShadoPass extends SimplePassGame {
    private static ShadowedRenderPass sPass = new ShadowedRenderPass();

    /**
     * Entry point for the test,
     *
     * @param args
     */
    public static void main(String[] args) {
TestSimpleShadoPass app = new TestSimpleShadoPass();
app.setDialogBehaviour(ALWAYS_SHOW_PROPS_DIALOG);
app.start();
    }

    TestSimpleShadoPass() {
stencilBits = 4;
    }

    /**
     * builds the scene.
     *
     * @see com.jme.app.BaseGame#initGame()
     */
    protected void simpleInitGame() {
display.getRenderer().setBackgroundColor(ColorRGBA.gray.clone());

// A rotated quad that sits on x-z plane.
Quad floor = new Quad("Floor", 15, 15);
rootNode.attachChild(floor);
Quaternion r = new Quaternion();
r.fromAngleAxis(-FastMath.PI / 2, new Vector3f(1, 0, 0));
floor.setLocalRotation(r);

// A box and a sphere to occlude the light.
Node occluders = new Node("Occluders");
Box b = new Box("Box", new Vector3f(-2, 0, -2), new Vector3f(1, 4, 1));
occluders.attachChild(b);
Sphere s = new Sphere("Sphere", 24, 24, 1.5f);
s.setLocalTranslation(new Vector3f(3, 1.5f, 0));
occluders.attachChild(s);
rootNode.attachChild(occluders);

// The light that cast shadows.
DirectionalLight dr = new DirectionalLight();
```

```
dr.setEnabled(true);
dr.setDiffuse(new ColorRGBA(1.0f, 1.0f, 1.0f, 1.0f));
dr.setAmbient(new ColorRGBA(.6f, .6f, .5f, .5f));
dr.setDirection(new Vector3f(0.5f, -0.8f, 0.5f).normalizeLocal());
dr.setShadowCaster(true);

// Another light so there is no full darkness.
PointLight pl = new PointLight();
pl.setEnabled(true);
pl.setDiffuse(new ColorRGBA(.2f, .2f, .23f, 1.0f));
pl.setAmbient(new ColorRGBA(.25f, .25f, .28f, .25f));
pl.setLocation(new Vector3f(20, 20, 20));

lightState.detachAll();
lightState.attach(dr);
lightState.attach(pl);
lightState.setGlobalAmbient(new ColorRGBA(0.8f, 0.8f, 0.8f, 1.0f));

// Set the cam.
cam.setLocation(new Vector3f(0, 15, -15));
cam.lookAt(new Vector3f(0, 0, 0), new Vector3f(0, 1, 0));

rootNode.setRenderQueueMode(Renderer.QUEUE_OPAQUE);

sPass.add(rootNode);
sPass.addOccluder(occluders);
sPass.setRenderShadows(true);
sPass.setLightingMethod(ShadowedRenderPass.ADDITIVE);
pManager.add(sPass);

RenderPass rPass = new RenderPass();
rPass.add(fpsNode);
pManager.add(rPass);
    }
}
```

1.11 Writing a simple jME enabled Java Applet

1.11.1 Before you begin

First, ask yourself if you really need to run in an applet. There are a few restrictions when using applets, so your application may be better suited to java web start or a standalone java application. For example, max memory in an applet is somewhere between 64 and 96MB by default. The only way to increase that is to make the end user go into their java control panel and manually type in a

java command line arg - not very likely to happen in most cases. Also, because applets are AWT based, you will be using jME's AWT/Swing support which means you have to use the AWT input system instead of jinput and your performance will be maybe a third or more impacted versus running in a regular opengl window. Let's Begin! Writing the Applet Writing your first Java applet that uses jMonkeyEngine is easy if you follow these few simple steps:

1.11.2 Step One

Like SimpleGame, jMonkeyEngine has a new simple-type base class for writing your first applet. This class is called com.jmex.awt.applet.SimpleJMEApplet. We're going to recreate the classic jme test TestBoxColor in an applet, so let's start off by extending that class and declaring a few variables we'll need in later steps:

```
public class AppletTestBoxColor extends SimpleJMEApplet {
    private TriMesh t;
    private Quaternion rotQuat;
    private float angle = 0;
    private Vector3f axis;
}
```

What to note:

We're going to assume you are familiar with jME code in general, so there isn't too much to explain at this step. Extending SimpleJMEApplet gives us the familiar FPS/Stats text at the bottom of the applet. It also sets up a wireframe state, zbufferstate and basic lightstate with a single PointLight. You can enabled/disable the lights, wireframe, normals debugging, bounds debugging and so forth with the same keystrokes as SimpleGame derived apps.

1.11.3 Step Two

To populate our 3d scene, we extend the method simpleAppletSetup. Here's the code:

```
    public void simpleAppletSetup() {
        getLightState().setEnabled(false);
        rotQuat = new Quaternion();
        axis = new Vector3f(1, 1, 0.5f).normalizeLocal();

        Vector3f max = new Vector3f(5, 5, 5);
        Vector3f min = new Vector3f(-5, -5, -5);

        t = new Box("Box", min, max);
        t.setModelBound(new BoundingBox());
        t.updateModelBound();
```

```
        t.setLocalTranslation(new Vector3f(0, 0, -15));
        getRootNode().attachChild(t);

        t.setRandomColors();

        TextureState ts = getRenderer().createTextureState();
        ts.setEnabled(true);
        ts.setTexture(TextureManager.loadTexture(
                TestBoxColor.class.getClassLoader().getResource(
                        "jmetest/data/images/Monkey.png"), Texture.MM_LINEAR,
                Texture.FM_LINEAR));

        getRootNode().setRenderState(ts);
    }
```

What to note:

Unlike SimpleGame apps, you don't have direct access to many of the variables. Instead, use the provided getXXXX methods. Currently those are: getCamera, getRenderer, getRootNode, getFPSNode, getTimePerFrame, get/setInputHandler, getLightState and getWireframeState. So instead of using display.getRenderer().createXXXXState() you just use getRenderer().createXXXXState(). Everything else is pretty much the same. Step Three

Well, we've got our box in the scene. Now we want it to rotate each frame. To do that, we override the method simpleAppletUpdate like so:

```
    public void simpleAppletUpdate() {
        float tpf = getTimePerFrame();
        if (tpf < 1) {
            angle = angle + (tpf * 25);
            if (angle > 360) {
                angle -= 360;
            }
        }

        rotQuat.fromAngleNormalAxis(angle * FastMath.DEG_TO_RAD, axis);
        t.setLocalRotation(rotQuat);
    }
```

What to note:

Nothing really odd going on here. Again note the usage of the get methods. Step Four

And that's it, now you have an applet that shows off a colorful spinning cube. Now you just need to package it up and show it to the world. Show it off! Deploying your Applet Setting up the page

To deploy your applet on a webpage, you must first create the html page that it will appear on. This html page contains the standard applet tag that

Table 1.3: Jars

Jars	Description
lwjgl_util_applet.jar	Used for the LWJGLInstaller, a class that installs the proper library files during load.
lwjgl.jar	LWJGL API used for low level OpenGL, Sound and Input Bindings.
jme.jar	jME API used as the middle-ware game engine.
natives.jar	This jar contains the native bindings to OpenGL, sound API and input systems. (this file can be found in the lwjgl applettest or lwjgl-applet download)
jme-awt.jar	This jar contains JME applet support stuff and other AWT jME extensions.
my.jar	This jar would contain the user created code.

defines the code that will be run in the browser. For example, if we created an Applet called MyApplet you might load it in a page as:

```
<html> <body> <h1>MyApplet Page</h1> <center>
<applet code="my.own.jar.package.MyApplet"
archive="lwjgl_util_applet.jar,lwjgl.jar,jme.jar,natives.jar,
jme-awt.jar,my.jar" width="800" height="600">
<param name="useAppletCanvasSize" value="true"/>
</applet> </center> </body> </html>
```

This applet page defines the applet to run (MyApplet) in the namespace my.own.jar.package. The required jars are defined in the archive sections. This section lists are the jars required to run your Applet.

These jars represent the minimum set-up to get a Applet up and running using jME. You may need to add jars to this resource as you use them. For example, if you add FMOD sound you'd add the lwjgl_fmod3.jar as well as jorbis and jogg jars.

If you want to run the example above, you need to add two jars from the jME test project: jmetest.jar for the TestBoxColor class jmetest-data-images.jar this jar contains the image for the texture (Monkey.png)

These jars must also live in the proper location that is being referenced in the html page. In our example above we would need to put all the referenced jars in the same directory as the html code.

The ¡param name=useAppletCanvasSize value=true/¿ tells the jME applet to create a glCanvas with a resolution equal to the specified applet size, if not specified the glCanvas of default (640480) resolution will be created.

1.11.4 Signing

You might notice that when running some of the examples that a dialog appears stating The application's digital signature is invalid. Do you want to run the application? where it then lists the name of the publisher and gives the option to run or cancel. This is because the jars are self-signed. That is, the developer created his/her own certificates rather than purchasing a certificate from an authority. This will always appear when self-signing as a security measure. For this dialog to not appear, a valid certificate must be purchased. See Deployment Tutorial for information on how to self-sign.

For purchasing of a digital signature see such services as verisign. A dialog box will still appear for the user to accept, but this dialog will be much friendlier with a verified signature. I.e. users will be much less likely to be scared off.

1.11.5 Issues and Gotchas

1.11.6 Cleaning up on exit

If you have OpenGL (rendering) resources that need to be cleaned up properly on exit, these must be called from the same thread in which they were created (the OpenGL thread). In most cases, this thread is the AWTEventQueue thread. Therefore, if you have specific resources, such as a PBuffer from rendering to texture, you'll need to handle this yourself in the stop() method of the Applet.

The following example will show how to handle cleaning up of a PBuffer.

You'll want to overwrite the stop() method to tell the applet to clean up before the destroy.

```
public void stop() {
  status = STATUS_DESTROYING;
  long time = System.currentTimeMillis();
  while (status != STATUS_DEAD && (System.currentTimeMillis() - time)
                              < 5000){ // only keep waiting for 5 secs
      try {
          Thread.sleep(100);
      } catch (InterruptedException e) {
          e.printStackTrace();
      }
  }
super.stop();
}
```

You'll notice that you first want to set the status to STATUS_DESTROYING, this will tell the main render queue that we are now destroying objects. We will then ensure that things have time to clean up, waiting on the status to change to STATUS_DEAD. Once STATUS_DEAD occurs we can let the applet

close normally. To prevent lock-up we put a 5 second timer on the wait, if the close takes longer than this, we have larger problems to resolve.

Of course, you'll also need to let the applet know when you've finished destroying things. Typically you would do that in either simpleAppletRender or simpleAppletUpdate. simpleAppletRender is generally safer since it happens at the end of a paint cycle. Here's an example:

```
public void simpleAppletRender() {
    if (status == STATUS_DESTROYING) {
        imposterNode.getTextureRenderer().cleanup();
        status = STATUS_DEAD;
    }
}
```

1.11.7 Threads and OpenGL

You can only call OpenGL from a single thread; the thread in which the OpenGL context was created. The thread that creates OpenGL is the AWTEventQueue. To insure that you are not calling OpenGL inappropriately, only use simpleAppletRender(), simpleAppletUpdate() and simpleAppletSetup() for any calls that might touch it.

1.11.8 Memory Usage

To reiterate the caution from the first section on this page, applets have a fixed amount of available memory that can not be increased by the programmer (unlike, for example, a webstart application where you can set your max memory size.) Thus, you are stuck with whatever the user has setup in their Java control panel - typically 64MB-96MB. So keep your object creation and retention as low as possible.

1.11.9 Next Steps

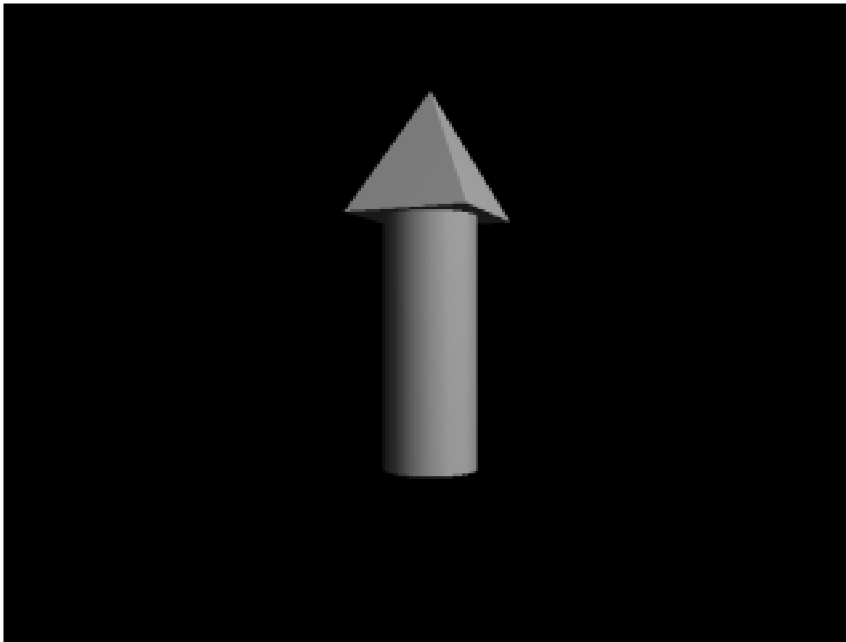
Now that you've got the basics down, open up the source code of SimpleJMEApplet and start poking around. Examine the automatic resizing, the repainting thread, the awt-based key and mouse input and the relationship to jme's awt canvas classes. Then try writing your next applet by extending Applet or JApplet and leave the SimpleGame crutch behind. :)

1.12 Fundamental Shapes

jME provides the means for users to create their own Geometry classes as TriMesh objects. However, creating these by hand can be tedious and unnecessary. In addtion to Models, jME provides you the means of creating basic geometric shapes. These 3D shapes will allow you to put standard shapes into

the scene without much trouble. This is useful for building up more complex shapes, or just using them as placeholders for more complicated models later.

1.12.1 Arrow



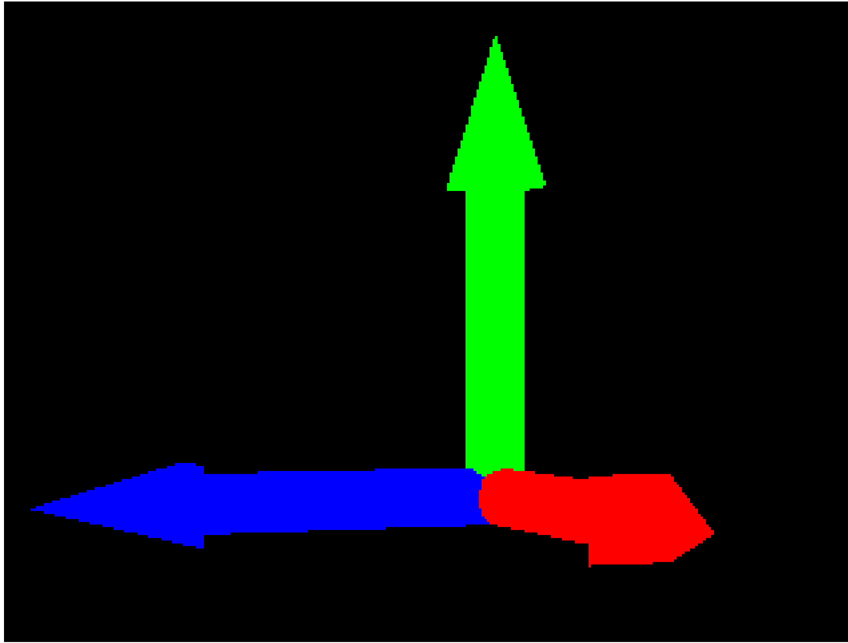
An Arrow provides a shape that indicates direction or placement.

In jME an Arrow is represented by a cylinder with a pyramid cap. To define the arrow, the length and width is provided. The arrow will always face (0, 1, 0), where the user will orient the arrow as needed using the Arrow's local rotation.

Example - Creating an Arrow

```
//Create a arrow with a length of 10 and width of 5
Arrow a = new Arrow("Arrow", 10, 5);
```

1.12.2 Axis Rods



Axis Rods allow the visualization of coordinate space. The rods provide three arrow representations for each major axis of the nodes orientation. Each axis is color coded to provide a quick visual distinction, where Red is the X Axis, Green is the Y Axis and Blue is the Z Axis.

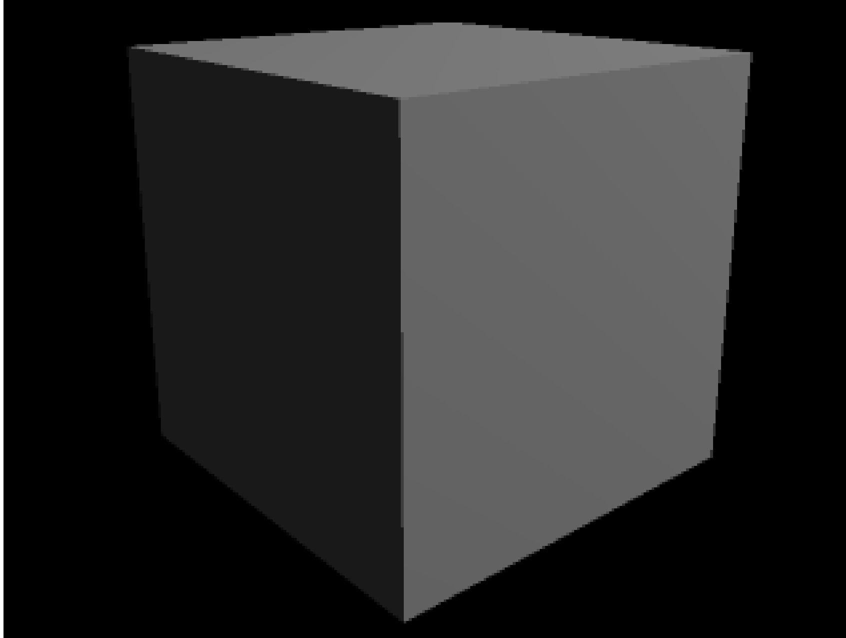
Axis Rods can be set to represent a right handed coordinate system or a left handed coordinate system.

To create an AxisRods object, provide the name of the rods, a boolean defining if it is right handed or left (true is right handed, false is left handed) and the scale of the rods. Once the object is created, you can attach it to any part of the scenegraph. It will inherit the parent's orientation and display this.

Example - Creating Axis Rods

```
//Create an right handed axisrods object with a scale of 1/2
AxisRods ar = new AxisRods("rods", true, 0.5f);
//Attach ar to the node we want to visualize
someNode.attachChild(ar);
```

1.12.3 Box



Box provides a TriMesh that is an enclosed shape with six sides that are Axis-aligned. The Box generated is based upon a number of possible criteria, first a minimum and maximum point. All eight points are generated from these two points. Secondly, you may provide the box with a center point and a number of extents from that center for each axis.

Example 1 - Generate a Box with Minimum and Maximum Points

```
//We will create a box with sides of length 10 and will be centered
//about the origin.
Vector3f min = new Vector3f(-5, -5, -5);
Vector3f max = new Vector3f(5, 5, 5);
Box b = new Box("box", min, max);

//Let’s change the box size
//Making it twice as wide
Vector3f min2 = new Vector3f(-10, -5, -5);
Vector3f max2 = new Vector3f(10, 5, 5);
b.setData(min2, max2);
```

Example 2 - Generate a Box using Center and Extents

```
//We will create a box with sides of length 10 and will be centered
//about the origin.
```

```
Vector3f center = new Vector3f(0, 0, 0);
Box b = new Box("box", center, 5, 5, 5);

//Let's change the box size
//Making it twice as wide
b.setData(center, 10, 5, 5);
```

1.12.4 Rounded Box

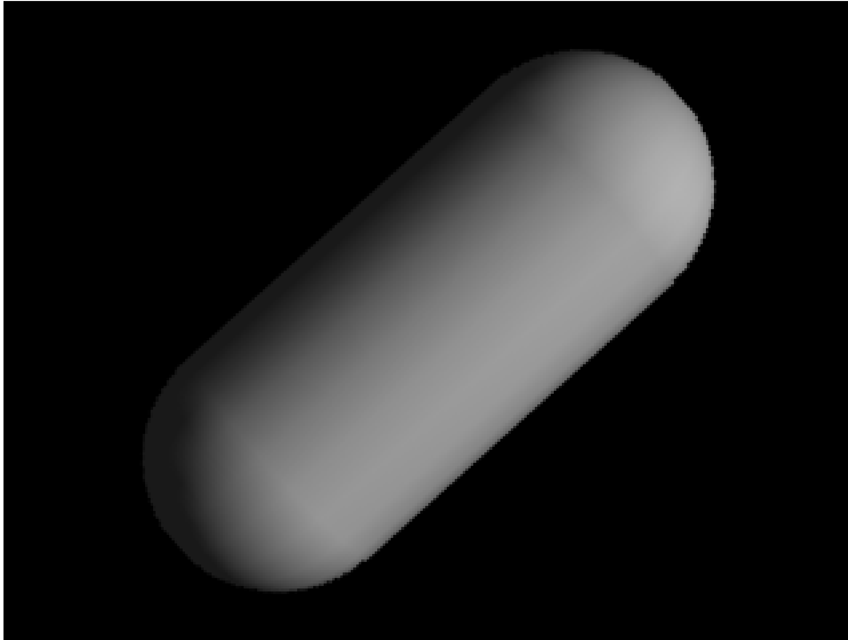
A rounded box is obviously very similar to a box but the constructor takes one additional Vector3f parameter: slope.

FIXME I am not sure what values the slope should be. The edges are super pointy and do not look rounded at all.

Example - Generate a Rounded Box with Minimum and Maximum Points and Slope

```
Vector3f min = new Vector3f(-5, -5, -5);
Vector3f max = new Vector3f(5, 5, 5);
Vector3f slope = new Vector3f(-0.1F, -0.1F, -0.1F);
RoundedBox rb = new RoundedBox("rounded box", min, max, slope);
rootNode.attachChild(rb);
```

1.12.5 Capsule

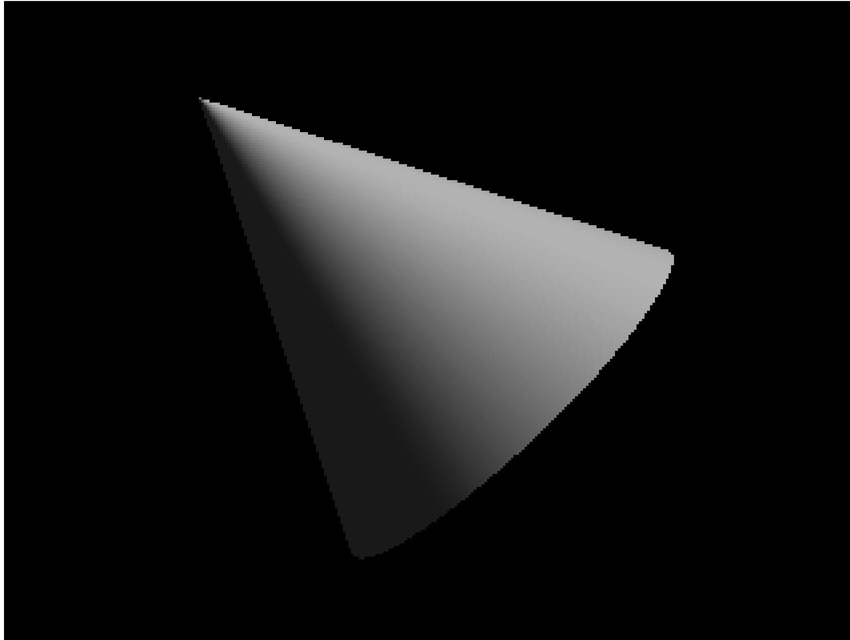


A finite filled Cylinder with rounded edge. It looks like a pill.

Example - Creating a Capusle

```
int axisSamples = 10;
int radialSamples = 10;
int sphereSamples = 10;
float radius = 2;
float height = 5;
Capsule c = new Capsule("capsule", axisSamples,
radialSamples, sphereSamples, radius, height);
rootNode.attachChild(c);
```

1.12.6 Cone

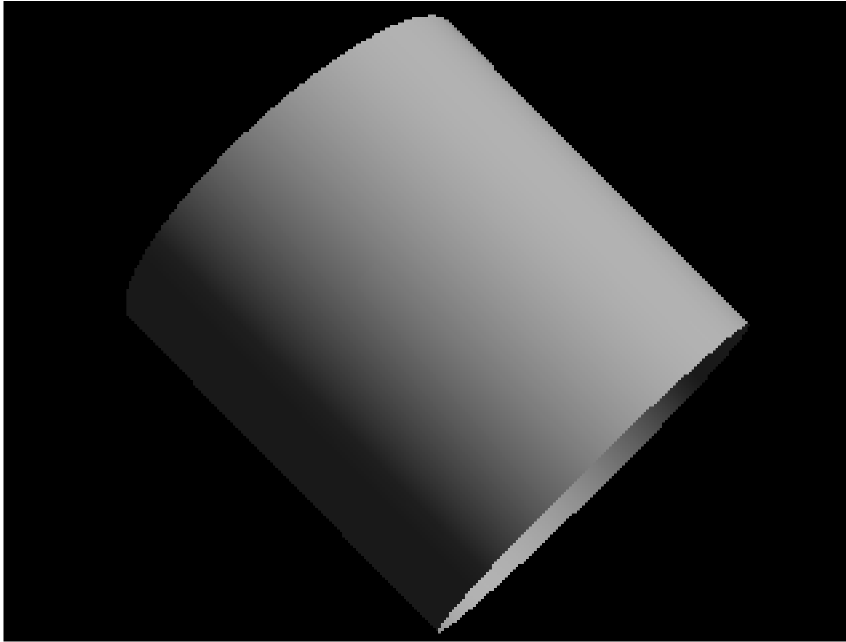


Is a finite Cylinder with a sharp point. Or it's like a Pyramid with rounded edges on sides.

Example - Creating a Cone

```
int axisSamples = 10;
int radialSamples = 10;
float radius = 2;
float height = 5;
Cone c = new Cone("Cone", axisSamples, radialSamples, radius, height);
rootNode.attachChild(c);
```

1.12.7 Cylinder



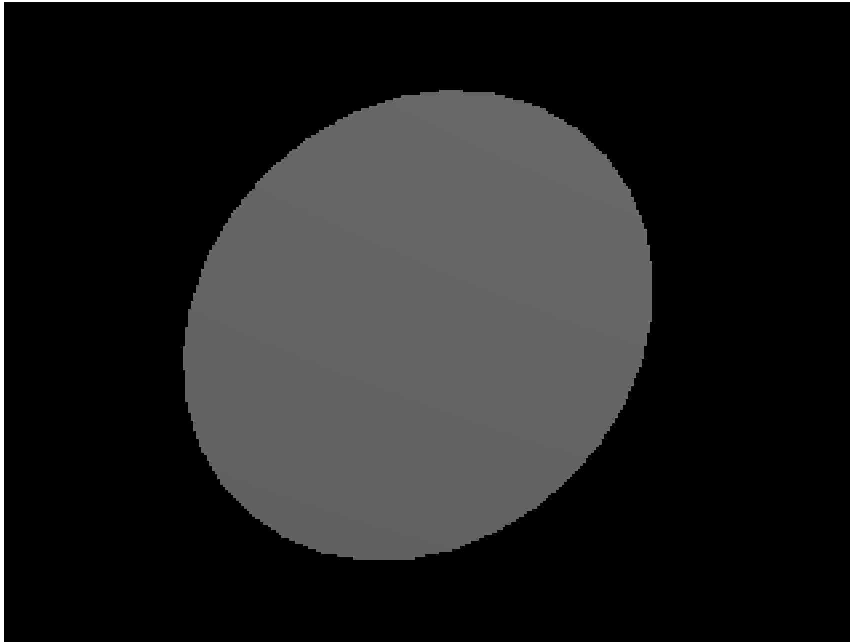
An infinite cylinder is defined as a set of points a constant distance from a line P + tD where t is a Real Number and D is unit length. A finite cylinder is a subset of the infinite cylinder where the length of t is less than a specified height. In jME Cylinder is always a finite cylinder.

Creation of a Cylinder requires supplying the height and the radius. Additionally, the number of divisions that make up the radial of the cylinder and that along its height are supplied. The higher number of divisions the more detailed the Cylinder. The Cylinder will always be oriented as lying down along the Z axis.

Example - Creating a Cylinder

```
//Create a cylinder with 10 division for both the height axis
//and the radial. The radius will be 5 making the cylinder a
//total width of 10. The height is 20.
Cylinder c = new Cylinder("Cylinder", 10, 10, 5, 20);
```

1.12.8 Disk



A Disk is the object defined by all points less than or equal to a radius about a central point, and within an Euclidean n-space (a plane). The Disk is a two dimensional object projected into three dimensional space.

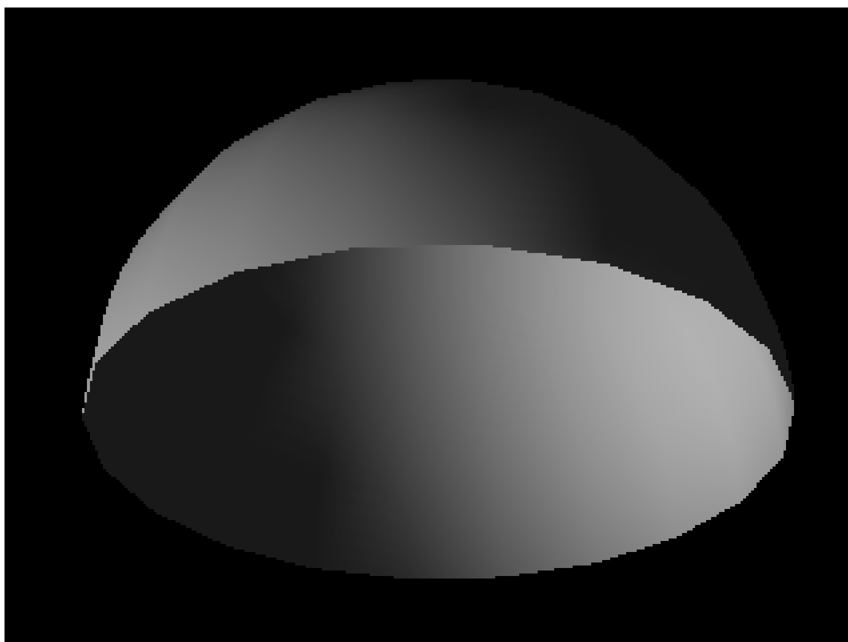
In jME a Disk is defined by a radius (the center will always be the local origin), as well as shell and radial samples. The radial samples define the number of divisions to make in the circumference (around the full circle), while the shell divisions define how many division to make coming away from the center (concentric circles). The higher number of divisions, the more detailed the Disk.

Example - Creating a Disk

```
//Create a disk with a radius of 10 and 16 concentric circles and
//32 radial divisions.
Disk d = new Disk("Disk", 16, 32, 10);
```

1.12.9 Dodecahedron

1.12.10 Dome



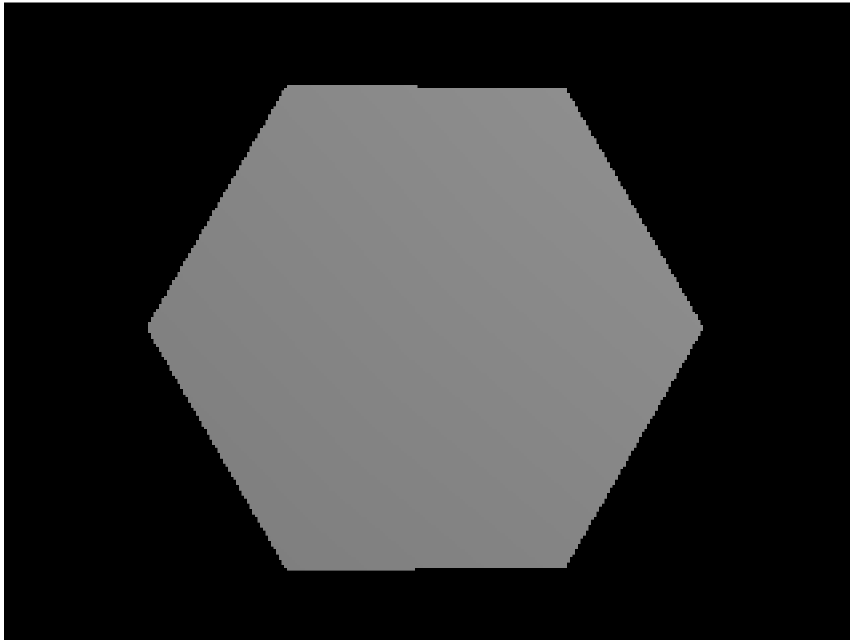
Dome (Spherical Cap) is a hemisphere Geometry object. It produces the upper (upper being the half-sphere along the positive Y Axis) half of a sphere.

Dome's parameters define its divisions along the Y axis and along the radial, as well as the radius. Much like Sphere.

Example - Creating a Dome

```
//Create a dome with 20 divisions on both the Y axis and radial.
//Create it with a radius of 20 units.
Dome dome = new Dome("My Dome", 20, 20, 20);
```

1.12.11 Hexagon



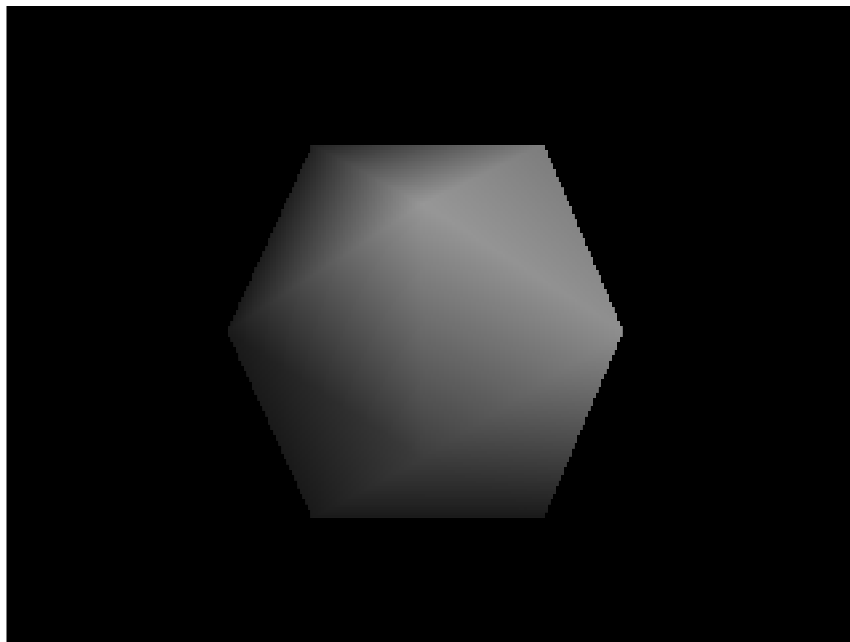
Hexagon defines a six-sided Polygon. Defined by a radius, six line segments are projected from a center each at a 60 degree angle from the lines next to it. This is a two dimensional object projected into three dimensional space.

In jME to define a Hexagon, simply supply the radius (or sideLength).

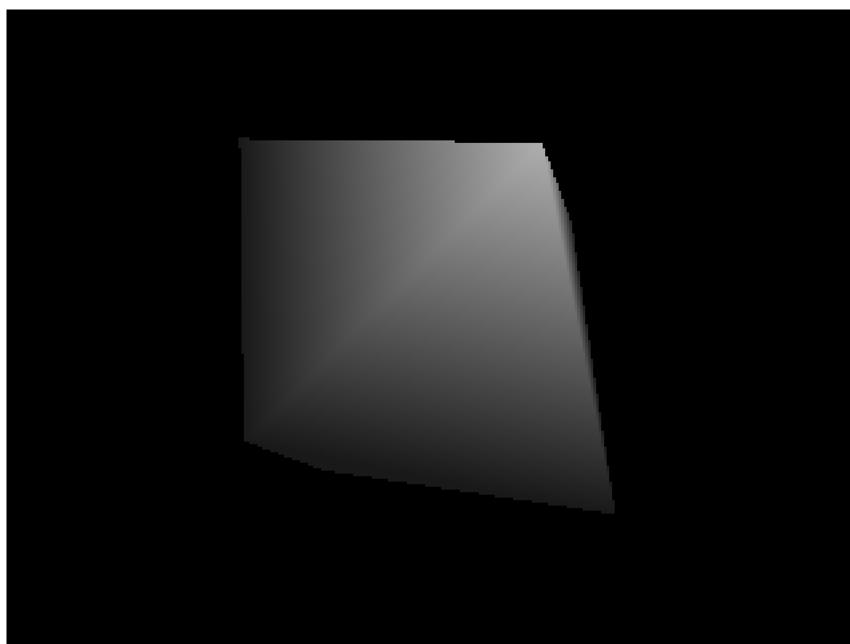
Example - Create a Hexagon

```
//We will build a Hexagon with sides of length 10
Hexagon h = new Hexagon("Hexagon", 10);
```

1.12.12 Icosahedron



1.12.13 Octahedron



An Octahedron is a polyhedron having eight faces. Each face is a triangle

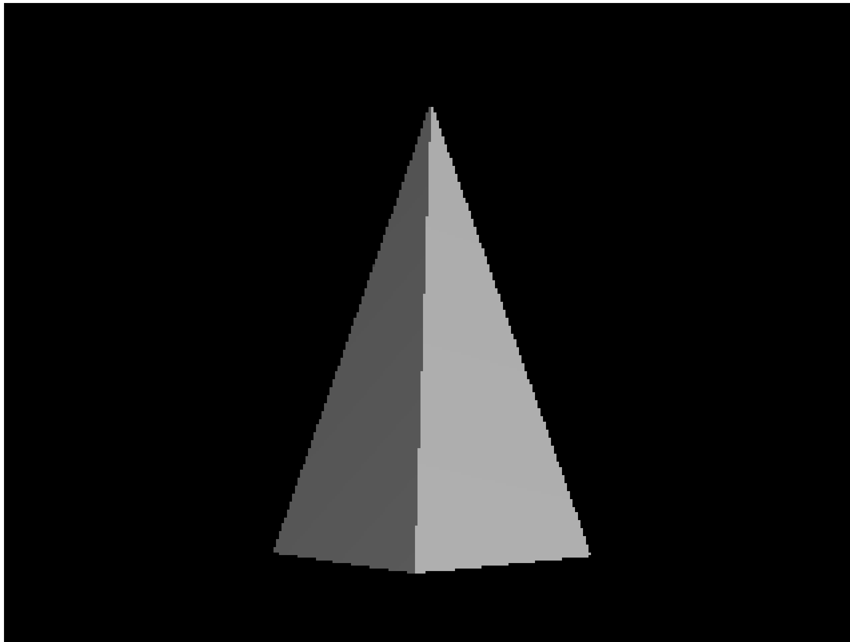
with equal sides. The appearance of the Octahedron is similar to two four sided pyramids placed together, bottom to bottom.

In jME to define an Octahedron you simply supply the side length. This length is the length of a side of one of the triangles that make up the Octahedron. Since all sides are equal, the shape can be built from this single value.

Example 1 - Building an Octahedron and Placing it in the Scene

```
Octahedron o = new Octahedron("Octahedron", 10);
rootNode.attachChild(o);
```

1.12.14 Pyramid



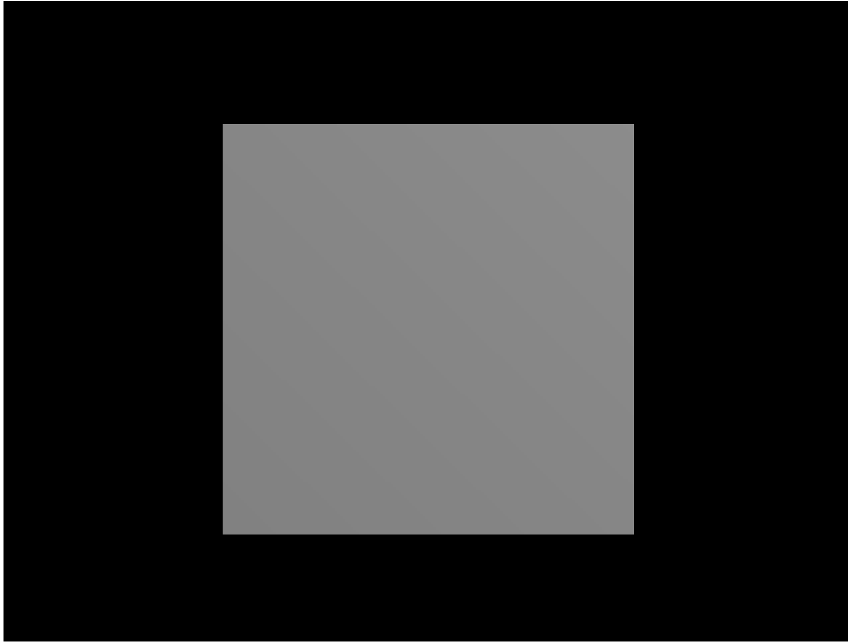
A Pyramid is a four sided object with a square base and triangle sides. All four sides are equal.

In jME to create a Pyramid simply supply the width and the height of the pyramid. The width is the width at the base, and height is from the base to the peak.

Example 1 - Create a Pyramid and Place it into the Scene

```
Pyramid p = new Pyramid("Pyramid", 10, 10);
rootNode.attachChild(p);
```

1.12.15 Quad



Quad defines a two dimensional four sided shape. By default the z width is always 0, where X corresponds to the width and Y corresponds to the height.

Example 1 - Create a Quad and Place in the Scene

```
Quad q = new Quad("quad", 800, 600);
rootNode.attachChild(q);
```

Tips Quad can be useful for UI type elements. If the Quad is placed in the ORTHO RenderQueue the width and height will be pixel sizes. Translating along the X and Y (NOT Z) will position the quad on the screen.

1.12.16 Sphere

A sphere is defined by a set of points that are equidistant from a center point. That is, every point that makes up a sphere is the same distance from the center (this distance is the radius).

In jME, the Sphere is defined by the center point, a radius, and the number of samples for the Z axis and the Radial axis. The higher the number of samples, the higher resolution (or number of triangles) the Sphere. Sphere can then be attached to the Scene Graph like any other Geometry.

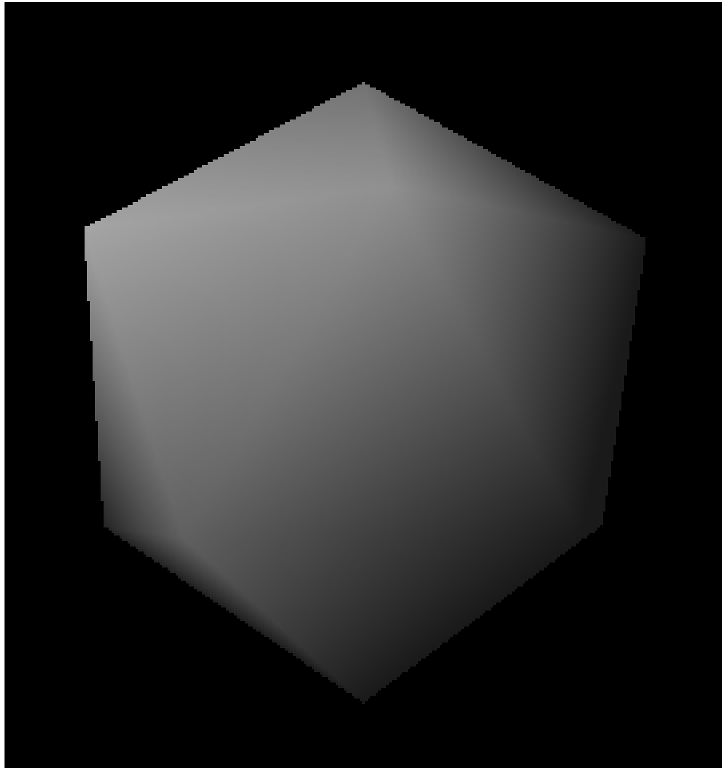
Example 1 - Creation of a Sphere and Added to the Scene

```
Sphere s = new Sphere("Sphere", 63, 50, 25);
rootNode.attachChild(s);
```

1.12.17 GeoSphere

GeoSphere is a generated polygon mesh approximating a sphere which is done by recursive subdivision. This is set by the maxlevels parameters passed into the constructor.

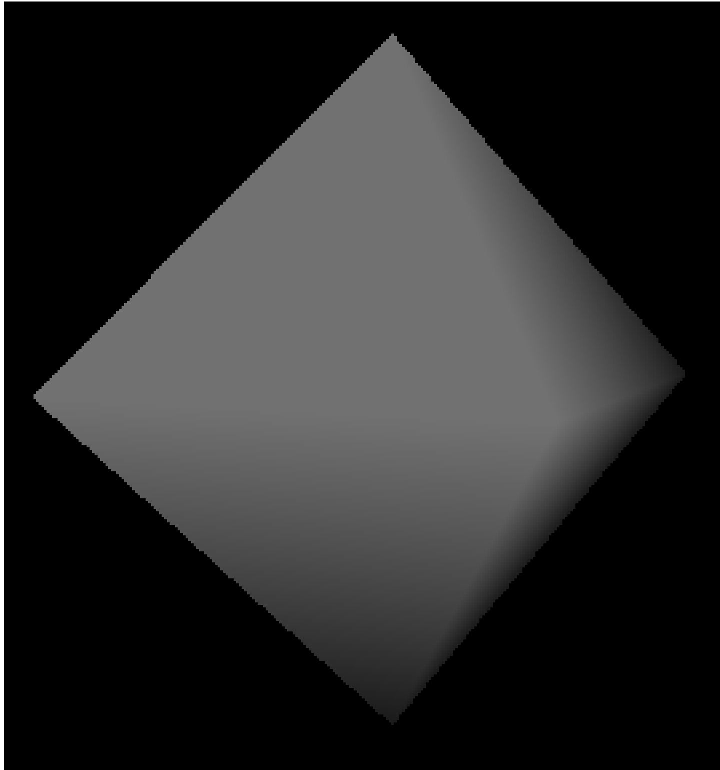
Example 1 - Creation of a Geo Sphere and Added to the Scene with ikosa set to true



Setting ikosa to true means you start with 20 triangles

```
boolean ikosa = true;
int maxLevels = 1;
GeoSphere g = new GeoSphere("GeoSphere", ikosa, maxLevels);
rootNode.attachChild(g);
```

Example 2 - Creation of a Geo Sphere and Added to the Scene with ikosa set to false



Setting ikosa to false means you start with 8 triangles

```
boolean ikosa = false;
int maxLevels = 1;
GeoSphere g = new GeoSphere("GeoSphere", ikosa, maxLevels);
rootNode.attachChild(g);
```

The higher the maxLevels the more rounder the GeoSphere will be. In the following example maxLevels has been set to 5.

1.12.18 Teapot

1.12.19 Torus

A Torus is a three dimensional surface having a genus (or hole) of one. This creates a shape much like a doughnut. It can be constructed from a rectangle by attaching both pairs of opposite edges together with no twists.

In jME, the Torus is defined by two radii and the number of samples for the Z axis and the Radial axis. The higher the number of samples, the higher resolution (or number of triangles) the Torus. The two radii that define the torus are the inner radius (the size of the hole in the doughnut), and the outer radius (from the center to the edge). Torus can then be attached to the Scene Graph like any other Geometry.

Example 1 - Creation of a Torus and Added to the Scene

```
Torus t = new Torus("Torus", 20, 20, 5, 10);
rootNode.attachChild(t);
```

1.13 Summary

This described the fundamentals of the jME system. After reading this chapter you should understand what is occuring during the game loop as well as how to create a basic system. You also saw how to add basic shapes to the scene.

Glossary

BaseGame BaseGame provides an implementation of AbstractGame, giving perhaps the simplest form of the main game loop. This loop (defined in the start) method does nothing more than execute as fast as it can. The ”start’ method initializes and then enters the loop, where the game is updated and drawn as fast as the computer is capable. All other methods are abstract, requiring subclasses to define what is to happen there. SimpleGame extends BaseGame to provide an easy launching pad for demos and prototypes., 1

Direct Render Texture versus glCopyTexImage2DDirect. Render to Texture allows you to make use of a PBuffer to draw directly to a texture buffer. This capability is available in most modern graphics cards. Previous to the PBuffer, you would have to render to the frame buffer and copy the values of the buffer to the texture buffer. This allows the PBuffer to be much faster when performing render to texture operations., 1

Frequency Frequency refers to the number of times the system’s monitor refreshes. This is measure it hertz (Hz). LCDs are typically locked at 60 Hz, CRTs are typically in the 80Hz range., 1

Fullscreen Fullscreen defines if the application is sharing space with the underlying operating system or not. Non-fullscreen (or windowed mode) places the application on the desktop as a window, sharing resources (rendering) with all other applications that happened to be shown at the same time. While, running in fullscreen allows the application to take up the entire display, and gives it a bit more freedom with tying up resources., 1

InputAction An InputAction can be subscribed at an InputHandler to get its performAction method called on specific event triggers. It also defines an interface that sets the criteria for input actions, e.g. the speed of the action. See InputSystem and InputHandler guide for usage information., 1

JOGL (Java OpenGL) are a set of bindings to OpenGL that are officially supported by Sun., 1

OpenGL From the OpenGL website: http://www.opengl.org/ OpenGL is the premier environment for developing portable, interactive 2D and 3D graphics applications. Since its introduction in 1992, OpenGL has become the industry's most widely used and supported 2D and 3D graphics application programming interface (API), bringing thousands of applications to a wide variety of computer platforms. OpenGL fosters innovation and speeds application development by incorporating a broad set of rendering, texture mapping, special effects, and other powerful visualization functions. Developers can leverage the power of OpenGL across all popular desktop and workstation platforms, ensuring wide application deployment., 1

Puppy Games Original creators of LWJGL. A Java game development house that focuses on fun minigames. `http://www.puppygames.net/`, 1

Resolution Resolution defines the number of pixels defined by a window (or entire display). For example a 640480 window would display 640 pixels on 480 lines (307,200 pixels)., 1

scene graph A scene-graph is a data structure that represents the spatial representation of a graphical scene. See: What is a Scene Graph?, 1