jMonkey Engine Programmer's Guide

Contact Us

Mojo Monkey Coding
For information regarding the jME API please refer to:

www.mojomonkeycoding.com

E-mail us at: info@mojomonkeycoding.com

Foreward

About the User Guide and jME

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, 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 group 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 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, models can be explained as well as how these can be animated. Level of detail will be explained and then Terrain. Next, spatial sorting will be explained including quad/octtrees, portals and BSPs. 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
Lead Software Engineer
Mojo Monkey Coding

Table of Contents

Chapter 1 – Getting Started

1.1– 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 now rebuilding the your project.

Each additional tool of the jME system are 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 achieved, making use of the 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.

1.2 - Abstract Game, the entry point

When creating the graphics application, you are going to need a main class which contains the main method. jME provides the AbstractGame class and it's subclasses. The AbstractGame classes provide the main game loop. This game loop is the backbone of the application and is drives the entirety of the application. The game loop called repeatedly until the application ends, where one iteration through the loop is called a “frame”. How the game loop is processed is dependent on which subclass of AbstractGame you choose. AbstractGame and all it's subclasses are abstract classes and therefore cannot be directly instantiated. The intent is the user's application extends one of the provided game types implementing the abstract methods. It is also perfectly accepted and expected that there will be time when the provided subclasses don't meet your games 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.

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.

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.

You may also set the behavior and appearance of the Properties Dialog using the setDialogBehaviour methods. The Properties Dialog is a dialog that appears (if so set), at the beginning of the 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 settings set in the setDialogBehaviour method you can have the dialog display always, display if there is not properties configuration file set, or never display. This done by passing a constant to the setDialogBehaviour method.

The constants:

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

There are five game types provided in the jME library.

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 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, any game system should be initialized here. We will go into more detail when we discuss the various system classes.

The initGame method is where all game 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.

The next game type and the one designed for quick usage and easy prototyping is SimpleGame. SimpleGame is a subclass of BaseGame and implements all the abstract methods of BaseGame. 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 wirestate (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.

All jME demos use the SimpleGame type.

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 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% of the CPU cycles. This is fine for fullscreen applications (as it will yield to important processes), but can cause issues with windowed applications.

FixedLogicrateGame allows the renderer to render as fast as possible but only giving a defined amount of time to the update method. This insures 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.

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.

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.

1.2Display System – Creating the window and renderer.

Recommended Reading:

3D Game Engine Design, David H. Eberly, Morgan Kaufmann Publishers, 2000