MTGame Programming Guide

Doug Twilleager

I – Introduction

Overview

MTGame is a Multi-Threaded Game engine for processing and rendering 3D graphics content. It is build on top of the jMonkey Engine (jME) scene graph. JME is a very feature rich scene graph technology, but it does not contain everything needed for the processing and rendering of a 3D real time simulation. It lacks a defined processing model, as well as support for taking advantage of the multi-threaded client systems that are now common.

This document describes the features, design philosophies, and the objects and interfaces for the MTGame engine. The reader of this document is expected to understand the basics of jME, but does not need to be completely versed in jME details. This guide is aimed at people who wish to develop games/simulations with this engine. This guide does not explain the architecture or algorithms of MTGame unless it is needed to explain a feature. Finally, it is intended that the reader utilize the javadoc for MTGame to follow along with this guide. This guide will discuss all the features of the system, but it will not replicate the javadoc here.

Design Philosophies

The basic premise which drove the design of MTGame was that multi-threaded client systems are now common, and therefore a game engine which took advantage of these threaded systems would be able to process much more complex worlds than are currently available. More specifically, many game engines are looking to parallel computation by function – like rendering, physics, and some artificial intelligence. While this does gain some performance, and we parallelize those functions as well, we take a slightly different approach. We parallelize mostly around objects, or entities, in the system. This allows us to place any type of processing on any object or character in a world, and allow them to process in parallel with all other entities in the system. This will allow us to scale to very large systems of parallel entities as more processors become available in a system.

We also want to make this parallel processing as transparent as possible to the developer. The system is set up such that synchronization points are automatically handled, and some are even configurable as needed by the developer. There is no need for the developer to consider manual synchronization – the system handles it all transparently.

The other main design philosophy is that of extensibility. We wanted to create a system which could be easily extended and managed by developers and designers. As we will cover more thoroughly in the Entity section, this led us to a component based design model. This allows us to add new components to the system without effecting other components of the system. It even allows us to dynamically add new features to a pre-existing object with little (and sometimes no) programmer intervention.

Features

MTGame has a number of features that extend the capabilities of jME. These include:

- A fully parallel processing model
- A fully featured rendering system with support for all major rendering techniques
- A pluggable Picking/Collision system that supports both queries and collision resolution

- A pluggable Physics system that allows for dynamic simulation of real world physics
- A pluggable input system for processing of Mouse and Keyboard events as well as other devices.

Each of these systems will be defined further throughout this document.

II – Data Structures

Entity

Many game engines use class hierarchies to define the various objects in their world. They include characters, non-player characters, vehicles, and just about every other thing in the world. Almost all of those systems start with a base object – or entity. The problem with these systems is that the functionality is baked into the class hierarchies. To add a new feature to an object, code needs to be added to the appropriate class. Then, if that same feature needs to be added to another object, the code is often duplicated, or worse, multiple inheritance collisions need to be resolved. We chose to go with a different approach.

We chose to implement our object structure using a component model approach. Our base object is the Entity object. But instead of this being an object at the base of a deep class hierarchy, our Entity is simply a container object for all components which define the characteristics of the object. There is very little state in our Entity object. It is mostly just a collection of components. Here is an example of adding a component to an entity.

```
entity = new Entity("Teapot");
renderComponent = worldManager.getRenderManager().createRenderComponent(teapot);
entity.addComponent(RenderComponent.class, renderComponent);
```

The key thing to note here is that components are added to entities by using their class as the key. This allows for interesting and useful cases of subclassing various processing features and the system being able to detect the advanced features. A component may also be removed as well as queried from an entity.

The only other feature of an entity is sub-entities. An entity may have children entities. There is no functional difference between an Entity and a sub-Entity. It is merely a convenient way for some applications to create and manage their scenes. An entity may have as many sub-entities as it wishes. There is also no state inherited from the parent entity to its sub-entities.

Note: Sub-entities do not currently work.

Managers, Systems, and Components

If an Entity is simply a collection of components, then the next question is how do we create components. The system uses a very similar scheme for all component creation, so that is described here. Specific components will be presented in later sections.

All components in the system are created and managed by Managers. So, to create a component, we must first gain access to a Manager. All Managers are managed by the WorldManager object. The WorldManager has two basic functions. First, it gives us access to all the other managers in the system. There are currently four managers in the system – RenderManager, InputManager, CollisionManager, and PhysicsManager. Each of these managers are used to create components which implement their

functionality. This allows new managers to be created and plugged into the system to provide new levels of functionality.

If we look at some example code again, we see how a manager is retrieved and creates a component.

```
WorldManager worldManager = new WorldManager("MyWorld");
RenderManager renderManager = worldManager.getRenderManager();
RenderComponent renderComponent = renderManager.createRenderComponent(teapot);
```

This creates a RenderComponent given the jME scene graph "teapot". This component can then be added to an entity.

Some Managers can delegate component creation to an underlying system. This allows for multiple systems to be active processing data as it sees appropriate. An example of this is the CollisionManager. We may have a world where some collision is being handled by one underlying system, while other collisions may be being handled in a different way. The CollisionManager therefore exposes the CollisionSystems to handle the appropriate types of data. Here is some pseudo code to illustrate this.

```
CollisionManager cm = worldManager.getCollisionManager();
// Use a jME based collision system
JMECollisionSystem jMECollision = cm.loadSystem(JMECollisionSystem.class);
CollisionComponent cc = jMECollision.createCollisionComponent(teapot1);

// Also use a JBullet based collision system
JBulletCollisionSystem jBulletCollision = cm.loadSystem(JBulletCollisionSystem.class);
CollisionComponent jbcc = jBulletCollisionSystem.createCollisionComponent(teapot2);
```

In this case, collision with teapot1 will be processed with jME, while collision with teapot2 will be processed by JBullet.

Currently, the RenderManager and InputManager manage their components, while the CollisionManager and PhysicsManager delegate to CollisionSystem's and PhysicsSystem's respectively.

The other main function of the WorldManager is to provide support for system wide functionality. These will be detailed in the WorldManager section below. That completes the basic introduction to the overall parts of the system. The rest of the document will describe the functionality of the Managers and Components in the system.

III – WorldManager

As stated above the WorldManager contains two groups of functions. First, it is the entry point for getting other Manager objects. Second, it contains functions that apply system wide. First, we will look at the Manager retrieval functions.

There are four Manager retrieval functions in the WorldManager:

```
RenderManager getRenderManager();
InputManager getInputManager();
CollisionManager getCollisionManager();
```

```
PhysicsManager getPhysicsManager();
```

There is currently only one manager per system, so the instance that is retrieved is the one for the world. In the future, if it is deemed necessary, there may be multiple managers of a type in the world. The details of each of the other Managers, and their components, will be discussed in the sections below.

Now, we can go through the various other functions of the WorldManager. The first is the most common function. The WorldManager is the object that is used to add and remove entities into and from the system. They are simply addEntity() and removeEntity(). Entities are viewed as an un-ordered collection, so the only methods currently provided are to add and remove an Entity by it's reference.

The next few functions are for updating and monitoring changes to jME nodes. JME requires that all nodes that have changes made to them be explicitly updated. Since MTGame is the engine responsible for making these updates and rendering the scene, it needs to know when a node has changed and needs to be updated. For this reason, anytime an application changes something on a jME Spatial object – which a Node is – it needs to tell MTGame. It does this by using the addToUpdateList() method. Since jME will traverse the graph from the specified Node, an application may make modifications to a whole scene graph and only pass the topmost node to MTGame via this method. This will optimize state updates.

There are many times that an application wishes to track when a node has changed. It can do that by adding a NodeChangedListener. The NodeChangedListener has a single nodeChanged() method in it, and the listener is called when any jME node has changed. To add a listener of this type to the system, you may use addNodeChangedListener() in the WorldManager. To remove the listener, use removeNodeChangedListener(). If an application is interested in tracking transform changes in a node, they may use the GeometricUpdateListener in jME.

Note: This is a new jME feature that we added.

In the Processor's section of this guide, the full description of MTGame's process model will be presented. There is one function in the WorldManager which relates to node updates and threads of execution. Most jME scene changes will happen in the MTGame processor system at the correct point in time. There are times though, where a processor is too heavyweight for some small amount of one time processing that needs to happen in the MTGame renderer thread. To facilitate this, there is the RenderUpdater callback. An application can create an object that implements the RenderUpdater interface, and then add the object to the list of objects to be called from the Renderer Thread. This is a one-time callback. It must be re-added to the list each time it wishes to be called. The method to add an object to the updater list is addRenderUpdater(). The object that is passed into this method is passed directly to the update method unchanged.

There is one last function in the WorldManager. The processor system has the ability to execute processors only when their trigger conditions have occurred. There are many trigger conditions that a processor may arm, and one of them it to listen for a user generated event. The WorldManager has the method to inject that event into the system. The event is sent into the system via the postEvent() method.

That completes all of the current functionality of the WorldManager. Next we will explore the other Managers in the system, starting with the RenderManager.

IV - RenderManager

The RenderManager handles all aspects of rendering a scene into a rendering surface. It is also the main object for interfacing with jME and the MTGame renderer thread. It contains three basic functions. First, it is used to create Canvas objects to be used for jME rendering. Second, it is used to create components and jME state objects for use in rendering. And finally, it has some parameters and listeners for monitoring the Renderer thread. First, let's look at Canvas creation.

There is currently one type of on screen rendering surface supported by the system. It is an AWT Canvas object. It is created by using the createCanvas() method as shown here.

```
Canvas canvas = renderManager.createCanvas(width, height);
```

You must use this method to create your Canvas so that the system may create the Canvas with the appropriate flags to enable it to be used for hardware 3D rendering. Also note that there are some threading issues around canvas creation and initial WorldManager creation as well. You should not create the WorldManager or the Canvas from a Swing or AWT event thread. This can cause a deadlock with startup.

The system supports having multiple Canvases for rendering. Because of this, the application must tell the system which Canvas it wishes to render into. This is done with the setCurrentCanvas() method. If an application is only going to render into one window, it may simply call setCurrentCanvas() after creating the Canvas, and then never call it again.

Note – We could add JPanel rendering support because JOGL supports it. We would have to add this to jME

Note – Official support for multi-Canvas rendering is not there

The next set of functionality in the RenderManager concerns the creation of state objects needed to render scenes. This first set is related to an artifact of jME. JME delegates the creation of all of it's RenderState objects to the underlying jME Renderer object. Note, that this is different than the MTGame Renderer. This requires applications to create these objects through a RenderManager method – so that it can get the objects from the jME Renderer. The method to use is createRendererState(). Here are a few examples of it's use:

```
ZBufferState zb = (ZBufferState) rm.createRendererState(RenderState.RS_ZBUFFER);
BlendState bs = (BlendState) rm.createRendererState(RenderState.RS_BLEND);
CullState cs = (CullState) rm.createRendererState(RenderState.RS_CULL);
```

Editorial: I know why jME chose to do this. They really wanted the underlying graphics system (lwjgl, jogl) to be able to create these. But, for a scene graph to have display specific representations of objects in the user graph is just bad. This means, for example, that it is unlikely that a scene graph saved while running with lwjgl can be loaded with jME running jogl.

The next type of objects that need to be created for the system to render are the RenderComponent objects. This is the EntityComponent in the Entity that the Renderer consumes and renders onto the canvas. The system currently supports two flavors of the RenderComponent. The first createRenderComponent() takes in a jME graph to be rendered. This one is straightforward. The

system simply renders the jME graph given. The second variant allows for an interesting feature. The second one allows for the definition of an attach point. The attach point is any jME Node. If this type of RenderComponent is encountered in an entity, it is attached to the Node specified by the attach point. This allows for a RenderComponent to be attached to an existing hierarchy, and can be changed dynamically. Imagine a character who is walking around, but want to drive a car for a while. Attach points may be used to temporarily attach the character to the car. Once the RenderComponent is created, the attach point may be modified with methods inside the RenderComponent.

The RenderComponent currently has very few methods. It allows you to get the jME graph it was created with via getSceneRoot(). And, it allows for the setting and getting of the attach point with setAttachPoint() and getAttachPoint().

It is useful to note here that all EntityComponent objects have the ability to get their Entity via the getEntity() method.

Note – setEntity is also exposed. This should be fixed.

The next interesting component that is created by the RenderManager is the CameraComponent. This component is created with createCameraComponent(). It allows the application to create all the nessesary camera parameters for jME rendering. It is also passed the jME scene graph which contains the jME CameraNode. The application positions and orients the Camera by manipulating the jME scene graph.

The CameraComponent object contains a number of parameters for controlling the viewport of the 3D camera. These are all standard 3D controls. There is one other attribute of the CameraComponent, and this is the concept of the CameraComponent being the primary camera. A 3D scene may have many camera points in the world. This allows the viewer to teleport from various cameras to get different views of the world without navigation. For any given frame of rendering, the scene is only rendered from one camera. That camera is designated the primary camera. So, while the application may have many cameras in the scene, it needs to set the one it wishes to render this frame from as the primary camera. It does this with the setPrimary() method of the CameraComponent. It can query this value with isPrimary().

Note – Dynamically changing some of these values may not work currently

ToDo – verify which methods in the CameraComponent need to be public.

Note – The following Components have not been implemented, but are on the to do list

---

There are a number of other Components that may be created with the RenderManager to add visuals to a world.

The SkyComponent allows for the specification of a Sky Box, or Sphere, or Rectangle, or .....

The AtmosphereComponent allows for the specification of atmosphere effects like fog.

The TerrainComponent allows for the specification of outdoor terrain environments.

The EffectsComponent allows for the specification of particle like effects.

The HUDComponent allows for the specification of geometry that wishes to be rendered in an orthographic projection on a given screen plane. It is useful for implementing heads up display (HUD)

like elements.

The last set of features in the RenderManager allow for the control and monitoring of the MTGame Renderer frame rate. The frame rate represents how quickly the renderer can render a scene. It is often specified in the number of frames per second. For real time graphics, an acceptable frame rate for rendering non-stereo and preserve good animation is usually between 24 and 60 frames per second. Often times, a scene can be rendered much faster than that, and it is not unusual for a scene to be rendered in the 100-200 frames per second rate. While this is impressive, it is not needed for most applications, and the processing time spent rendering those extra frames could be used for other application uses. To facilitate this MTGame allows you to specify a desired frame rate with the setDesiredFrameRate() method. If the rendering is processing faster than this, the renderer will calculate the appropriate amount of time to wait until rendering the next frame to maintain this frame rate. If the scene is rendering at a slower rate than this, it uses different mechanisms with the Processor system to try to speed up rendering. That will be covered more fully in the Processor section.

Finally, an application may monitor the actual frame rate by implementing the FrameRateListener interface and adding a frame rate listener with the setFrameRateListener() method. The system will callback the listener with the current frame rate. It calls the listener with the frequency as specified in setFrameRateListener().

V – CollisionManager

The CollisionManager provides support for collision and picking queries in the system. When a CollisionSystem is paired with a PhysicsSystem, we can even support real world dynamics and physics simulations in the engine.

The CollisionManager currently only supports one feature, and that is to load and manage CollisionSystem's. The only method in CollisonManager is load(), which takes the classname of the desired collision system.

CollisionSystem's create and manage CollisionComponent objects. If you wish for an object to be considered for collision or picking, you must create a CollisionComponent for it and add it to the entity. There are currently two CollisionSystem's supported – the JMECollisionSystem and the JBulletCollisionSystem. The JBullet system is only supported if JBullet is installed on the system.

ToDo – Remove JBulletCollisionSystem and make JBulletDynamicCollisionSystem the default

---

For both collision systems, there are two main functions – create CollisionComponent objects and provide methods for picking and collision queries. For JMECollisionSystem, the createCollisionComponent() method takes a jME scene graph. The CollisionSystem will then use this graph for collision queries.

The default CollisionComponent object has support for holding a jME graph node, so the JMECollisionComponent contains no other functionality.

The supported queries on the JMECollisionSystem include pickAll() and findCollisions(). These methods are direct copies of the picking and collision queries found in jME.

Note – More picking/collision queries to be defined. This includes retrieval of interpolated data.

ToDo – Make add/removeCollisionComponent non-public

The JBulletCollisionSystem also has the createCollisionComponent() which takes a jME graph in it's constructor. It also has a createCollisionComponent() method which takes a JBullet CollisionShape. This allows applications to have invisible collision shapes.

The only query supported for the JBulletCollisionSystem is rayTest() which maps directly to the JBullet query.

Note – More picking/collision queries to be defined. This includes retrieval of interpolated data.

ToDo – Make add/removeCollisionComponent non-public

ToDo – Hide the GeometricStateUpdate listener

The only extra feature that the JBulletCollisionComponent exposes is the ability to get the JBullet CollsionShape, if it applies to this collision component. This is done with getCollisionShape().

ToDo – Try to get rid of extra methods in JbulletCollisionComponent

VI – PhysicsManager

The PhysicsManager follows the same model as the CollisionManager. It's role is to manage a collection of PhysicsSystem's. Since physics needs to query collision and respond to those collisions, a PhysicsSystem is always paired with a CollisionSystem. Because of this, the loadSystem() method in the PhysicsManager takes not only the name of the requested PhysicsSystem, but also the corresponding CollisionSystem. Here is an example.

```
JBulletCollisionSystem collisionSystem = (JBulletCollisionSystem)
        wm.getCollisionManager().loadCollisionSystem(JBulletCollisionSystem.class);

JBulletPhysicsSystem physicsSystem = (JbulletPhysicsSystem)
        wm.getPhysicsManager().loadPhysicsSystem(JBulletPhysicsSystem.class, collisionSystem);
```

Once we have a PhysicsSystem, we can then create a PhysicsComponent. And, just as a PhysicsSystem needs to be paired with a CollisionSystem, a PhysicsComponent needs to be paired with a CollisionComponent. So, in the case of the JBulletPhysicsSystem, it's createPhysicsComponent() method requires a JBulletCollisionComponent to be passed in.

PhysicsSystem's provide a wide range of functionality. Therefore, each PhysicsSystem and PhysicsComponent's must be examined to see what functionality they provide. As an example, the JBulletPhysicsComponent provides for mass, inertia, linear velocity, and other attributes for each object. Check out the javadoc for each for the full set of features.

ToDo: Add all Jbullet Attributes

ToDo: Clean up public variables/methods

VII – InputManager

The InputManager gives applications access to Input events and having them distributed to Entities. The InputManager has only one method. That method createInputComponent(), creates an InputComponent given the set of interested events.

The InputManager registers interest in those events for that InputComponent and then delivers those events to the InputComponent. The only InputComponent currently supported is the AWTInputComponent. This component has methods to query if any events are available – via eventsPending(). You may also get the pending events with getEvents().

These objects in themselves are not all that useful, but when used in conjunction with the AWTEventProcessorComponent, which will be described in the processor section, applications can receive AWT events in their processors. By using that processor, objects can be notified and receive events, and then take action upon those events.

Note: This section will get much more content when Deron implements the Input System.

VIII – Processor Manager

MTGame Threading Model

There are a few principles that drive the threading model in MTGame. The first is that MTGame tries to maintain a constant frame rate, independent of the actual processing load. This means that the Renderer thread is decoupled from the other processing threads in the system. There are obvious threads that are decoupled from rendering – such as the physics dynamics processor. However, in this model, potentially every processor associated with any object is not synchronized with the Renderer on every frame. This is important because most real time game technologies are built with all world objects synchronized to the renderer. The MTGame model will allow us to keep rendering at a constant rate, even though there may be complex processing happening in the world. This also allows us to scale almost linearly with the number of processors in the system, because almost all of the processors are directly used by the entities in the system, without an artificial synchronization point in the renderer.

If the system is running at the desired frame rate, the system may decide to run some processors in the renderer thread. It determines which ones to run in the renderer by how far away from the camera the processor they are. The processors that run in the renderer will produce the smoothest animations, so the closest ones will be selected. The switch from running in the renderer to not is mostly transparent to the processor.

Note: Add LOD system in here

There are a few things to think about when creating your processors with this kind of model. First, the processing must be instrumented to process based on elapsed time rather than elapsed frames for clean animation. Next, this model imposes a two phase calculation scheme to the processor. We want almost all of the computation to occur in an independent thread. In our processors, this phase of the

calculation is done in the compute() method. Then, when a value is ready to be written to a visual component, it is done in the second phase of processing, which is done in our commit() method. We want as little processing as possible to happen in the commit() method because the commit processing does actually occur in the Renderer thread – due to multi-threaded safety. Finally, in traditional thread models, a processor needed to limit its processing due to the nature that this processing was being done in the renderer - this is no longer necessary. A processor may take as much time as it needs to without effecting the frame rate. The processor just needs to know that its updates may not take effect on every frame. With this backdrop, let's now look at the objects that implement this model.

ProcessorComponent

A ProcessorComponent is the main building block for application processing in MTGame. There are a number of features in this object. An application needs to subclass this object and override a few methods to enable the processor to do something useful. The ProcessorComponent has an initialize method. This is the method where the processor needs to set it's arming condition. Arming conditions and collections will be discussed shortly. The initialize() method is called when the ProcessorComponent is added to the system.

As previously mentioned, there is a compute() and commit() method in the ProcessorComponent. They are passed in the arming collection that triggered this processor. The compute() method is used to do any calculation that is needed in the processor. The commit() method is used to update jME objects in an MT-safe manner. Compute/commit methods are paired. That means that every compute method will be followed by a call to the commit method.

Note: Need to specify and implement Scheduling Bounds

Unlike regular system threads, ProcessorComponent's are not allowed to run in an unbounded fashion. Instead, we adopt a scheme where ProcessorComponent's express a desire to execute when a specific condition has occurred. We express these conditions with a set of objects starting with the ProcessorArmingCondition base class. This base class is an abstract class, with the useful arming conditions being subclasses of this. An example is the NewFrameCondition which expresses an interest in being executed each time a frame is rendered. Here is an example of setting this condition in a ProcessorComponent's initialize method.

```
public void initialize() {
   setArmingCondition(new NewFrameCondition(this));
}
```

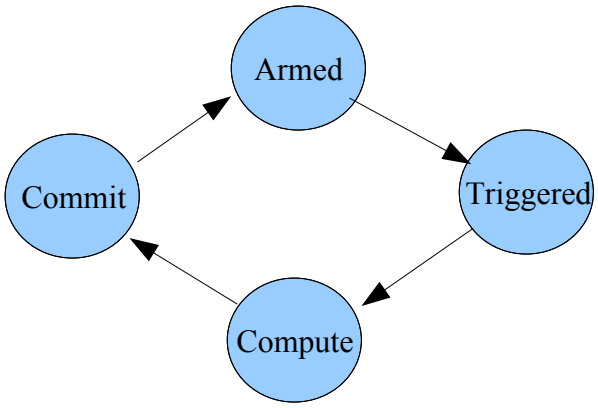
In some cases, a ProcessorComponent may wish to express interest in multiple conditions simultaneously. To do this, you may use the ProcessorArmingCollection. It is a subclass of the ProcessorArmingCondition, so it may used as an arming condition on any ProcessorComponent. The ProcessorArmingCollection allows for a collection of arming conditions. If any of these conditions trigger, then the collections is considered to be triggered.

Note: Should we add a flag to indicate “or” or “and” based triggers

Here is an example of instantiating a ProcessorArmingCollection.

```
public void initialize() {
        collection = new ProcessorArmingCollection(this);
        collection.addCondition(new AwtEventCondition(this));
        collection.addCondition(new NewFrameCondition(this));
        setArmingCondition(collection);
    }
```

Once an arming condition has been set on a ProcessorComponent, it stays in effect until it is explicitly changed by the application. With this system, a ProcessorComponent can be in a number of different states as it executes. It starts in an armed state, where it is ready to execute. Once it's arming condition triggers, it's compute and commit methods are executed. It is then re-armed, waiting for it's arming condition to trigger again. Here is a simple state diagram which shows this.



Now let's take a brief look at the currently available arming conditions in the system. The NewFrameCondition was discussed above. There is also an AWTEventCondition which expresses interest in AWT events. There is a TimerExpiredCondition which triggers when it's specified time has elapsed. Finally, there is a PostEventCondition which expresses interest in a set of user generated events. This triggers when one of the expressed events is sent into the system with the WorldManager.postEvent() method.

Note: Should we add a number of frames argument to NewFrameCondition

Note: The TimerExpiredCondition does not currently work

ToDo: We will be adding a RegionEntryExitCondition

The ProcessorComponent has a couple more features to be discussed. The first is called chaining. There are some cases where one processor is dependent upon the computations of another processor. An example of this is a weapon in a game following the camera processor. If it is not synchronized correctly, then you will see a jittering effect in the scene. To accommodate this we have the notion of chaining. A ProcessorComponent may add another ProcessorComponent to its chain. When a ProcessorComponent is triggered, it is executed, and then each ProcessorComponent in its chain is executed in the order of the chain. The chained ProcessorComponent's in the chain do not have to have their conditions triggered in order to execute. Since their computation must be updated based upon the

previous ProcessorComponent, they are executed even though their condition may not have triggered. The methods addToChain() and removeFromChain() facilitate this process.

There is one last feature in the ProcessorComponent. Even though we have a system where almost all of the ProcessorComponent's are expected to run asynchronously from the Renderer, there are some cases where a ProcessorComponent needs to run in the Renderer all the time. An example of this is the Camera's in the system. They need to be able to update every frame – no matter the current frame rate. To facilitate this, a ProcessorComponent may set a flag to always be run in the renderer. This, combined with the NewFrameCondition can guarantee that a ProcessorComponent can run in the Renderer every frame. The method to set this flag is setRunInRenderer().

There is one last subject to cover in relation to ProcessorComponent's. It is sometimes useful to have more than one ProcessorComponent associated with an Entity. An example would be a room with a ceiling fan and a door animation. Both are separate ProcessorComponent's and both are part of the room Entity. The system allows this through the use of the ProcessorCollectionComponent. It simple holds a collection of ProcessorComponent's. By creating one of these, and adding it to an Entity, you can have multiple ProcessorComponent's in a single Entity.

ToDo: removeProcessor is missing.

IX – Putting it All Together

So, now that we've explored the components of the MTGame system, let's put it all together in a simple example – a spinning teapot. This will focus on the MTGame aspects of the program. Fully running examples can be found in the distribution.

There are a few things we need to do during the initialization of the application. Here are those items.

```
WorldManager worldManager = new WorldManager("Hello World");
worldManager.getRenderManager().setDesiredFrameRate(desiredFrameRate);
canvas = worldManager.getRenderManager().createCanvas(width, height);
worldManager.getRenderManager().setCurrentCanvas(canvas);
worldManager.getRenderManager().setFrameRateListener(this, 100);
```

This gives us our WorldManager and our Canvas for rendering. We are also setting our desired frame rate – the default is 60 frames per second – and sets up a frame rate listener.

Next up, we need to set up our camera.

```
Node cameraSG = new Node("MyCamera SG");
cameraNode = new CameraNode("MyCamera", null);
cameraSG.attachChild(cameraNode);

Entity camera = new Entity("DefaultCamera");
CameraComponent cc = worldManager.getRenderManager().createCameraComponent(cameraSG, cameraNode,
      width, height, 45.0f, aspect, 1.0f, 1000.0f, true);
camera.addComponent(CameraComponent.class, cc);

// Create the input listener and process for the camera
int eventMask = InputManager.KEY_EVENTS | InputManager.MOUSE_EVENTS;
AWTInputComponent cameraListener =
```

```
(AWTInputComponent)worldManager.getInputManager().createInputComponent(eventMask);
    OrbitCameraProcessor eventProcessor = new OrbitCameraProcessor(cameraListener, cameraNode, wm, camera);
    eventProcessor.setRunInRenderer(true);

    ProcessorCollectionComponent pcc = new ProcessorCollectionComponent();
    pcc.addProcessor(eventProcessor);
    camera.addComponent(ProcessorCollectionComponent.class, pcc);

    worldManager.addEntity(camera);
```

There are a number of things happening here. First, we need the jME camera scene graph. Then we use that to create the CameraComponent for the camera Entity. Next, we create a processor, called the OrbitCameraProcessor, which controls the position and orientation of the camera. We add that processor into a processor collection, and then add it to the entity. Finally, we add the entity into the WorldManager. Notice, than since this is a camera processor, we set its run in renderer flag to true.

Finally, we need to create our spinning teapot.

```
    Node node = new Node();
    Teapot teapot = new Teapot();
    teapot.resetData();
    node.attachChild(teapot);

    ZBufferState buf = (ZBufferState)
worldManager.getRenderManager().createRendererState(RenderState.RS_ZBUFFER);
    buf.setEnabled(true);
    buf.setFunction(ZBufferState.TestFunction.LessThanOrEqualTo);

    node.setRenderState(buf);
    node.setLocalTranslation(x, y, z);

    Entity e = new Entity("Teapot");
    RenderComponent sc = worldManager.getRenderManager().createRenderComponent(teapot);
    e.addComponent(RenderComponent.class, sc);

    RotationProcessor rp = new RotationProcessor("Teapot Rotator", worldManager,
        teapot, (float) (6.0f * Math.PI / 180.0f));
    e.addComponent(ProcessorComponent.class, rp);
    wm.addEntity(e);
```

This uses the Teapot object from jME. It also uses a built-in processor to do the rotation – which is the RotationProcessor.