

jMonkeyEngine

3

Documentation

jMonkeyEngine

Published
with GitBook

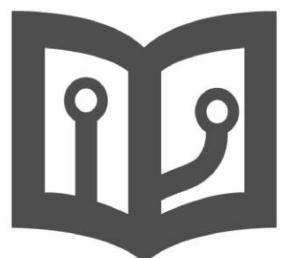


Table of Contents

Introduction	0
Beginner	1
Hello SimpleApplication	1.1
Hello Node	1.2
Hello Asset	1.3
Hello Loop	1.4
Hello Input	1.5
Hello Material	1.6
Hello Animation	1.7
Hello Picking	1.8
Hello Collision	1.9
Hello Terrain	1.10
Hello Audio	1.11
Hello Effects	1.12
Hello Physics	1.13
Hello Vector	1.14
Hello ChaseCam	1.15
What is an IDE	1.16
Intermediate	2
File Types	2.1
App Settings	2.2
SimpleApplication	2.3
Best Practices	2.4
Engine Overview	2.5
How to use materials	2.6
Math	2.7
Monkeyblaster	2.8
Optimization	2.9
Rolling Madness	2.10
Transparency Sorting	2.11

Media Asset Pipeline	2.12
Advanced	3
3D Models	3.1
AI	3.2
Android	3.3
Animation	3.4
Anisotropic Filtering	3.5
Application States	3.6
AppStates Demo	3.7
Asset Manager	3.8
atom_framework	3.9
Audio Environment Presets	3.10
Audio	3.11
Bloom and Glow	3.12
Building Recast	3.13
bullet_multithreading.md	3.14
Bullet Pitfalls	3.15
Camera	3.16
Capture Audio Video to a file	3.17
Cinematics	3.18
Collision and Intersection	3.19
Combo Moves	3.20
Custom Controls	3.21
Custom Meshes	3.22
Debugging	3.23
Effects Overview	3.24
Endless Terraingrid	3.25
Fade	3.26
Headless Server	3.27
Hinges and Joints	3.28
HUD	3.29
Input Handling	3.30
j3m Material Files	3.31
jME3 Renderbuckets	3.32

jME3 ShaderNodes	3.33
jME3 Shaders	3.34
jME3 SRGB Pipeline	3.35
Level of Detail	3.36
Light and Shadow	3.37
Loading Screen	3.38
Localization	3.39
logging.md	3.40
Makehuman Blender OgreXML Toolchain	3.41
Making the camera follow a character	3.42
Material Definitions	3.43
Material Specification	3.44
Materials Overview	3.45
Mesh	3.46
Monkey Brains	3.47
MonkeyZone	3.48
MotionPath	3.49
Mouse Picking	3.50
Multiple Camera Views	3.51
Multithreading	3.52
Networking video tutorials	3.53
Networking	3.54
Migration	3.55
Compression	3.56
Connection	3.57
Sending and Receiving messages	3.58
Serializing	3.59
Services	3.60
Streaming	3.61
Nifty GUI	3.62
NiftyGUI Best practices	3.63
NiftyGUI Editor	3.64
NiftyGUI Java Interaction	3.65

NiftyGUI Java Layout	3.66
NiftyGUI Overlay	3.67
NiftyGUI Popup Menu	3.68
NiftyGUI Projection	3.69
Nifty GUI Scenarios	3.70
Nifty GUI XML Layout	3.71
Nifty GUI v1.3 Notes	3.72
Ogre Compatibility	3.73
Open Game Finder	3.74
Particle Emitters	3.75
Physics Listeners	3.76
Physics	3.77
Ragdoll	3.78
Read graphic card capabilities	3.79
Recast	3.80
Save and Load	3.81
Screenshots	3.82
Shape	3.83
Sky	3.84
Spatial	3.85
Statsview	3.86
Steer Behaviours	3.87
Swing Canvas	3.88
Terrain Collision	3.89
Terrain	3.90
texture_atlas.md	3.91
Traverse Scenegraph	3.92
Update Loop	3.93
Vehicles	3.94
Video	3.95
Walking Character	3.96
Water	3.97
Post - Processor Water	3.98
Remote - Controlling the camera	3.99

jME3	4
Android	4.1
Ouya	4.2
Atomixtuts - WIP	4.3
Blade Game	4.4
Blade Game - Design	4.5
Cards Game	4.6
Cards Game - Gameplay	4.7
Cards Game - Ad Techs	4.8
Cards Game - AI	4.9
blendswaparcade.md	4.10
Build from sources	4.11
Build jME3 Sources with NetBeans	4.12
Contributions	4.13
Eclipse jME3 Android JNINDK	4.14
Features	4.15
Hardware Compatibility List	4.16
iOS	4.17
jME3 Source Structure	4.18
Materials	4.19
Math for Dummies	4.20
Math video tutorials	4.21
Math	4.22
Matrix	4.23
Maven	4.24
Quaternion	4.25
Requirements	4.26
Rise of Mutants Project	4.27
Rotate	4.28
Scenegraph for Dummoes	4.29
Scripting	4.30
Groovy Basic Scripts	4.31
Groovy Event	4.32

Groovy Learn	4.33
Snippets	4.34
Groovy AI - RENAMED	4.35
Setting up jME3 in Eclipse	4.36
Setting up NetBeans and jME3	4.37
Shader Video Tutorials	4.38
Shaderblow Project	4.39
SimpleApplication from the command line	4.40
Terminology	4.41
The Scenegraph	4.42
Update Geometric State	4.43
user_examples_project	4.44
Users Guide	4.45
Virtual Reality	4.46
webstart.md	4.47
SDK	5
3ds to Blender to jMP	5.1
Application Deployment	5.2
Asset Packs	5.3
Blender	5.4
Build Platform	5.5
Code Editor	5.6
Comic	5.7
Debugging Profiling Testing	5.8
Default Build Script	5.9
Development Intro	5.10
Development	5.11
Extension Library	5.12
Filters	5.13
Font Creation	5.14
General	5.15
Increasing Heap Memory	5.16
Log Files	5.17
Material Editing	5.18

Model Loader and Viewer	5.19
Model Loader	5.20
NeoTexture	5.21
Platform Development	5.22
Project Creation	5.23
Projects Assets	5.24
Sample Code	5.25
Scene Composer	5.26
Scene Explorer	5.27
Scene	5.28
Sceneexplorer	5.29
SDK Intro	5.30
Setup	5.31
Terrain Editor	5.32
Troubleshooting	5.33
Update Center	5.34
Use own jME	5.35
Vehicle Creator	5.36
Version Control	5.37
Why not Eclipse?	5.38
Unofficial - Contributions	6
External: 3ds Max	6.1
External: Blender	6.2
External: Blender - Example	6.3
External: Fonts	6.4
External: MakeHuman	6.5
Tools: Charts	6.6
Tools: Navigation	6.7
Third Person Camera	6.8
Atom MOVED	6.9
Atom - AI	6.10
Atom - Atom2D	6.11
Atom - WIP	6.12

Cubes	6.13
Cubes - Basic Example	6.14
Cubes - Build your block world	6.15
Cubes - Register your blocks	6.16
Cubes - Settings	6.17
Entity System	6.18
ES - Advanced	6.19
ES - Beginner	6.20
ES - Detailed	6.21
ES - Entityset	6.22
ES - Interviews	6.23
ES - Introduction	6.24
ES - Points	6.25
ES - Terms	6.26
ES - Usage	6.27
ES - Example: Damage System	6.28
ES - Example: Own Logic Thread	6.29
Particles	6.30
Particles - Reference	6.31
TonegodGUI - WIP	6.32
TonegodGUI Quickstart	6.33
TonegodGUI Scrollarea	6.34
TonegodGUI Alert Box	6.35
TonegodGUI Button	6.36
Vegetation System	6.37
VS - Grass	6.38
VS - Trees	6.39
Miscellaneous	7
Solutions	7.1
Compare jME2-jME3	7.2
Submission	7.3
Plugin: Shaderblow	7.4
Welcome: v3.1	7.5
GSOC	7.6

GSOC - Application	7.7
GSOC - Ideas	7.8
GSOC - Students Handbook	7.9
FAQ	8
Report Bugs	9
Logo	10
Sample Chapter	11
 Sample page	11.1
 second-sample-page	11.2

jMonkeyEngine Documentation

This documentation wiki contains installation and configuration guides, jME coding tutorials and other information that will help you get your game project going. You can search the contents of this wiki using the search box in the upper right.

You are also very welcome to fix mistakes or spelling as well as unclear paragraphs using the “Wiki” menu on top or the “Edit” buttons after each paragraph. You have to be logged in to edit the wiki.

</div>

Install

Before installing, check the [license](#), [features](#), and [requirements](#). Then choose one of these options:

	Recommended	Optional	Optional
You want to...	Get started with jMonkeyEngine	Use jMonkeyEngine in another IDE	Build custom engine from sources
Then download...	jMonkeyEngine SDK	Binaries	Sources (Subversion)
You receive...	Sources, binaries, javadoc, SDK	Latest stable binary build, sources, javadoc.	Sources
Learn more here...	Using the SDK Project Creation 	Setting up jME3 with maven compatible IDEs * Setting up JME3 in the NetBeans IDE * Setting up JME3 in the Eclipse IDE * Setting up JME3 in the Eclipse IDE (with Android and/or JNI/NDK) *	Building JME3 from the Sources Building JME3 from the sources with NetBeans Setting up JME3 on the commandline

(*) The SDK creates Ant-based projects that any Java IDE can import. We recommend users of other IDEs to also download the jMonkeyEngine SDK and choose “File→Import Project→External Project Assets” to create a codeless project for managing assets only. This way you can code in the IDE of your choice, and use the SDK to convert your models to .j3o format.

Create

After downloading and installing, [bookmark the jME Documentation page](#) and start writing your first game!

Tutorials	jMonkeyEngine SDK	Other Documentation
jME3 beginner tutorials	jMonkeyEngine SDK Documentation and Video Tutorials	Full API JavaDoc
jME3 intermediate articles	jMonkeyEngine SDK - the Comic :-)	Blender Modeling Guide
jME3 advanced documentation		Answers to Frequently Asked Questions

Contribute

Are you an experienced Java developer who wants to add new features or contribute patches to the jME3 project?

- Get inspired by existing contributions
- [Read the Contributors Handbook](#)
- Chime in on the [Contributors Forum](#)
- Learn about the source structure
- Write jMonkeyEngine SDK plugins and visual editors
- Report bugs and submit patches

Contact

You are welcome to contribute and inquire about the project: Please [contact the developers](#) or ask on the [forums](#).

- [Contact the jME team](#)
 - [Core team - Who are we?](#)
- [Report a bug](#)
- [Report unclear or missing documentation](#)

Languages

[Документация на Русском](#)

[Documentação em Português](#)

[中文版](#)

[documentation](#), [sdk](#), [install](#)

</div>

Beginner

And so begins the beginner chapter. Done.

title: jMonkeyEngine 3 Tutorial (1) - Hello SimpleApplication

jMonkeyEngine 3 Tutorial (1) - Hello SimpleApplication

Previous: [Installing JME3](#), Next: [Hello Node](#)

Prerequisites: This tutorial assumes that you have [downloaded the jMonkeyEngine SDK](#).

In this tutorial series, we assume that you use the jMonkeyEngine [SDK](#). As an intermediate or advanced Java developer, you will quickly see that, in general, you can develop jMonkeyEngine code in any integrated development environment (NetBeans IDE, Eclipse, IntelliJ) or even from the [command line](#).

OK, let's get ready to create our first jMonkeyEngine3 application.

Create a project

In the jMonkeyEngine SDK:

1. Choose File→New Project... from the main menu.
2. In the New Project wizard, select the template JME3→Basic Game. Click Next.
 - i. Specify a project name, e.g. "HelloWorldTutorial"
 - ii. Specify a path where to store your new project, e.g. a `jMonkeyProjects` directory in your home directory.
3. Click Finish.

If you have questions, read more about [Project Creation](#) here.

We recommend to go through the steps yourself, as described in the tutorials. Alternatively, you can create a project based on the [JmeTests](#) template in the jMonkeyEngine SDK. It will create a project that already contains the `jme3test.helloworld` samples (and many others). For example, you can use the JmeTests project to verify whether you got the solution right.
 </div>

Extend SimpleApplication

For this tutorial, you want to create a jme3test.helloworld package in your project, and create a file HelloJME3.java in it.

In the jMonkeyEngine SDK:

1. Right-click the Source Packages node of your project.
2. Choose New...→Java Class to create a new file.
3. Enter the class name: HelloJME3
4. Enter the package name: jme3test.helloworld.
5. Click Finish.

The SDK creates the file HelloJME3.java for you.

Code Sample

Replace the contents of the HelloJME3.java file with the following code.

```

package jme3test.helloworld;

import com.jme3.app.SimpleApplication;
import com.jme3.material.Material;
import com.jme3.math.Vector3f;
import com.jme3.scene.Geometry;
import com.jme3.scene.shape.Box;
import com.jme3.math.ColorRGBA;

/** Sample 1 - how to get started with the most simple JME 3 applic
 * Display a blue 3D cube and view from all sides by
 * moving the mouse and pressing the WASD keys. */
public class HelloJME3 extends SimpleApplication {

    public static void main(String[] args){
        HelloJME3 app = new HelloJME3();
        app.start(); // start the game
    }

    @Override
    public void simpleInitApp() {
        Box b = new Box(1, 1, 1); // create cube shape
        Geometry geom = new Geometry("Box", b); // create cube geo
        Material mat = new Material(assetManager,
            "Common/MatDefs/Misc/Unshaded.j3md"); // create a simple
        mat.setColor("Color", ColorRGBA.Blue); // set color of ma
        geom.setMaterial(mat); // set the cube's
        rootNode.attachChild(geom); // make the cube a
    }
}

```

Right-click the HelloJME3 class and choose Run. If a jME3 settings dialog pops up, confirm the default settings.

1. You should see a simple window displaying a 3D cube.
2. Press the WASD keys and move the mouse to navigate around.
3. Look at the FPS text and object count information in the bottom left. You will use this information during development, and you will remove it for the release. (To read the numbers correctly, consider that the 14 lines of text counts as 14 objects with 914

- vertices.)
4. Press Escape to close the application.

Congratulations! Now let's find out how it works!

Understanding the Code

The code above has initialized the scene, and started the application.

Start the SimpleApplication

Look at the first line. Your HelloJME3.java class extends `com.jme3.app.SimpleApplication`.

```
public class HelloJME3 extends SimpleApplication {  
    // your code...  
}
```

Every JME3 game is an instance of the `com.jme3.app.SimpleApplication` class. The SimpleApplication class is the simplest example of an application: It manages a 3D scene graph, checks for user input, updates the game state, and automatically draws the scene to the screen. These are the core features of a game engine. You extend this simple application and customize it to create your game.

You start every JME3 game from the `main()` method, as every standard Java application:

1. Instantiate your `SimpleApplication`-based class
2. Call the application's `start()` method to start the game engine.

```
public static void main(String[] args){  
    HelloJME3 app = new HelloJME3(); // instantiate the game  
    app.start(); // start the game!  
}
```

The `app.start();` line opens the application window. Let's learn how you put something into this window (the scene) next.

Understanding the Terminology

What you want to do	How you say that in JME3 terminology
You want to create a cube.	I create a Geometry with a 1x1x1 Box shape.
You want to use a blue color.	I create a Material with a blue Color property.
You want to colorize the cube blue.	I set the Material of the Box Geometry.
You want to add the cube to the scene.	I attach the Box Geometry to the rootNode.
You want the cube to appear in the center.	I create the Box at the origin = at Vector3f.ZERO .

If you are unfamiliar with the vocabulary, read more about [the Scene Graph](#) here.

Initialize the Scene

Look at rest of the code sample. The `simpleInitApp()` method is automatically called once at the beginning when the application starts. Every JME3 game must have this method. In the `simpleInitApp()` method, you load game objects before the game starts.

```
public void simpleInitApp() {
    // your initialization code...
}
```

The initialization code of a blue cube looks as follows:

```
public void simpleInitApp() {
    Box b = new Box(1, 1, 1); // create a 1x1x1 box shape
    Geometry geom = new Geometry("Box", b); // create a cube geometry
    Material mat = new Material(assetManager,
        "Common/MatDefs/Misc/Unshaded.j3md"); // create a simple material
    mat.setColor("Color", ColorRGBA.Blue); // set color of material
    geom.setMaterial(mat); // set the cube geometry
    rootNode.attachChild(geom); // make the cube appear
}
```

A typical JME3 game has the following initialization process:

1. You initialize game objects:
 - You create or load objects and position them.
 - You make objects appear in the scene by attaching them to the `rootNode`.
 - **Examples:** Load player, terrain, sky, enemies, obstacles, ..., and place them in

their start positions.

2. You initialize variables

- You create variables to track the game state.
- You set variables to their start values.
- **Examples:** Set the `score` to 0, set `health` to 100%, ...

3. You initialize keys and mouse actions.

- The following input bindings are pre-configured:
 - W,A,S,D keys – Move around in the scene
 - Mouse movement and arrow keys – Turn the camera
 - Escape key – Quit the game
- Define your own additional keys and mouse click actions.
- **Examples:** Click to shoot, press Space to jump, ...

Conclusion

You have learned that a `SimpleApplication` is a good starting point because it provides you with:

- A `simpleInitApp()` method where you create objects.
- A `rootNode` where you attach objects to make them appear in the scene.
- Useful default input settings that you can use for navigation in the scene.

When developing a game application, you want to:

1. Initialize the game scene
2. Trigger game actions
3. Respond to user input.

The now following tutorials teach how you accomplish these tasks with the jMonkeyEngine 3.

Continue with the [Hello Node](#) tutorial, where you learn more details about how to initialize the game world, also known as the scene graph.

See also:

- [Install the jMonkeyEngine](#)
- [SimpleApplication From the Commandline](#)
- [Create a JME3 project.](#)

[beginner](#), [intro](#), [documentation](#), [init](#), [simpleapplication](#), [basegame](#)

</div>

title: jMonkeyEngine 3 Tutorial (2) - Hello Node

jMonkeyEngine 3 Tutorial (2) - Hello Node

Previous: [Hello SimpleApplication](#), Next: [Hello Assets](#).

In this tutorial we will have a look at the creation of a 3D scene.

- This tutorial assumes that you know what [the Scene Graph](#) is.
- For a visual introduction, check out [Scene Graph for Dummies](#).

When creating a 3D game

1. You create some scene objects like players, buildings, etc.
2. You add the objects to the scene.
3. You move, resize, rotate, color, and animate them.

You will learn that the scene graph represents the 3D world, and why the rootNode is important. You will learn how to create simple objects, how to let them carry custom data (such as health points), and how to “transform” them by moving, scaling, and rotating. You will understand the difference between the two types of “Spatials” in the scene graph: Nodes and Geometries.

Code Sample

```
package jme3test.helloworld;

import com.jme3.app.SimpleApplication;
import com.jme3.material.Material;
import com.jme3.math.ColorRGBA;
import com.jme3.math.Vector3f;
import com.jme3.scene.Geometry;
import com.jme3.scene.Node;
import com.jme3.scene.shape.Box;

/** Sample 2 - How to use nodes as handles to manipulate objects in
 * You can rotate, translate, and scale objects by manipulating the
```

```
* The Root Node is special: Only what is attached to the Root Node
public class HelloNode extends SimpleApplication {

    public static void main(String[] args){
        HelloNode app = new HelloNode();
        app.start();
    }

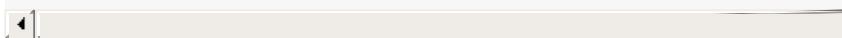
    @Override
    public void simpleInitApp() {

        /** create a blue box at coordinates (1,-1,1) */
        Box box1 = new Box(1,1,1);
        Geometry blue = new Geometry("Box", box1);
        blue.setLocalTranslation(new Vector3f(1,-1,1));
        Material mat1 = new Material(assetManager,
            "Common/MatDefs/Misc/Unshaded.j3md");
        mat1.setColor("Color", ColorRGBA.Blue);
        blue.setMaterial(mat1);

        /** create a red box straight above the blue one at (1,3,1)
        Box box2 = new Box(1,1,1);
        Geometry red = new Geometry("Box", box2);
        red.setLocalTranslation(new Vector3f(1,3,1));
        Material mat2 = new Material(assetManager,
            "Common/MatDefs/Misc/Unshaded.j3md");
        mat2.setColor("Color", ColorRGBA.Red);
        red.setMaterial(mat2);

        /** Create a pivot node at (0,0,0) and attach it to the root
        Node pivot = new Node("pivot");
        rootNode.attachChild(pivot); // put this node in the scene

        /** Attach the two boxes to the *pivot* node. (And transition)
        pivot.attachChild(blue);
        pivot.attachChild(red);
        /** Rotate the pivot node: Note that both boxes have rotated
        pivot.rotate(.4f,.4f,0f);
    }
}
```





Build and run the code sample. You should see two colored boxes tilted at the same angle.

Understanding the Terminology

In this tutorial, you learn some new terms:

What you want to do	How you say it in JME3 terminology
Lay out the 3D scene	Populate the scene graph
Create scene objects	Create Spatial (e.g. create Geometries)
Make an object appear in the scene	Attach a Spatial to the rootNode
Make an object disappear from the scene	Detach the Spatial from the rootNode
Position/move, turn, or resize an object	Translate, or rotate, or scale an object = transform an object.

Every JME3 application has a `rootNode`: Your game automatically inherits the `rootNode` object from `SimpleApplication`. Everything attached to the `rootNode` is part of the scene graph. The elements of the scene graph are **Spatial**s.

- A Spatial contains the location, rotation, and scale of an object.
- A Spatial can be loaded, transformed, and saved.
- There are two types of Spatial: Nodes and Geometries.

	Geometry	Node
Visibility:	A Geometry is a visible scene object.	A Node is an invisible “handle” for scene objects.
Purpose:	A Geometry stores an object's looks.	A Node groups Geometries and other Nodes together.
Examples:	A box, a sphere, a player, a building, a piece of terrain, a vehicle, missiles, NPCs, etc...	The <code>rootNode</code> , a floor node grouping several terrains, a custom vehicle-with-passengers node, a player-with-weapon node, an audio node, etc...

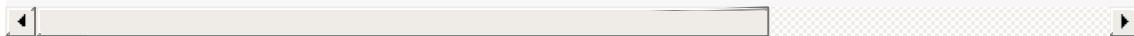
Understanding the Code

What happens in the code snippet? You use the `simpleInitApp()` method that was introduced in the first tutorial to initialize the scene.

1. You create the first box Geometry.

- Create a Box shape with extents of (1,1,1), that makes the box 2x2x2 world units big.
- Position the box at (1,-1,1) using the setLocalTranslation() method.
- Wrap the Box shape into a Geometry.
- Create a blue material.
- Apply the blue material to the Box Geometry.

```
Box box1 = new Box(1,1,1);
Geometry blue = new Geometry("Box", box1);
blue.setLocalTranslation(new Vector3f(1, -1, 1));
Material mat1 = new Material(assetManager, "Common/MatDefs/Misc/Unshaded.j3md");
mat1.setColor("Color", ColorRGBA.Blue);
blue.setMaterial(mat1);
```



2. You create a second box Geometry.

- Create a second Box shape with the same size.
- Position the second box at (1,3,1). This is straight above the first box, with a gap of 2 world units inbetween.
- Wrap the Box shape into a Geometry.
- Create a red material.
- Apply the red material to the Box Geometry.

```
Box box2 = new Box(1,1,1);
Geometry red = new Geometry("Box", box2);
red.setLocalTranslation(new Vector3f(1, 3, 1));
Material mat2 = new Material(assetManager,
    "Common/MatDefs/Misc/Unshaded.j3md");
mat2.setColor("Color", ColorRGBA.Red);
red.setMaterial(mat2);
```

3. You create a pivot Node.

- Name the Node “pivot”.
- By default the Node is positioned at (0,0,0).
- Attach the Node to the rootNode.
- The Node has no visible appearance in the scene.

```
Node pivot = new Node("pivot");
rootNode.attachChild(pivot);
```

If you run the application with only the code up to here, the scene appears empty. This is because a Node is invisible, and you have not yet attached any visible Geometries to the rootNode.

4. Attach the two boxes to the pivot node.

```
pivot.attachChild(blue);
pivot.attachChild(red);
```

If you run the app with only the code up to here, you see two cubes: A red cube straight above a blue cube.

5. Rotate the pivot node.

```
pivot.rotate( 0.4f , 0.4f , 0.0f );
```

If you run the app now, you see two boxes on top of each other – both tilted at the same angle.

What is a Pivot Node?

You can transform (e.g. rotate) Geometries around their own center, or around a user defined center point. A user defined center point for one or more Geometries is called a pivot.

- In this example, you have grouped two Geometries by attaching them to one pivot Node. You use the pivot Node as a handle to rotate the two Geometries together around one common center. Rotating the pivot Node rotates all attached Geometries, in one step. The pivot node is the center of the rotation. Before attaching the other Geometries, make certain that the pivot node is at (0,0,0). Transforming a parent Node to transform all attached child Spatial is a common task. You will use this method a lot in your games when you move Spatial around.

Examples: A vehicle and its driver move together; a planet with its moon orbits the sun.

- Contrast this case with the other option: If you don't create an extra pivot node and transform a Geometry, then every transformation is done relative to the Geometry's origin (typically the center).

Examples: If you rotate each cube directly (using `red.rotate(0.1f , 0.2f , 0.3f);` and `blue.rotate(0.5f , 0.0f , 0.25f);`), then each cube is rotated individually around its center. This is similar to a planet rotating around its own center.

How do I Populate the Scenegraph?

Task...?	Solution!
Create a Spatial	<p>Create a Mesh shape, wrap it into a Geometry, and give it a Material. For example:</p> <pre>Box mesh = new Box(Vector3f.ZERO, 1, 1, 1); // a cuboid Geometry thing = new Geometry("thing", mesh); Material mat = new Material(assetManager, "Common/MatDefs/Misc/ShowNormals.j3md"); thing.setMaterial(mat);</pre>
Make an object appear in the scene	<p>Attach the Spatial to the <code>rootNode</code>, or to any node that is attached to the root:</p> <pre>rootNode.attachChild(thing);</pre>
Remove objects from the scene	<p>Detach the Spatial from the <code>rootNode</code>, and from any node that is attached to it:</p> <pre>rootNode.detachChild(thing);</pre> <pre>rootNode.detachAllChildren();</pre>
Find a Spatial in the scene by the object's name, or ID, or by its position in the parent-child hierarchy.	<p>Look at the node's children or parent:</p> <pre>Spatial thing = rootNode.getChild("thing");</pre> <pre>Spatial twentyThird = rootNode.getChild(22);</pre> <pre>Spatial parent = myNode.getParent();</pre>
Specify what should be loaded at the start	<p>Everything you initialize and attach to the <code>rootNode</code> in the <code>simpleInitApp()</code> runs in the scene at the start of the game.</p>

How do I Transform Spatials?

There are three types of 3D transformation: Translation, Scaling, and Rotation.

Translation moves Spatial

Specify the new location in three dimensions: How far away is it from the origin going right-up-forward?

To move a Spatial *to* specific coordinates, such as (0,40.2f,-2), use:

```
thing.setLocalTranslation( new Vector3f( 0.0f, 40.2f, -2.0f ) );
```

To move a Spatial *by* a certain amount, e.g. higher up (y=40.2f) and further back (z=-2.0f):

```
thing.move( 0.0f, 40.2f, -2.0f );
```

Scaling resizes Spatial	X-axis	Y-axis	Z-axis
<p>Specify the scaling factor in each dimension: length, height, width.</p> <p>A value between 0.0f and 1.0f shrinks the Spatial; bigger than 1.0f stretches it; 1.0f keeps it the same.</p> <p>Using the same value for each dimension scales proportionally, different values stretch it.</p> <p>To scale a Spatial 10 times longer, one tenth the height, and keep the same width:</p> <pre>thing.scale(10.0f, 0.1f, 1.0f);</pre>	length	height	width

Rotation turns Spatial

3-D rotation is a bit tricky ([learn details here](#)). In short: You can rotate around three axes: P. You can specify angles in degrees by multiplying the degrees value with `FastMath.DEG_TO_R`. To roll an object 180° around the z axis:

```
thing.rotate( 0f, 0f, 180*FastMath.DEG_TO_RAD );
```

Tip: If your game idea calls for a serious amount of rotations, it is worth looking into [quaternions](#) structure that can combine and store rotations efficiently.

```
thing.setLocalRotation(
    new Quaternion().fromAngleAxis(180*FastMath.DEG_TO_RAD, new Vec3f(0, 1, 0)));
```

How do I Troubleshoot Spatial?

If you get unexpected results, check whether you made the following common mistakes:

Problem?	Solution!
A created Geometry does not appear in the scene.	<p>Have you attached it to (a node that is attached to) the rootNode?</p> <p>Does it have a Material?</p> <p>What is its translation (position)? Is it behind the camera or covered up by another Geometry?</p> <p>Is it too tiny or too gigantic to see?</p> <p>Is it too far from the camera? (Try <code>cam.setFrustumFar(111111f);</code> to see further)</p>
A Spatial rotates in unexpected ways.	<p>Did you use radian values, and not degrees? (If you used degrees, multiply them with <code>FastMath.DEG_TO_RAD</code> to convert them to radians)</p> <p>Did you create the Spatial at the origin (<code>Vector.ZERO</code>) before moving it?</p> <p>Did you rotate around the intended pivot node or around something else?</p> <p>Did you rotate around the right axis?</p>
A Geometry has an unexpected Color or Material.	Did you reuse a Material from another Geometry and have inadvertently changed its properties? (If so, consider cloning it: <code>mat2 = mat.clone();</code>)

How do I Add Custom Data to Spatials?

Many Spatials represent game characters or other entities that the player can interact with. The above code that rotates the two boxes around a common center (pivot) could be used for a spacecraft docked to a orbiting space station, for example.

Depending on your game, game entities do not only change their position, rotation, or scale (the transformations that you just learned about). Game entities also have custom properties, such as health, inventory carried, equipment worn for a character, or hull strength and fuel left for a spacecraft. In Java, you represent entity data as class variables, e.g. floats, Strings, or Arrays.

You can add custom data directly to any Node or Geometry. **You do not need to extend the Node class to include variables!** For example, to add a custom id number to a node, you would use:

```
pivot.setUserdata( "pivot_id", 42 );
```

To read this Node's id number elsewhere, you would use:

```
int id = pivot.getUserData( "pivot id" );
```

By using different Strings keys (here the key is `pivot id`), you can get and set several values for whatever data the Spatial needs to carry. When you start writing your game, you might add a fuel value to a car node, speed value to an airplane node, or number of gold coins to a player node, and much more. However, one should note that only custom objects that implements Savable can be passed.

Conclusion

You have learned that your 3D scene is a scene graph made up of Spatial: Visible Geometries and invisible Nodes. You can transform Spatial, or attach them to nodes and transform the nodes. You know the easiest way how to add custom entity properties (such as player health or vehicle speed) to Spatial.

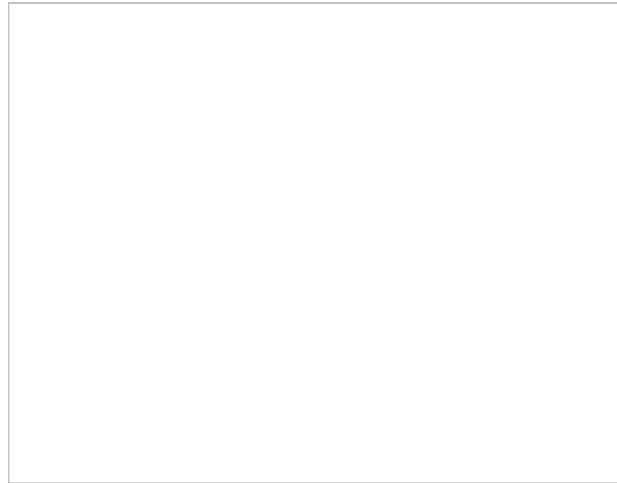
Since standard shapes like spheres and boxes get old fast, continue with the next chapter where you learn to [load assets such as 3-D models](#).

[beginner](#), [rootNode](#), [node](#), [intro](#), [documentation](#), [color](#), [spatial](#), [geometry](#), [scenegraph](#), [mesh](#)
</div>

jMonkeyEngine 3 Tutorial (3) - Hello Assets

Previous: [Hello Node](#), Next: [Hello Update Loop](#)

In this tutorial we will learn to load 3D models and text into the scene graph, using the jME [Asset Manager](#). You will also learn how to determine the correct paths, and which file formats to use.



Trouble finding the files to run this sample? To get the assets (3D models) used in this example, add the included `jME3-testdata.jar` to your classpath. In project created with the jMonkeyEngine SDK (recommended), simply right-click your project, choose “Properties”, go to “Libraries”, press “Add Library” and add the preconfigured “jme3-test-data” library.
</div>

Code Sample

```
package jme3test.helloworld;

import com.jme3.app.SimpleApplication;
import com.jme3.font.BitmapText;
import com.jme3.light.DirectionalLight;
import com.jme3.material.Material;
import com.jme3.math.Vector3f;
import com.jme3.scene.Geometry;
import com.jme3.scene.Spatial;
import com.jme3.scene.shape.Box;
```

```
/** Sample 3 - how to load an OBJ model, and OgreXML model,
 * a material/texture, or text. */
public class HelloAssets extends SimpleApplication {

    public static void main(String[] args) {
        HelloAssets app = new HelloAssets();
        app.start();
    }

    @Override
    public void simpleInitApp() {

        Spatial teapot = assetManager.loadModel("Models/Teapot/Teap");
        Material mat_default = new Material(
            assetManager, "Common/MatDefs/Misc/ShowNormals.j3md");
        teapot.setMaterial(mat_default);
        rootNode.attachChild(teapot);

        // Create a wall with a simple texture from test_data
        Box box = new Box(2.5f, 2.5f, 1.0f);
        Spatial wall = new Geometry("Box", box );
        Material mat_brick = new Material(
            assetManager, "Common/MatDefs/Misc/Unshaded.j3md");
        mat_brick.setTexture("ColorMap",
            assetManager.loadTexture("Textures/Terrain/BrickWall/Br");
        wall.setMaterial(mat_brick);
        wall.setLocalTranslation(2.0f, -2.5f, 0.0f);
        rootNode.attachChild(wall);

        // Display a line of text with a default font
        guiNode.detachAllChildren();
        guiFont = assetManager.loadFont("Interface/Fonts/Default.fr");
        BitmapText helloText = new BitmapText(guiFont, false);
        helloText.setSize(guiFont.getCharSet().getRenderedSize());
        helloText.setText("Hello World");
        helloText.setLocalTranslation(300, helloText.getLineHeight());
        guiNode.attachChild(helloText);

        // Load a model from test_data (OgreXML + material + texture)
        Spatial ninja = assetManager.loadModel("Models/Ninja/Ninja.
```

```
ninja.scale(0.05f, 0.05f, 0.05f);
ninja.rotate(0.0f, -3.0f, 0.0f);
ninja.setLocalTranslation(0.0f, -5.0f, -2.0f);
rootNode.attachChild(ninja);
// You must add a light to make the model visible
DirectionalLight sun = new DirectionalLight();
sun.setDirection(new Vector3f(-0.1f, -0.7f, -1.0f));
rootNode.addLight(sun);

}

}
```

Build and run the code sample. You should see a green Ninja with a colorful teapot standing behind a wall. The text on the screen should say “Hello World”.

The Asset Manager

By game assets we mean all multi-media files, such as models, materials, textures, whole scenes, custom shaders, music and sound files, and custom fonts. JME3 comes with a handy AssetManager object that helps you access your assets. The AssetManager can load files from:

- the current classpath (the top level of your project directory),
- the `assets` directory of your project, and
- optionally, custom paths that you register.

The following is the recommended directory structure for storing assets in your project directoy:

```

MyGame/assets/
MyGame/assets/Interface/
MyGame/assets/MatDefs/
MyGame/assets/Materials/
MyGame/assets/Models/      <-- your .j3o models go here
MyGame/assets/Scenes/
MyGame/assets/Shaders/
MyGame/assets/Sounds/     <-- your audio files go here
MyGame/assets/Textures/   <-- your textures go here
MyGame/build.xml          <-- Default Ant build script
MyGame/src/...             <-- your Java sources go here
MyGame/...

```

This is just a suggested best practice, and it's what you get by default when creating a new Java project in the jMokeyEngine [SDK](#). You can create an `assets` directory and technically name the subdirectories whatever you like.

Loading Textures

Place your textures in a subdirectory of `assets/Textures/`. Load the texture into the material before you set the Material. The following code sample is from the `simpleInitApp()` method and loads a simple wall model:

```

// Create a wall with a simple texture from test_data
Box box = new Box(2.5f,2.5f,1.0f);
Spatial wall = new Geometry("Box", box );
Material mat_brick = new Material(
    assetManager, "Common/MatDefs/Misc/Unshaded.j3md");
mat_brick.setTexture("ColorMap",
    assetManager.loadTexture("Textures/Terrain/BrickWall/BrickWall.
wall.setMaterial(mat_brick);
wall.setLocalTranslation(2.0f, -2.5f, 0.0f);
rootNode.attachChild(wall);

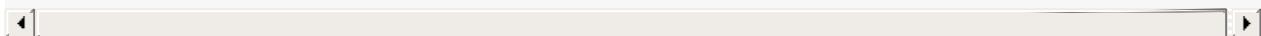
```

In this case, you [create your own Material](#) and apply it to a Geometry. You base Materials on default material descriptions (such as “Unshaded.j3md”), as shown in this example.

Loading Text and Fonts

This example displays the text “Hello World” in the default font at the bottom edge of the window. You attach text to the `guiNode` – this is a special node for flat (orthogonal) display elements. You display text to show the game score, player health, etc. The following code sample goes into the `simpleInitApp()` method.

```
// Display a line of text with a default font
guiNode.detachAllChildren();
guiFont = assetManager.loadFont("Interface/Fonts/Default.fnt");
BitmapText helloText = new BitmapText(guiFont, false);
helloText.setSize(guiFont.getCharSet().getRenderedSize());
helloText.setText("Hello World");
helloText.setLocalTranslation(300, helloText.getLineHeight(), 0);
guiNode.attachChild(helloText);
```



Tip: Clear existing text in the `guiNode` by detaching all its children.

Loading a Model

Export your 3D model in OgreXML format (.mesh.xml, .scene, .material, .skeleton.xml) and place it in a subdirectory of `assets/Models/`. The following code sample goes into the `simpleInitApp()` method.

```
// Load a model from test_data (OgreXML + material + texture)
Spatial ninja = assetManager.loadModel("Models/Ninja/Ninja.mesh.xml");
ninja.scale(0.05f, 0.05f, 0.05f);
ninja.rotate(0.0f, -3.0f, 0.0f);
ninja.setLocalTranslation(0.0f, -5.0f, -2.0f);
rootNode.attachChild(ninja);
// You must add a directional light to make the model visible!
DirectionalLight sun = new DirectionalLight();
sun.setDirection(new Vector3f(-0.1f, -0.7f, -1.0f).normalizeLocal());
rootNode.addLight(sun);
```



Note that you do not need to create a Material if you exported the model with a material. Remember to add a light source, as shown, otherwise the material (and the whole model) is not visible!

Loading Assets From Custom Paths

What if your game relies on user supplied model files, that are not included in the distribution? If a file is not located in the default location (e.g. assets directory), you can register a custom Locator and load it from any path.

Here is a usage example of a ZipLocator that is registered to a file `town.zip` in the top level of your project directory:

```
assetManager.registerLocator("town.zip", ZipLocator.class);
Spatial scene = assetManager.loadModel("main.scene");
rootNode.attachChild(scene);
```

Here is a HttpZipLocator that can download zipped models and load them:

```
assetManager.registerLocator(
    "http://jmonkeyengine.googlecode.com/files/wildhouse.zip",
    HttpZipLocator.class);
Spatial scene = assetManager.loadModel("main.scene");
rootNode.attachChild(scene);
```

JME3 offers ClasspathLocator, ZipLocator, FileLocator, HttpZipLocator, and UrlLocator (see `com.jme3.asset.plugins`).

Creating Models and Scenes

To create 3D models and scenes, you need a 3D Mesh Editor. If you don't have any tools, install Blender and the OgreXML Exporter plugin. Then you [create fully textured models \(e.g. with Blender\)](#) and export them to your project. Then you use the [SDK to load models, convert models](#), and [create 3D scenes](#) from them.

Example: From Blender, you export your models as Ogre XML meshes with materials as follows:

1. Open the menu File > Export > OgreXML Exporter to open the exporter dialog.
2. In the Export Materials field: Give the material the same name as the model. For example, the model `something.mesh.xml` goes with `something.material`, plus (optionally) `something.skeleton.xml` and some JPG texture files.
3. In the Export Meshes field: Select a subdirectory of your `assets/Models/` directory. E.g. `assets/Models/something/`.
4. Activate the following exporter settings:

- Copy Textures: YES
- Rendering Materials: YES
- Flip Axis: YES
- Require Materials: YES
- Skeleton name follows mesh: YES

5. Click export.

Model File Formats

JME3 can convert and load

- Ogre XML models + materials,
- Ogre DotScenes,
- Wavefront OBJ + MTL models,
- .Blend files.

The `loadModel()` method loads these original file formats when you run your code directly from the SDK. If you however build the executables using the default build script, then the original model files (XML, OBJ, etc) are *not included*. This means, when you run the executable outside the SDK, and load any original models directly, you get the following error message:

```
com.jme3.asset.DesktopAssetManager loadAsset
WARNING: Cannot locate resource: Models/Ninja/Ninja.mesh.xml
com.jme3.app.Application handleError
SEVERE: Uncaught exception thrown in Thread[LWJGL Renderer Thread,5
java.lang.NullPointerException
```

You see that loading the **XML/OBJ/Blend files** directly is only acceptable during the development phase in the SDK. For example, every time your graphic designer pushes updated files to the asset directory, you can quickly review the latest version in your development environment.

But for QA test builds and for the final release build, you use **.j3o files** exclusively. J3o is an optimized binary format for jME3 applications. When you do QA test builds, or are ready to release, use the [SDK to convert](#) all .obj/.scene/.xml/.blend files to .j3o files, and update all code to load the .j3o files. The default build script automatically packages .j3o files in the executables.

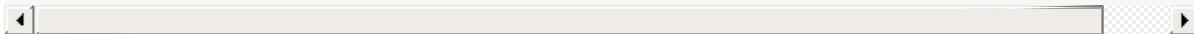
Open your JME3 Project in the jMonkeyEngine SDK.

1. Right-click a .Blend, .OBJ, or .mesh.xml file in the Projects window, and choose “convert

to JME3 binary".

2. The .j3o file appears next to the .mesh.xml file and has the same name.
3. Update all your `loadModel()` lines accordingly. For example:

```
Spatial ninja = assetManager.loadModel("Models/Ninja/Ninja.j3o")
```



If your executable throws a "Cannot locate resource" runtime exception, check all load paths and make sure you have converted all models to .j3o files!

</div>

Loading Models and Scenes

Task?	Solution!
Load a model with materials	<p>Use the asset manager's <code>loadModel()</code> method and attach the Spatial to the root node:</p> <pre>Spatial elephant = assetManager.loadModel("Models/Elephant.j3o"); rootNode.attachChild(elephant);</pre> <p>Or, if you want to use materials:</p> <pre>Spatial elephant = assetManager.loadModel("Models/Elephant.j3o"); Material mat = new Material(assetManager, "Common/MatDefs/Misc/Unshaded.jcm"); elephant.setMaterial(mat); rootNode.attachChild(elephant);</pre>
Load a model without materials	<p>If you have a model without materials, you have to give it a material to make it render:</p> <pre>Spatial teapot = assetManager.loadModel("Models/Teapot.j3o"); Material mat = new Material(assetManager, "Common/MatDefs/Misc/Unshaded.jcm"); teapot.setMaterial(mat); rootNode.attachChild(teapot);</pre>
Load a scene	<p>You load scenes just like you load models:</p> <pre>Spatial scene = assetManager.loadModel("Scenes/town/mair.j3o"); rootNode.attachChild(scene);</pre> <p>Or, if you want to use materials:</p> <pre>Spatial scene = assetManager.loadModel("Scenes/town/mair.j3o"); Material mat = new Material(assetManager, "Common/MatDefs/Misc/Unshaded.jcm"); scene.setMaterial(mat); rootNode.attachChild(scene);</pre>

Excercise - How to Load Assets

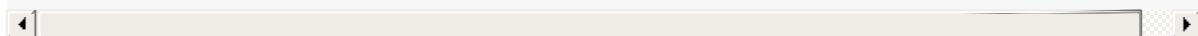
As an exercise, let's try different ways of loading a scene. You will learn how to load the scene directly, or from a zip file.

1. [Download the town.zip sample scene](#).
2. (Optional:) Unzip the town.zip to see the structure of the contained Ogre dotScene:
You'll get a directory named `town`. It contains XML and texture files, and file called `main.scene`. (This is just for your information, you do not need to do anything with it.)
3. Place the town.zip file in the top level directory of your JME3 project, like so:

```
jMonkeyProjects/MyGameProject/assets/
jMonkeyProjects/MyGameProject/build.xml
jMonkeyProjects/MyGameProject/src/
jMonkeyProjects/MyGameProject/town.zip
...
...
```

Use the following method to load models from a zip file:

1. Verify `town.zip` is in the project directory.
 2. Register a zip file locator to the project directory: Add the following code under `simpleInitApp()` {
- ```
assetManager.registerLocator("town.zip", ZipLocator.class);
Spatial gameLevel = assetManager.loadModel("main.scene");
gameLevel.setLocalTranslation(0, -5.2f, 0);
gameLevel.setLocalScale(2);
rootNode.attachChild(gameLevel);
```



The `loadModel()` method now searches this zip directly for the files to load.

(This means, do not write `loadModel(town.zip/main.scene)` or similar!)

3. Clean, build and run the project.  
You should now see the Ninja+wall+teapot standing in a town.

**Tip:** If you register new locators, make sure you do not get any file name conflicts: Don't name all scenes `main.scene` but give each scene a unique name.

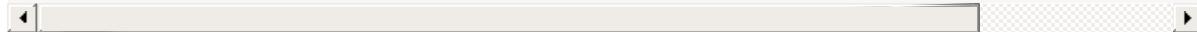
Earlier in this tutorial, you loaded scenes and models from the asset directory. This is the most common way you will be loading scenes and models. Here is the typical procedure:

1. Remove the code that you added for the previous exercise.
2. Move the unzipped `town/` directory into the `assets/Scenes/` directory of your project.
3. Add the following code under `simpleInitApp()` {

```

 Spatial gameLevel = assetManager.loadModel("Scenes/town/main.j3o");
 gameLevel.setLocalTranslation(0, -5.2f, 0);
 gameLevel.setLocalScale(2);
 rootNode.attachChild(gameLevel);

```



Note that the path is relative to the `assets/...` directory.

- Clean, build and run the project. Again, you should see the Ninja+wall+teapot standing in a town.

Here is a third method you must know, loading a scene/model from a `.j3o` file:

- Remove the code from the previous exercise.
- If you haven't already, open the [SDK](#) and open the project that contains the `HelloAsset` class.
- In the projects window, browse to the `assets/Scenes/town` directory.
- Right-click the `main.scene` and convert the scene to binary: The jMonkeyPlatform generates a `main.j3o` file.
- Add the following code under `simpleInitApp() {`

```

 Spatial gameLevel = assetManager.loadModel("Scenes/town/main.j3o");
 gameLevel.setLocalTranslation(0, -5.2f, 0);
 gameLevel.setLocalScale(2);
 rootNode.attachChild(gameLevel);

```



Again, note that the path is relative to the `assets/...` directory.

- Clean, Build and run the project.

Again, you should see the Ninja+wall+teapot standing in a town.

## Conclusion

Now you know how to populate the scenegraph with static shapes and models, and how to build scenes. You have learned how to load assets using the `assetManager` and you have seen that the paths start relative to your project directory. Another important thing you have learned is to convert models to `.j3o` format for the executable JARs etc.

Let's add some action to the scene and continue with the [Update Loop!](#)

**See also:**

- [The definitive Blender import tutorial](#)
- [Asset pipeline introduction](#)
- If you want to learn how to load sounds, see [Hello Audio](#)
- If you want to learn more about loading textures and materials, see [Hello Material](#)

[beginner](#), [intro](#), [documentation](#), [lightnode](#), [material](#), [model](#), [node](#), [gui](#), [hud](#), [texture](#)

</div>

# title: jMonkeyEngine 3 Tutorial (4) - Hello Update Loop

## jMonkeyEngine 3 Tutorial (4) - Hello Update Loop

Previous: [Hello Assets](#), Next: [Hello Input System](#)

Now that you know how to load assets, such as 3D models, you want to implement some gameplay that uses these assets. In this tutorial we look at the update loop. The update loop of your game is where the action happens.

### Code Sample

```
package jme3test.helloworld;

import com.jme3.app.SimpleApplication;
import com.jme3.material.Material;
import com.jme3.math.ColorRGBA;
import com.jme3.scene.Geometry;
import com.jme3.scene.shape.Box;

/** Sample 4 - how to trigger repeating actions from the main event
 * In this example, you use the loop to make the player character
 * rotate continuously. */
public class HelloLoop extends SimpleApplication {

 public static void main(String[] args){
 HelloLoop app = new HelloLoop();
 app.start();
 }

 protected Geometry player;
```

```

@Override
public void simpleInitApp() {
 /** this blue box is our player character */
 Box b = new Box(1, 1, 1);
 player = new Geometry("blue cube", b);
 Material mat = new Material(assetManager,
 "Common/MatDefs/Misc/Unshaded.j3md");
 mat.setColor("Color", ColorRGBA.Blue);
 player.setMaterial(mat);
 rootNode.attachChild(player);
}

/* Use the main event loop to trigger repeating actions. */
@Override
public void simpleUpdate(float tpf) {
 // make the player rotate:
 player.rotate(0, 2*tpf, 0);
}

```

Build and run the file: You see a constantly rotating cube.

## Understanding the Code

Compared to our previous code samples you note that the player Geometry is now a class field. This is because we want the update loop to be able to access and transform this Geometry. As usual, we initialize the player object in the `simpleInitApp()` method.

Now have a closer look at the `simpleUpdate()` method – this is the update loop.

- The `player.rotate(0, 2*tpf, 0);` line changes the rotation of the player object.
- We use the `tpf` variable (“time per frame”) to time this action depending on the current frames per second rate. This simply means that the cube rotates with the same speed on fast and slow machines, and the game remains playable.
- When the game runs, the `rotate()` code is executed again and again.

## Using the Update Loop

A rotating object is just a simple example. In the update loop, you typically have many tests and trigger various game actions. This is where you update score and health points, check for collisions, make enemies calculate their next move, roll the dice whether a trap has been set off, play random ambient sounds, and much more.

- The `simpleUpdate()` method starts running after the `simpleInitApp()` method has initialized the scene graph and state variables.
- JME3 executes everything in the `simpleUpdate()` method repeatedly, as fast as possible.
  1. Use the loop to poll the game state and then initiate actions.
  2. Use the loop to trigger reactions and update the game state.
  3. Use the loop wisely, because having too many calls in the loop also slows down the game.

## Init - Update - Render

Note the the three phases of every game:

- **Init:** The `simpleInitApp()` method is executed only *once*, right at the beginning;
- **Update:** The `simpleUpdate()` method runs *repeatedly*, during the game.
- **Render:** After every update, the jMonkeyEngine *automatically* redraws ( `renders` ) the screen for you.

Since rendering is automatic, initialization and updating are the two most important concepts in a SimpleApplication-based game for you:

- The `simpleInitApp()` method is the application's "first breath".  
Here you load and create game data (once).
- The `simpleUpdate()` method is the application's "heartbeat" (the time unit is called `ticks` ).  
Here you change their properties to update the game state (repeatedly).

Everything in a game happens either during initialization, or during the update loop. This means that these two methods grow very long over time. Follow these two strategies to spread out init and update code over several modular Java classes:

- Move code blocks from the `simpleInitApp()` method to [AppStates](#).
- Move code blocks from the `simpleUpdate()` method to [Custom Controls](#).

Keep this in mind for later when your application grows.

</div>

## Exercises

Here are some fun things to try:

1. What happens if you give the `rotate()` method negative numbers?
2. Can you create two Geometries next to each other, and make one rotate twice as fast as the other? (use the `tpf` variable)
3. Can you make a cube that pulsates? (grows and shrinks)
4. Can you make a cube that changes color? (change and set the Material)
5. Can you make a rolling cube? (rotate around the x axis, and translate along the z axis)

Look back at the [Hello Node](#) tutorial if you do not remember the transformation methods for scaling, translating, and rotating.

Link to user-proposed solutions: <http://jmonkeyengine.org/wiki/doku.php/jm3:solutions> Be sure to try to solve them for yourself first!

</div>

## Conclusion

Now you are listening to the update loop, “the heart beat” of the game, and you can add all kinds of action to it.

The next thing the game needs is some *interaction*! Continue learning how to [respond to user input](#).

---

See also:

- Advanced jME3 developers use [Application States](#) and [Custom Controls](#) to implement game mechanics in their update loops. You will come across these topics again later when you proceed to more advanced documentation.

[documentation](#), [state](#), [states](#), [intro](#), [beginner](#), [control](#), [loop](#)

</div>

# title: jMonkeyEngine 3 Tutorial (5) - Hello Input System

## jMonkeyEngine 3 Tutorial (5) - Hello Input System

Previous: [Hello Update Loop](#), Next: [Hello Material](#)

By default, SimpleApplication sets up a camera control that allows you to steer the camera with the WASD keys, the arrow keys, and the mouse. You can use it as a flying first-person camera right away. But what if you need a third-person camera, or you want keys to trigger special game actions?

Every game has its custom keybindings, and this tutorial explains how you define them. We first define the key presses and mouse events, and then we define the actions they should trigger.

## Sample Code

```
package jme3test.helloworld;

import com.jme3.app.SimpleApplication;
import com.jme3.material.Material;
import com.jme3.math.Vector3f;
import com.jme3.scene.Geometry;
import com.jme3.scene.shape.Box;
import com.jme3.math.ColorRGBA;
import com.jme3.input.KeyInput;
import com.jme3.input.MouseInput;
import com.jme3.input.controls.ActionListener;
import com.jme3.input.controls.AnalogListener;
import com.jme3.input.controls.KeyTrigger;
import com.jme3.input.controls.MouseButtonTrigger;

/** Sample 5 - how to map keys and mousebuttons to actions */
```

```
public class HelloInput extends SimpleApplication {

 public static void main(String[] args) {
 HelloInput app = new HelloInput();
 app.start();
 }
 protected Geometry player;
 Boolean isRunning=true;

 @Override
 public void simpleInitApp() {
 Box b = new Box(1, 1, 1);
 player = new Geometry("Player", b);
 Material mat = new Material(assetManager, "Common/MatDefs/Misc/
mat.setColor("Color", ColorRGBA.Blue);
 player.setMaterial(mat);
 rootNode.attachChild(player);
 initKeys(); // load my custom keybinding
 }

 /** Custom Keybinding: Map named actions to inputs. */
 private void initKeys() {
 // You can map one or several inputs to one named action
 inputManager.addMapping("Pause", new KeyTrigger(KeyInput.KEY_F
 inputManager.addMapping("Left", new KeyTrigger(KeyInput.KEY_J
 inputManager.addMapping("Right", new KeyTrigger(KeyInput.KEY_K
 inputManager.addMapping("Rotate", new KeyTrigger(KeyInput.KEY_S
 new MouseButtonTrigger(MouseI
 // Add the names to the action listener.
 inputManager.addListener(actionListener,"Pause");
 inputManager.addListener(analogListener,"Left", "Right", "Rotat

 }

 private ActionListener actionListener = new ActionListener() {
 public void onAction(String name, boolean keyPressed, float tpf
 if (name.equals("Pause") && !keyPressed) {
 isRunning = !isRunning;
 }
 }
 }
}
```

```

};

private AnalogListener analogListener = new AnalogListener() {
 public void onAnalog(String name, float value, float tpf) {
 if (isRunning) {
 if (name.equals("Rotate")) {
 player.rotate(0, value*speed, 0);
 }
 if (name.equals("Right")) {
 Vector3f v = player.getLocalTranslation();
 player.setLocalTranslation(v.x + value*speed, v.y, v.z);
 }
 if (name.equals("Left")) {
 Vector3f v = player.getLocalTranslation();
 player.setLocalTranslation(v.x - value*speed, v.y, v.z);
 }
 } else {
 System.out.println("Press P to unpause.");
 }
 }
};
}

```



Build and run the example.

- Press the Spacebar or click to rotate the cube.
- Press the J and K keys to move the cube.
- Press P to pause and unpause the game. While paused, the game should not respond to any input, other than P .

## Defining Mappings and Triggers

First you register each mapping name with its trigger(s). Remember the following:

- An input trigger can be a key press or mouse action.  
For example a mouse movement, a mouse click, or pressing the letter “P”.
- The mapping name is a string that you can choose.  
The name should describe the action (e.g. “Rotate”), and not the trigger. Because the trigger can change.

- One named mapping can have several triggers.

For example, the “Rotate” action can be triggered by a click and by pressing the spacebar.

Have a look at the code:

1. You register the mapping named “Rotate” to the Spacebar key trigger.

```
new KeyTrigger(KeyInput.KEY_SPACE)).
```

2. In the same line, you also register “Rotate” to an alternative mouse click trigger.

```
new MouseButtonTrigger(MouseInput.BUTTON_LEFT)
```

3. You map the `Pause` , `Left` , `Right` mappings to the P, J, K keys, respectively.

```
// You can map one or several inputs to one named action
inputManager.addMapping("Pause", new KeyTrigger(KeyInput.KEY_F));
inputManager.addMapping("Left", new KeyTrigger(KeyInput.KEY_J));
inputManager.addMapping("Right", new KeyTrigger(KeyInput.KEY_K));
inputManager.addMapping("Rotate", new KeyTrigger(KeyInput.KEY_S));
new MouseButtonTrigger(MouseInput.BUTTON_LEFT))
```

Now you need to register your trigger mappings.

1. You register the pause action to the ActionListener, because it is an “on/off” action.
2. You register the movement actions to the AnalogListener, because they are gradual actions.

```
// Add the names to the action listener.
inputManager.addListener(actionListener, "Pause");
inputManager.addListener(analogListener, "Left", "Right", "Rotate");
```

This code goes into the `simpleInitApp()` method. But since we will likely add many keybindings, we extract these lines and wrap them in an auxiliary method, `initKeys()` . The `initKeys()` method is not part of the Input Controls interface – you can name it whatever you like. Just don't forget to call your method from the `initSimpleApp()` method.

## Implementing the Actions

You have mapped action names to input triggers. Now you specify the actions themselves.

The two important methods here are the `ActionListener` with its `onAction()` method, and the `AnalogListener` with its `onAnalog()` method. In these two methods, you test for each named mapping, and call the game action you want to trigger.

In this example, we trigger the following actions:

1. The *Rotate* mapping triggers the action `player.rotate(0, value, 0)`.
2. The *Left* and *Right* mappings increase and decrease the player's x coordinate.
3. The *Pause* mapping flips a boolean `isRunning`.
4. We also want to check the boolean `isRunning` before any action (other than unpause) is executed.

```
private ActionListener actionListener = new ActionListener() {
 public void onAction(String name, boolean keyPressed, float tpf) {
 if (name.equals("Pause") && !keyPressed) {
 isRunning = !isRunning;
 }
 }
};

private AnalogListener analogListener = new AnalogListener() {
 public void onAnalog(String name, float value, float tpf) {
 if (isRunning) {
 if (name.equals("Rotate")) {
 player.rotate(0, value*speed, 0);
 }
 if (name.equals("Right")) {
 Vector3f v = player.getLocalTranslation();
 player.setLocalTranslation(v.x + value*speed, v.y, v.z);
 }
 if (name.equals("Left")) {
 Vector3f v = player.getLocalTranslation();
 player.setLocalTranslation(v.x - value*speed, v.y, v.z);
 }
 } else {
 System.out.println("Press P to unpause.");
 }
 }
};
```

You can also combine both listeners into one, the engine will send the appropriate events to each method (onAction or onAnalog). For example:

```
private MyCombinedListener combinedListener = new MyCombinedListener();

private static class MyCombinedListener implements AnalogListener {
 public void onAction(String name, boolean keyPressed, float tpf) {
 if (name.equals("Pause") && !keyPressed) {
 isRunning = !isRunning;
 }
 }

 public void onAnalog(String name, float value, float tpf) {
 if (isRunning) {
 if (name.equals("Rotate")) {
 player.rotate(0, value*speed, 0);
 }
 if (name.equals("Right")) {
 Vector3f v = player.getLocalTranslation();
 player.setLocalTranslation(v.x + value*speed, v.y, v.z);
 }
 if (name.equals("Left")) {
 Vector3f v = player.getLocalTranslation();
 player.setLocalTranslation(v.x - value*speed, v.y, v.z);
 }
 } else {
 System.out.println("Press P to unpause.");
 }
 }
}
// ...
inputManager.addListener(combinedListener, new String[]{"Pause", "L
```

It's okay to use only one of the two Listeners, and not implement the other one, if you are not using this type of interaction. In the following, we have a closer look how to decide which of the two listeners is best suited for which situation.

# Analog, Pressed, or Released?

Technically, every input can be either an “analog” or a “digital” action. Here is how you find out which listener is the right one for which type of input.

Mappings registered to the **AnalogListener** are triggered repeatedly and gradually.

- Parameters:
  1. JME gives you access to the name of the triggered action.
  2. JME gives you access to a gradual value showing the strength of that input. In the case of a keypress that will be the tpf value for which it was pressed since the last frame. For other inputs such as a joystick which give analogue control though then the value will also indicate the strength of the input premultiplied by tpf. For an example on this go to [jMonkeyEngine 3 Tutorial \(5\) - Hello Input System - Variation over time key is pressed](#)

In order to see the total time that a key has been pressed for then the incoming value can be accumulated. The analogue listener may also need to be combined with an action listener so that you are notified when the key is released.

- Example: Navigational events (e.g. Left, Right, Rotate, Run, Strafe), situations where you interact continuously.

Mappings registered to the **ActionListener** are digital either-or actions – “Pressed or released? On or off?”

- Parameters:
  1. JME gives you access to the name of the triggered action.
  2. JME gives you access to a boolean whether the key is pressed or not.
- Example: Pause button, shooting, selecting, jumping, one-time click interactions.

**Tip:** It's very common that you want an action to be only triggered once, in the moment when the key is *released*. For instance when opening a door, flipping a boolean state, or picking up an item. To achieve that, you use an `ActionListener` and test for `... && !keyPressed`. For an example, look at the Pause button code:

```
if (name.equals("Pause") && !keyPressed) {
 isRunning = !isRunning;
}
```

## Table of Triggers

You can find the list of input constants in the files `src/core/com/jme3/input/KeyInput.java`, `JoyInput.java`, and `MouseInput.java`. Here is an overview of the most common triggers constants:

| Trigger                          | Code                                                                                                                              |
|----------------------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| Mouse button: Left Click         | MouseButtonTrigger(MouseInput.BUTTON_LEFT)                                                                                        |
| Mouse button: Right Click        | MouseButtonTrigger(MouseInput.BUTTON_RIGHT)                                                                                       |
| Keyboard: Characters and Numbers | KeyTrigger(KeyInput.KEY_X)                                                                                                        |
| Keyboard: Spacebar               | KeyTrigger(KeyInput.KEY_SPACE)                                                                                                    |
| Keyboard: Return, Enter          | KeyTrigger(KeyInput.KEY_RETURN),<br>KeyTrigger(KeyInput.KEY_NUMPADENTER)                                                          |
| Keyboard: Escape                 | KeyTrigger(KeyInput.KEY_ESCAPE)                                                                                                   |
| Keyboard: Arrows                 | KeyTrigger(KeyInput.KEY_UP),<br>KeyTrigger(KeyInput.KEY_DOWN)<br>KeyTrigger(KeyInput.KEY_LEFT),<br>KeyTrigger(KeyInput.KEY_RIGHT) |

**Tip:** If you don't recall an input constant during development, you benefit from an IDE's code completion functionality: Place the caret after e.g. `KeyInput.` and trigger code completion to select possible input identifiers.

## Exercises

1. Add mappings for moving the player (box) up and down with the H and L keys!
2. Switch off the flyCam and override the WASD keys.
  - o Tip: Use `flyCam.setEnabled(false);`
3. Modify the mappings so that you can also trigger the up and down motion with the mouse scroll wheel!
  - o Tip: Use `new MouseAxisTrigger(MouseInput.AXIS_WHEEL, true)`
4. In which situation would it be better to use variables instead of literals for the MouseInput/KeyInput definitions?

```
int usersPauseKey = KeyInput.KEY_P;
...
inputManager.addMapping("Pause", new KeyTrigger(usersPauseKey))
```

Link to user-proposed solutions: <http://jmonkeyengine.org/wiki/doku.php/jm3:solutions> Be

*sure to try to solve them for yourself first!*

</div>

## Conclusion

You are now able to add custom interactions to your game: You know that you first have to define the key mappings, and then the actions for each mapping. You have learned to respond to mouse events and to the keyboard. You understand the difference between “analog” (gradually repeated) and “digital” (on/off) inputs.

Now you can already write a little interactive game! But wouldn't it be cooler if these old boxes were a bit more fancy? Let's continue with learning about [materials](#).

[input](#), [intro](#), [beginner](#), [documentation](#), [keyinput](#), [click](#)

</div>

# title: jMonkeyEngine 3 Tutorial (6) - Hello Materials

## jMonkeyEngine 3 Tutorial (6) - Hello Materials

Previous: [Hello Input System](#), Next: [Hello Animation](#)

The term Material includes everything that influences what the surface of a 3D model looks like: The color, texture, shininess, and opacity/transparency. Plain coloring is covered in [Hello Node](#). Loading models that come with materials is covered in [Hello Asset](#). In this tutorial you learn to create and use custom JME3 Material Definitions.



To use the example assets in a new jMonkeyEngine SDK project, right-click your project, select “Properties”, go to “Libraries”, press “Add Library” and add the “jme3-test-data” library.  
</div>

## Sample Code

```
package jme3test.helloworld;

import com.jme3.app.SimpleApplication;
import com.jme3.lightDirectionalLight;
import com.jme3.material.Material;
import com.jme3.material.RenderState.BlendMode;
```

```
import com.jme3.math.ColorRGBA;
import com.jme3.math.Vector3f;
import com.jme3.renderer.queue.RenderQueue.Bucket;
import com.jme3.scene.Geometry;
import com.jme3.scene.shape.Box;
import com.jme3.scene.shape.Sphere;
import com.jme3.texture.Texture;
import com.jme3.util.TangentBinormalGenerator;

/** Sample 6 - how to give an object's surface a material and texture
 * How to make objects transparent. How to make bumpy and shiny surfaces.
public class HelloMaterial extends SimpleApplication {

 public static void main(String[] args) {
 HelloMaterial app = new HelloMaterial();
 app.start();
 }

 @Override
 public void simpleInitApp() {

 /** A simple textured cube -- in good MIP map quality. */
 Box cube1Mesh = new Box(1f,1f,1f);
 Geometry cube1Geo = new Geometry("My Textured Box", cube1Mesh);
 cube1Geo.setLocalTranslation(new Vector3f(-3f,1.1f,0f));
 Material cube1Mat = new Material(assetManager,
 "Common/MatDefs/Misc/Unshaded.j3md");
 Texture cube1Tex = assetManager.loadTexture(
 "Interface/Logo/Monkey.jpg");
 cube1Mat.setTexture("ColorMap", cube1Tex);
 cube1Geo.setMaterial(cube1Mat);
 rootNode.attachChild(cube1Geo);

 /** A translucent/transparent texture, similar to a window frame */
 Box cube2Mesh = new Box(1f,1f,0.01f);
 Geometry cube2Geo = new Geometry("window frame", cube2Mesh);
 Material cube2Mat = new Material(assetManager,
 "Common/MatDefs/Misc/Unshaded.j3md");
 cube2Mat.setTexture("ColorMap",
 assetManager.loadTexture("Textures/ColoredTex/Monkey.png"))
 }
}
```

```

 cube2Mat.getAdditionalRenderState().setBlendMode(BlendMode.Alpha);
 cube2Geo.setQueueBucket(Bucket.Transparent);
 cube2Geo.setMaterial(cube2Mat);
 rootNode.attachChild(cube2Geo);

 /** A bumpy rock with a shiny light effect.*/
 Sphere sphereMesh = new Sphere(32, 32, 2f);
 Geometry sphereGeo = new Geometry("Shiny rock", sphereMesh);
 sphereMesh.setTextureMode(Sphere.TextureMode.Projected); // better
 TangentBinormalGenerator.generate(sphereMesh); // for
 Material sphereMat = new Material(assetManager,
 "Common/MatDefs/Light/Lighting.j3md");
 sphereMat.setTexture("DiffuseMap",
 assetManager.loadTexture("Textures/Terrain/Pond/Pond.jpg"));
 sphereMat.setTexture("NormalMap",
 assetManager.loadTexture("Textures/Terrain/Pond/Pond_normal"));
 sphereMat.setBoolean("UseMaterialColors", true);
 sphereMat.setColor("Diffuse", ColorRGBA.White);
 sphereMat.setColor("Specular", ColorRGBA.White);
 sphereMat.SetFloat("Shininess", 64f); // [0,128]
 sphereGeo.setMaterial(sphereMat);
 sphereGeo.setLocalTranslation(0, 2, -2); // Move it a bit
 sphereGeo.rotate(1.6f, 0, 0); // Rotate it a bit
 rootNode.attachChild(sphereGeo);

 /** Must add a light to make the lit object visible! */
 DirectionalLight sun = new DirectionalLight();
 sun.setDirection(new Vector3f(1, 0, -2).normalizeLocal());
 sun.setColor(ColorRGBA.White);
 rootNode.addLight(sun);

 }
}

```

You should see

- Left – A cube with a brown monkey texture.
- Right – A translucent monkey picture in front of a shiny bumpy rock.

Move around with the WASD keys to have a closer look at the translucency, and the rock's bumpiness.

## Simple Unshaded Texture

Typically you want to give objects in your scene textures: It can be rock, grass, brick, wood, water, metal, paper... A texture is a normal image file in JPG or PNG format. In this example, you create a box with a simple unshaded Monkey texture as material.

```
/** A simple textured cube -- in good MIP map quality. */
Box cube1Mesh = new Box(1f,1f,1f);
Geometry cube1Geo = new Geometry("My Textured Box", cube1Mesh);
cube1Geo.setLocalTranslation(new Vector3f(-3f,1.1f,0f));
Material cube1Mat = new Material(assetManager,
 "Common/MatDefs/Misc/Unshaded.j3md");
Texture cube1Tex = assetManager.loadTexture(
 "Interface/Logo/Monkey.jpg");
cube1Mat.setTexture("ColorMap", cube1Tex);
cube1Geo.setMaterial(cube1Mat);
rootNode.attachChild(cube1Geo);
```

Here is what we did: to create a textured box:

1. Create a Geometry `cube1Geo` from a Box mesh `cube1Mesh`.
2. Create a Material `cube1Mat` based on jME3's default `Unshaded.j3md` material definition.
3. Create a texture `cube1Tex` from the `Monkey.jpg` file in the `assets/Interface/Logo/` directory of the project.
4. Load the texture `cube1Tex` into the `ColorMap` layer of the material `cube1Mat`.
5. Apply the material to the cube, and attach the cube to the rootnode.

## Transparent Unshaded Texture

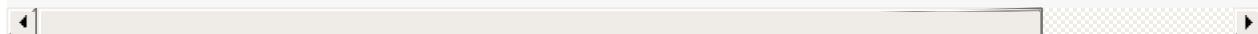
`Monkey.png` is the same texture as `Monkey.jpg`, but with an added alpha channel. The alpha channel allows you to specify which areas of the texture you want to be opaque or transparent: Black areas of the alpha channel remain opaque, gray areas become translucent, and white areas become transparent.

For a partially translucent/transparent texture, you need:

- A Texture with alpha channel
- A Texture with blend mode of `BlendMode.Alpha`
- A Geometry in the `Bucket.Transparent` render bucket.

This bucket ensures that the transparent object is drawn on top of objects behind it, and they show up correctly under the transparent parts.

```
/** A translucent/transparent texture, similar to a window frame */
Box cube2Mesh = new Box(1f,1f,0.01f);
Geometry cube2Geo = new Geometry("window frame", cube2Mesh);
Material cube2Mat = new Material(assetManager,
"Common/MatDefs/Misc/Unshaded.j3md");
cube2Mat.setTexture("ColorMap",
assetManager.loadTexture("Textures/ColoredTex/Monkey.png"))
cube2Mat.getAdditionalRenderState().setBlendMode(BlendMode.Alpha);
cube2Geo.setQueueBucket(Bucket.Transparent);
cube2Geo.setMaterial(cube2Mat);
rootNode.attachChild(cube2Geo);
```



For non-transparent objects, the drawing order is not so important, because the z-buffer already keeps track of whether a pixel is behind something else or not, and the color of an opaque pixel doesn't depend on the pixels under it, this is why opaque Geometries can be drawn in any order.

What you did for the transparent texture is the same as before, with only one added step for the transparency.

1. Create a Geometry `cube2Geo` from a Box mesh `cube2Mesh`. This Box Geometry is flat upright box (because  $z=0.01f$ ).
2. Create a Material `cube2Mat` based on jME3's default `Unshaded.j3md` material definition.
3. Create a texture `cube2Tex` from the `Monkey.png` file in the `assets/Textures/ColoredTex/` directory of the project. This PNG file must have an alpha layer.
4. **Activate transparency in the material by setting the blend mode to Alpha.**
5. **Set the QueueBucket of the Geometry to `Bucket.Transparent`.**
6. Load the texture `cube2Tex` into the `ColorMap` layer of the material `cube2Mat`.
7. Apply the material to the cube, and attach the cube to the rootnode.

**Tip:** Learn more about creating PNG images with an alpha layer in the help system of your graphic editor.

# Shininess and Bumpiness

But textures are not all. Have a close look at the shiny sphere – you cannot get such a nice bumpy material with just a plain texture. You see that JME3 also supports so-called Phong-illuminated materials:

In a lit material, the standard texture layer is referred to as *DiffuseMap*, any material can use this layer. A lit material can additionally have lighting effects such as *Shininess* used together with the *SpecularMap* layer and *Specular* color. And you can even get a realistically bumpy or cracked surface with help of the *NormalMap* layer.

Let's have a look at the part of the code example where you create the shiny bumpy rock.

1. Create a Geometry from a Sphere shape. Note that this shape is a normal smooth sphere mesh.

```
Sphere sphereMesh = new Sphere(32, 32, 2f);
Geometry sphereGeo = new Geometry("Shiny rock", sphereMesh)
```

- i. (Only for Spheres) Change the sphere's TextureMode to make the square texture project better onto the sphere.

```
sphereMesh.setTextureMode(Sphere.TextureMode.Project);
```

- ii. You must generate TangentBinormals for the mesh so you can use the NormalMap layer of the texture.

```
TangentBinormalGenerator.generate(sphereMesh);
```

2. Create a material based on the `Lighting.j3md` default material.

```
Material sphereMat = new Material(assetManager,
 "Common/MatDefs/Light/Lighting.j3md");
```

- i. Set a standard rocky texture in the `DiffuseMap` layer.



```
sphereMat.setTexture("DiffuseMap",
 assetManager.loadTexture("Textures/Terrain/Pond/Pond
```

- ii. Set the `NormalMap` layer that contains the bumpiness. The `NormalMap` was

generated for this particular `DiffuseMap` with a special tool (e.g. Blender).

```
sphereMat.setTexture("NormalMap",
 assetManager.loadTexture("Textures/Terrain/Pond/Pond
```

- iii. Set the Material's Shininess to a value between 1 and 128. For a rock, a low fuzzy shininess is appropriate. Use material colors to define the shiny Specular color.

```
sphereMat.setBoolean("UseMaterialColors",true);
sphereMat.setColor("Diffuse",ColorRGBA.White); // minim
sphereMat.setColor("Specular",ColorRGBA.White); // for s
sphereMat.setFloat("Shininess", 64f); // [1,128] for shi
```

3. Assign your newly created material to the Geometry.

```
sphereGeo.setMaterial(sphereMat);
```

4. Let's move and rotate the geometry a bit to position it better.

```
sphereGeo.setLocalTranslation(0,2,-2); // Move it a bit
sphereGeo.rotate(1.6f, 0, 0); // Rotate it a bit
rootNode.attachChild(sphereGeo);
```

Remember that any `Lighting.j3md`-based material requires a light source, as shown in the full code sample above.

**Tip:** To deactivate Shininess, do not set `Shininess` to 0, but instead set the `specular` color to `ColorRGBA.Black`.

## Default Material Definitions

As you have seen, you can find the following default materials in `jme/core-data/Common/...`.

| Default Definition                 | Usage                                                                                               | Parameters                                                                                                |
|------------------------------------|-----------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------|
| Common/MatDefs/Misc/Unshaded.j3md  | Colored: Use with mat.setColor() and ColorRGBA.<br>Textured: Use with mat.setTexture() and Texture. | Color : Color<br>ColorMap : Texture2D                                                                     |
| Common/MatDefs/Light/Lighting.j3md | Use with shiny Textures, Bump- and NormalMaps textures. Requires a light source.                    | Ambient, Diffuse, Specular : Color<br>DiffuseMap, NormalMap, SpecularMap : Texture2D<br>Shininess : Float |

For a game, you create custom Materials based on these existing MaterialDefinitions – as you have just seen in the example with the shiny rock's material.

## Exercises

### Exercise 1: Custom .j3m Material

Look at the shiny rocky sphere above again. It takes several lines to create and set the Material.

- Note how it loads the `Lighting.j3md` Material definition.
- Note how it sets the `DiffuseMap` and `NormalMap` to a texture path.
- Note how it activates `useMaterialColors` and sets `Specular` and `Diffuse` to 4 float values (RGBA color).
- Note how it sets `shininess` to 64.

If you want to use one custom material for several models, you can store it in a `.j3m` file, and save a few lines of code every time.

You create a `j3m` file as follows:

1. Create a plain text file `assets/Materials/MyCustomMaterial.j3m` in your project directory, with the following content:

```

Material My shiny custom material : Common/MatDefs/Light/Lighti
 MaterialParameters {
 DiffuseMap : Textures/Terrain/Pond/Pond.jpg
 NormalMap : Textures/Terrain/Pond/Pond_normal.png
 UseMaterialColors : true
 Specular : 1.0 1.0 1.0 1.0
 Diffuse : 1.0 1.0 1.0 1.0
 Shininess : 64.0
 }
}

```

- Note that `Material` is a fixed keyword.
- Note that `My shiny custom material` is a String that you can choose to describe the material.
- Note how the code sets all the same properties as before!

2. In the code sample, comment out the eight lines that have `sphereMat` in them.
3. Below this line, add the following line:

```

sphereGeo.setMaterial((Material) assetManager.loadMaterial(
 "Materials/MyCustomMaterial.j3m"));

```

4. Run the app. The result is the same.

Using this new custom material `MyCustomMaterial.j3m` only takes one line. You have replaced the eight lines of an on-the-fly material definition with one line that loads a custom material from a file. Using .j3m files is very handy if you use the same material often.

## Exercise 2: Bumpiness and Shininess

Go back to the bumpy rock sample above:

1. Comment out the `DiffuseMap` line, and run the app. (Uncomment it again.)
  - Which property of the rock is lost?
2. Comment out the `NormalMap` line, and run the app. (Uncomment it again.)
  - Which property of the rock is lost?
3. Change the value of `Shininess` to values like 0, 63, 127.
  - What aspect of the Shininess changes?

## Conclusion

You have learned how to create a Material, specify its properties, and use it on a Geometry. You know how to load an image file (.png, .jpg) as texture into a material. You know to save texture files in a subfolder of your project's `assets/Textures/` directory.

You have also learned that a material can be stored in a `.j3m` file. The file references a built-in `MaterialDefinition` and specifies values for properties of that `MaterialDefinition`. You know to save your custom `.j3m` files in your project's `assets/Materials/` directory.

Now that you know how to load models and how to assign good-looking materials to them, let's have a look at how to animate models in the next chapter, [Hello Animation](#).

---

## See also

- [How to Use Materials](#)
- [Material Editing](#)
- [Materials forum thread](#)
- [jME3 Materials documentation \(PDF\)](#)
- [Video Tutorial: Editing and Assigning Materials to Models in jMonkeyEngine SDK \(from 2010, is there a newer one?\)](#)
- [Creating textures in Blender](#)
- [Various Material screenshots](#) (Not done with JME3, this is just to show the fantastic range of Material parameters in the hands of an expert, until we have a JME3 demo for it.)

[documentation](#), [beginner](#), [intro](#), [model](#), [material](#), [color](#), [texture](#), [transparency](#)

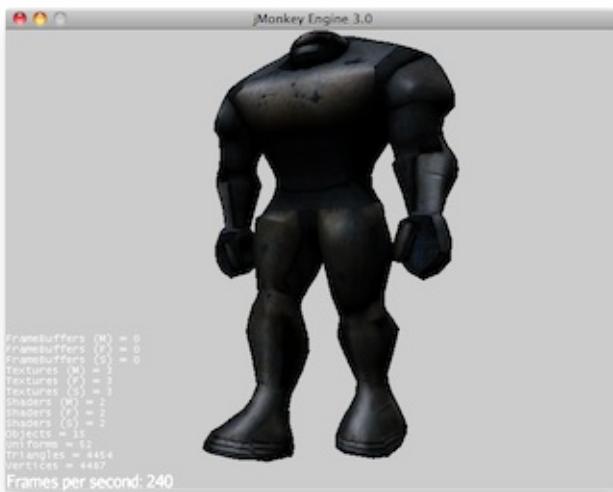
</div>

# title: jMonkeyEngine 3 Tutorial (7) - Hello Animation

## jMonkeyEngine 3 Tutorial (7) - Hello Animation

Previous: [Hello Material](#), Next: [Hello Picking](#)

This tutorial shows how to add an animation controller and channels, and how to respond to user input by triggering an animation in a loaded model.



To use the example assets in a new jMonkeyEngine SDK project, right-click your project, select “Properties”, go to “Libraries”, press “Add Library” and add the “jme3-test-data” library.  
  </div>

## Sample Code

```
package jme3test.helloworld;

import com.jme3.animation.AnimChannel;
import com.jme3.animation.AnimControl;
import com.jme3.animation.AnimEventListener;
import com.jme3.animation.LoopMode;
```

```
import com.jme3.app.SimpleApplication;
import com.jme3.input.KeyInput;
import com.jme3.input.controls.ActionListener;
import com.jme3.input.controls.KeyTrigger;
import com.jme3.light.DirectionalLight;
import com.jme3.math.ColorRGBA;
import com.jme3.math.Vector3f;
import com.jme3.scene.Node;

/** Sample 7 - how to load an OgreXML model and play an animation,
 * using channels, a controller, and an AnimEventListener. */
public class HelloAnimation extends SimpleApplication
 implements AnimEventListener {
 private AnimChannel channel;
 private AnimControl control;
 Node player;
 public static void main(String[] args) {
 HelloAnimation app = new HelloAnimation();
 app.start();
 }

 @Override
 public void simpleInitApp() {
 viewPort.setBackgroundColor(ColorRGBA.LightGray);
 initKeys();
 DirectionalLight dl = new DirectionalLight();
 dl.setDirection(new Vector3f(-0.1f, -1f, -1).normalizeLocal());
 rootNode.addLight(dl);
 player = (Node) assetManager.loadModel("Models/0to/0to.mesh.xml");
 player.setLocalScale(0.5f);
 rootNode.attachChild(player);
 control = player.getControl(AnimControl.class);
 control.addListener(this);
 channel = control.createChannel();
 channel.setAnim("stand");
 }

 public void onAnimCycleDone(AnimControl control, AnimChannel char
 if (animName.equals("Walk")) {
 channel.setAnim("stand", 0.50f);
```

```

 channel.setLoopMode(LoopMode.DontLoop);
 channel.setSpeed(1f);
 }

}

public void onAnimChange(AnimControl control, AnimChannel channel)
 // unused
}

/** Custom Keybinding: Map named actions to inputs. */
private void initKeys() {
 inputManager.addMapping("Walk", new KeyTrigger(KeyInput.KEY_SPACE));
 inputManager.addListener(actionListener, "Walk");
}
private ActionListener actionListener = new ActionListener() {
 public void onAction(String name, boolean keyPressed, float tpf) {
 if (name.equals("Walk") && !keyPressed) {
 if (!channel.getAnimationName().equals("Walk")) {
 channel.setAnim("Walk", 0.50f);
 channel.setLoopMode(LoopMode.Loop);
 }
 }
 }
};
}

```

## Creating and Loading Animated Models

You create animated models with a tool such as Blender. Take some time and learn how to create your own models in these [Blender Animation Tutorials](#). For now, download and use a free model, such as the one included here as an example ([Oto Golem](#), and [Ninja](#)).

Loading an animated model is pretty straight-forward, just as you have learned in the previous chapters. Animated Ogre models come as a set of files: The model is in `oto.mesh.xml`, and the animation details are in `oto.skeleton.xml`, plus the usual files for materials and textures. Check that all files of the model are together in the same `Model` subdirectory.

```

public void simpleInitApp() {
 /* Displaying the model requires a light source */
 DirectionalLight dl = new DirectionalLight();
 dl.setDirection(new Vector3f(-0.1f, -1f, -1).normalizeLocal());
 rootNode.addLight(dl);
 /* load and attach the model as usual */
 player = assetManager.loadModel("Models/Oto/Oto.mesh.xml");
 player.setLocalScale(0.5f); // resize
 rootNode.attachChild(player);
 ...
}

```

Don't forget to add a light source to make the material visible.

## Animation Controller and Channel

After you load the animated model, you register it to the Animation Controller.

- The controller object gives you access to the available animation sequences.
- The controller can have several channels, each channel can run one animation sequence at a time.
- To run several sequences, you create several channels, and set them each to their animation.

```

private AnimChannel channel;
private AnimControl control;

public void simpleInitApp() {
 ...
 /* Load the animation controls, listen to animation events,
 * create an animation channel, and bring the model in its defa
 control = player.getControl(AnimControl.class);
 control.addListener(this);
 channel = control.createChannel();
 channel.setAnim("stand");
 ...
}

```

In response to a question about animations on different channels interfering with each other, **Nehon** on the [jME forum](#) wrote,

“You have to consider channels as part of the skeleton that are animated. The default behavior is to use the whole skeleton for a channel. In your example the first channel plays the walk anim, then the second channel plays the dodge animation. Arms and feet are probably not affected by the doge animation so you can see the walk anim for them, but the rest of the body plays the dodge animation.

Usually multiple channels are used to animate different part of the body. For example you create one channel for the lower part of the body and one for the upper part. This allow you to play a walk animation with the lower part and for example a shoot animation with the upper part. This way your character can walk while shooting.

In your case, where you want animations to chain for the whole skeleton, you just have to use one channel.”

```
control = player.getControl(AnimControl.class);
```

This line of code will return NULL if the AnimeControl is not in the main node of your model. To check this, right click your model and click “Edit in SceneComposer” You can then see the tree for the model. You can then move everything to the main node. You can also access a subnode with the following code.

```
player.getChild("Subnode").getControl(AnimControl.class);
```

## Responding to Animation Events

Add `implements AnimEventListener` to the class declaration. This interface gives you access to events that notify you when a sequence is done, or when you change from one sequence to another, so you can respond to it. In this example, you reset the character to a standing position after a `walk` cycle is done.

```

public class HelloAnimation extends SimpleApplication
 implements AnimEventListener {

 ...

 public void onAnimCycleDone(AnimControl control,
 AnimChannel channel, String animName)
 if (animName.equals("Walk")) {
 channel.setAnim("stand", 0.50f);
 channel.setLoopMode(LoopMode.DontLoop);
 channel.setSpeed(1f);
 }
}

public void onAnimChange(AnimControl control, AnimChannel channel
// unused
}

```

## Trigger Animations After User Input

There are ambient animations like animals or trees that you may want to trigger in the main event loop. In other cases, animations are triggered by user interaction, such as key input. You want to play the Walk animation when the player presses a certain key (here the spacebar), at the same time as the avatar performs the walk action and changes its location.

1. Initialize a new input controller (in `simpleInitApp()`).
  - Write the `initKey()` convenience method and call it from `simpleInitApp()` .
2. Add a key mapping with the name as the action you want to trigger.
  - Here for example, you map `walk` to the Spacebar key.
3. Add an input listener for the `walk` action.

```

private void initKeys() {
 inputManager.addMapping("Walk", new KeyTrigger(KeyInput.KEY_SPACE));
 inputManager.addListener(actionListener, "Walk");
}

```

To use the input controller, you need to implement the `actionListener` by testing for each action by name, then set the channel to the corresponding animation to run.

- The second parameter of `setAnim()` is the `blendTime` (how long the current animation

should overlap with the last one).

- LoopMode can be Loop (repeat), Cycle (forward then backward), and DontLoop (only once).
- If needed, use channel.setSpeed() to set the speed of this animation.
- Optionally, use channel.setTime() to Fast-forward or rewind to a certain moment in time of this animation.

```
private ActionListener actionListener = new ActionListener() {
 public void onAction(String name, boolean keyPressed, float tpf
 if (name.equals("Walk") && !keyPressed) {
 if (!channel.getAnimationName().equals("Walk")){
 channel.setAnim("Walk", 0.50f);
 channel.setLoopMode(LoopMode.Cycle);
 }
 }
 };
};
```

## Exercises

### Exercise 1: Two Animations

Make a mouse click trigger another animation sequence!

1. Create a second channel in the controller
2. Create a new key trigger mapping and action (see: [Hello Input](#))
3. Tip: Do you want to find out what animation sequences are available in the model? Use:

```
for (String anim : control.getAnimationNames()) { System.out.pr
```

### Exercise 2: Revealing the Skeleton (1)

Open the `skeleton.xml` file in a text editor of your choice. You don't have to be able to read or write these xml files (Blender does that for you) – but it is good to know how skeletons work. “There's no magic to it!”

- Note how the bones are numbered and named. All names of animated models follow a naming scheme.
- Note the bone hierarchy that specifies how the bones are connected.

- Note the list of animations: Each animation has a name, and several tracks. Each track tells individual bones how and when to transform. These animation steps are called keyframes.

## Exercise 3: Revealing the Skeleton (2)

Add the following import statements for the SkeletonDebugger and Material classes:

```
import com.jme3.scene.debug.SkeletonDebugger;
import com.jme3.material.Material;
```

Add the following code snippet to `simpleInitApp()` to make the bones (that you just read about) visible!

```
SkeletonDebugger skeletonDebug =
 new SkeletonDebugger("skeleton", control.getSkeleton());
Material mat = new Material(assetManager, "Common/MatDefs/Misc/
mat.setColor("Color", ColorRGBA.Green);
mat.getAdditionalRenderState().setDepthTest(false);
skeletonDebug.setMaterial(mat);
player.attachChild(skeletonDebug);
```

Can you identify individual bones in the skeleton?

</div>

## Conclusion

Now you can load animated models, identify stored animations, and trigger animations by using `onAnimCycleDone()` and `onAnimChange()`. You also learned that you can play several animations simultaneously, by starting each in a channel of its own. This could be useful if you ever want to animate the lower and upper part of the characters body independently, for example the legs run, while the arms use a weapon.

Now that your character can walk, wouldn't it be cool if it could also pick up things, or aim a weapon at things, or open doors? Time to reveal the secrets of [mouse picking](#)!

---

See also: [Creating Animated OgreXML Models in Blender](#)

[beginner](#), [intro](#), [animation](#), [documentation](#), [keyinput](#), [input](#), [node](#), [model](#)

</div>

# title: jMonkeyEngine 3 Tutorial (8) - Hello Picking

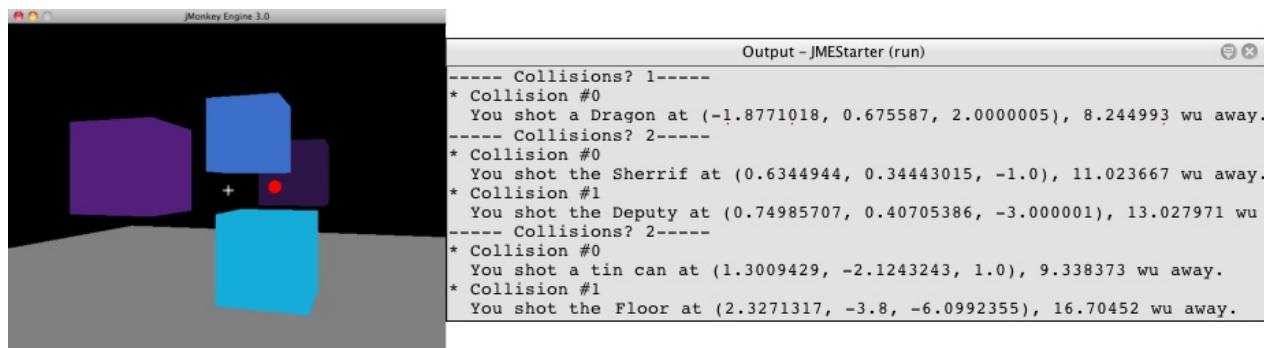
## jMonkeyEngine 3 Tutorial (8) - Hello Picking

Previous: [Hello Animation](#), Next: [Hello Collision](#)

Typical interactions in games include shooting, picking up objects, and opening doors. From an implementation point of view, these apparently different interactions are surprisingly similar: The user first aims and selects a target in the 3D scene, and then triggers an action on it. We call this process picking.

You can pick something by either pressing a key on the keyboard, or by clicking with the mouse. In either case, you identify the target by aiming a ray –a straight line– into the scene. This method to implement picking is called *ray casting* (which is not the same as *ray tracing*).

This tutorial relies on what you have learned in the [Hello Input](#) tutorial. You find more related code samples under [Mouse Picking](#) and [Collision and Intersection](#).



## Sample Code

```
package jme3test.helloworld;

import com.jme3.app.SimpleApplication;
import com.jme3.collision.CollisionResult;
import com.jme3.collision.CollisionResults;
```

```

import com.jme3.font.BitmapText;
import com.jme3.input.KeyInput;
import com.jme3.input.MouseInput;
import com.jme3.input.controls.ActionListener;
import com.jme3.input.controls.KeyTrigger;
import com.jme3.input.controls.MouseButtonTrigger;
import com.jme3.light.DirectionalLight;
import com.jme3.material.Material;
import com.jme3.math.ColorRGBA;
import com.jme3.math.Ray;
import com.jme3.math.Vector3f;
import com.jme3.scene.Geometry;
import com.jme3.scene.Node;
import com.jme3.scene.Spatial;
import com.jme3.scene.shape.Box;
import com.jme3.scene.shape.Sphere;

/** Sample 8 - how to let the user pick (select) objects in the sce
 * using the mouse or key presses. Can be used for shooting, openir
public class HelloPicking extends SimpleApplication {

 public static void main(String[] args) {
 HelloPicking app = new HelloPicking();
 app.start();
 }
 private Node shootables;
 private Geometry mark;

 @Override
 public void simpleInitApp() {
 initCrossHairs(); // a "+" in the middle of the screen to help
 initKeys(); // load custom key mappings
 initMark(); // a red sphere to mark the hit

 /** create four colored boxes and a floor to shoot at: */
 shootables = new Node("Shootables");
 rootNode.attachChild(shootables);
 shootables.attachChild(makeCube("a Dragon", -2f, 0f, 1f));
 shootables.attachChild(makeCube("a tin can", 1f, -2f, 0f));
 shootables.attachChild(makeCube("the Sheriff", 0f, 1f, -2f));
 }
}

```

```

 shootables.attachChild(makeCube("the Deputy", 1f, 0f, -4f));
 shootables.attachChild(makeFloor());
 shootables.attachChild(makeCharacter());
 }

/** Declaring the "Shoot" action and mapping to its triggers. */
private void initKeys() {
 inputManager.addMapping("Shoot",
 new KeyTrigger(KeyInput.KEY_SPACE), // trigger 1: spacebar
 new MouseButtonTrigger(MouseInput.BUTTON_LEFT)); // trigger 2
 inputManager.addListener(actionListener, "Shoot");
}

/** Defining the "Shoot" action: Determine what was hit and how to
 * handle it. */
private ActionListener actionListener = new ActionListener() {

 public void onAction(String name, boolean keyPressed, float tpf) {
 if (name.equals("Shoot") && !keyPressed) {
 // 1. Reset results list.
 CollisionResults results = new CollisionResults();
 // 2. Aim the ray from cam loc to cam direction.
 Ray ray = new Ray(cam.getLocation(), cam.getDirection());
 // 3. Collect intersections between Ray and Shootables in range.
 shootables.collideWith(ray, results);
 // 4. Print the results
 System.out.println("----- Collisions? " + results.size());
 for (int i = 0; i < results.size(); i++) {
 // For each hit, we know distance, impact point, name of
 float dist = results.getCollision(i).getDistance();
 Vector3f pt = results.getCollision(i).getContactPoint();
 String hit = results.getCollision(i).getGeometry().getName();
 System.out.println("* Collision #" + i);
 System.out.println(" You shot " + hit + " at " + pt + ",");
 }
 // 5. Use the results (we mark the hit object)
 if (results.size() > 0) {
 // The closest collision point is what was truly hit:
 CollisionResult closest = results.getClosestCollision();
 // Let's interact - we mark the hit with a red dot.
 mark.setLocalTranslation(closest.getContactPoint());
 rootNode.attachChild(mark);
 }
 }
 }
}

```

```

 } else {
 // No hits? Then remove the red mark.
 rootNode.detachChild(mark);
 }
 }
};

/** A cube object for target practice */
protected Geometry makeCube(String name, float x, float y, float
 Box box = new Box(1, 1, 1);
 Geometry cube = new Geometry(name, box);
 cube.setLocalTranslation(x, y, z);
 Material mat1 = new Material(assetManager, "Common/MatDefs/Misc
 mat1.setColor("Color", ColorRGBA.randomColor());
 cube.setMaterial(mat1);
 return cube;
}

/** A floor to show that the "shot" can go through several object
protected Geometry makeFloor() {
 Box box = new Box(15, .2f, 15);
 Geometry floor = new Geometry("the Floor", box);
 floor.setLocalTranslation(0, -4, -5);
 Material mat1 = new Material(assetManager, "Common/MatDefs/Misc
 mat1.setColor("Color", ColorRGBA.Gray);
 floor.setMaterial(mat1);
 return floor;
}

/** A red ball that marks the last spot that was "hit" by the "sh
protected void initMark() {
 Sphere sphere = new Sphere(30, 30, 0.2f);
 mark = new Geometry("BOOM!", sphere);
 Material mark_mat = new Material(assetManager, "Common/MatDefs/
 mark_mat.setColor("Color", ColorRGBA.Red);
 mark.setMaterial(mark_mat);
}

/** A centred plus sign to help the player aim. */

```

```

protected void initCrossHairs() {
 setDisplayStatView(false);
 guiFont = assetManager.loadFont("Interface/Fonts/Default.fnt");
 BitmapText ch = new BitmapText(guiFont, false);
 ch.setSize(guiFont.getCharSet().getRenderedSize() * 2);
 ch.setText("+"); // crosshairs
 ch.setLocalTranslation(// center
 settings.getWidth() / 2 - ch.getLineWidth()/2, settings.getHeight() / 2);
 guiNode.attachChild(ch);
}

protected Spatial makeCharacter() {
 // load a character from jme3test-test-data
 Spatial golem = assetManager.loadModel("Models/Oto/Oto.mesh.xml");
 golem.scale(0.5f);
 golem.setLocalTranslation(-1.0f, -1.5f, -0.6f);

 // We must add a light to make the model visible
 DirectionalLight sun = new DirectionalLight();
 sun.setDirection(new Vector3f(-0.1f, -0.7f, -1.0f));
 golem.addLight(sun);
 return golem;
}
}

```

You should see four colored cubes floating over a gray floor, and cross-hairs. Aim the cross-hairs and click, or press the spacebar to shoot. The hit spot is marked with a red dot.

Keep an eye on the application's output stream, it will give you more details: The name of the mesh that was hit, the coordinates of the hit, and the distance.

## Understanding the Helper Methods

The methods `makeCube()`, `makeFloor()`, `initMark()`, and `initCrossHairs`, are custom helper methods. We call them from `simpleInitApp()` to initialize the scenegraph with sample content.

1. `makeCube()` creates simple colored boxes for “target practice”.
2. `makeFloor()` creates a gray floor node for “target practice”.

3. `initMark()` creates a red sphere (“mark”). We will use it later to mark the spot that was hit.
  - Note that the mark is not attached and therefore not visible at the start!
4. `initCrossHairs()` creates simple cross-hairs by printing a “+” sign in the middle of the screen.
  - Note that the cross-hairs are attached to the `guiNode`, not to the `rootNode`.

In this example, we attached all “shootable” objects to one custom node, `Shootables`. This is an optimization so the engine only has to calculate intersections with objects we are actually interested in. The `shootables` node is attached to the `rootNode` as usual.

## Understanding Ray Casting for Hit Testing

Our goal is to determine which box the user “shot” (picked). In general, we want to determine which mesh the user has selected by aiming the cross-hairs at it. Mathematically, we draw a line from the camera and see whether it intersects with objects in the 3D scene. This line is called a ray.

Here is our simple ray casting algorithm for picking objects:

1. Reset the results list.
2. Cast a ray from cam location into the cam direction.
3. Collect all intersections between the ray and `Shootable` nodes in the `results` list.
4. Use the results list to determine what was hit:
  - i. For each hit, JME reports its distance from the camera, impact point, and the name of the mesh.
  - ii. Sort the results by distance.
  - iii. Take the closest result, it is the mesh that was hit.

## Implementing Hit Testing

### Loading the scene

First initialize some shootable nodes and attach them to the scene. You will use the `mark` object later.

```

Node shootables;
Geometry mark;

@Override
public void simpleInitApp() {
 initCrossHairs();
 initKeys();
 initMark();

 shootables = new Node("Shootables");
 rootNode.attachChild(shootables);
 shootables.attachChild(makeCube("a Dragon", -2f, 0f, 1f));
 shootables.attachChild(makeCube("a tin can", 1f, -2f, 0f));
 shootables.attachChild(makeCube("the Sheriff", 0f, 1f, -2f));
 shootables.attachChild(makeCube("the Deputy", 1f, 0f, -4));
 shootables.attachChild(makeFloor());
}


```



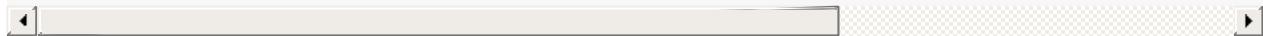
## Setting Up the Input Listener

Next you declare the shooting action. It can be triggered either by clicking, or by pressing the space bar. The `initKeys()` method is called from `simpleInitApp()` to set up these input mappings.

```

/** Declaring the "Shoot" action and its triggers. */
private void initKeys() {
 inputManager.addMapping("Shoot", // Declare...
 new KeyTrigger(KeyInput.KEY_SPACE), // trigger 1: spacebar, c
 new MouseButtonTrigger(MouseInput.BUTTON_LEFT)); // t
 inputManager.addListener(actionListener, "Shoot"); // ... and a
}


```



## Picking Action Using Crosshairs

Next we implement the ActionListener that responds to the Shoot trigger with an action. The action follows the ray casting algorithm described above:

1. For every click or press of the spacebar, the `Shoot` action is triggered.

2. The action casts a ray forward and determines intersections with shootable objects (= ray casting).
3. For any target that has been hit, it prints name, distance, and coordinates of the hit.
4. Finally it attaches a red mark to the closest result, to highlight the spot that was actually hit.
5. When nothing was hit, the results list is empty, and the red mark is removed.

Note how it prints a lot of output to show you which hits were registered.

```

/** Defining the "Shoot" action: Determine what was hit and how to
private ActionListener actionListener = new ActionListener() {
 @Override
 public void onAction(String name, boolean keyPressed, float tpf
 if (name.equals("Shoot") && !keyPressed) {
 // 1. Reset results list.
 CollisionResults results = new CollisionResults();
 // 2. Aim the ray from cam loc to cam direction.
 Ray ray = new Ray(cam.getLocation(), cam.getDirection());
 // 3. Collect intersections between Ray and Shootables in root
 shootables.collideWith(ray, results);
 // 4. Print results.
 System.out.println("----- Collisions? " + results.size() +
 for (int i = 0; i < results.size(); i++) {
 // For each hit, we know distance, impact point, name of
 float dist = results.getCollision(i).getDistance();
 Vector3f pt = results.getCollision(i).getContactPoint();
 String hit = results.getCollision(i).getGeometry().getName();
 System.out.println("* Collision #" + i);
 System.out.println(" You shot " + hit + " at " + pt + ",");
 }
 // 5. Use the results (we mark the hit object)
 if (results.size() > 0){
 // The closest collision point is what was truly hit:
 CollisionResult closest = results.getClosestCollision();
 mark.setLocalTranslation(closest.getContactPoint());
 // Let's interact - we mark the hit with a red dot.
 rootNode.attachChild(mark);
 } else {
 // No hits? Then remove the red mark.
 rootNode.detachChild(mark);
 }
 }
 }
};

```

**Tip:** Notice how you use the provided method

`results.getClosestCollision().getContactPoint()` to determine the *closest* hit's location. If your game includes a “weapon” or “spell” that can hit multiple targets, you could also loop

over the list of results, and interact with each of them.

## Picking Action Using Mouse Pointer

The above example assumes that the player is aiming crosshairs (attached to the center of the screen) at the target. But you can change the picking code to allow you to freely click at objects in the scene with a visible mouse pointer. In order to do this you have to convert the 2d screen coordinates of the click to 3D world coordinates to get the start point of the picking ray.

1. Reset result list.
2. Get 2D click coordinates.
3. Convert 2D screen coordinates to their 3D equivalent.
4. Aim the ray from the clicked 3D location forwards into the scene.
5. Collect intersections between ray and all nodes into a results list.

```
...
CollisionResults results = new CollisionResults();
Vector2f click2d = inputManager.getCursorPosition();
Vector3f click3d = cam.getWorldCoordinates(
 new Vector2f(click2d.x, click2d.y), 0f).clone();
Vector3f dir = cam.getWorldCoordinates(
 new Vector2f(click2d.x, click2d.y), 1f).subtractLocal(click3d);
Ray ray = new Ray(click3d, dir);
shootables.collideWith(ray, results);
...
```

Use this together with `inputManager.setCursorVisible(true)` to make certain the cursor is visible.

Note that since you now use the mouse for picking, you can no longer use it to rotate the camera. If you want to have a visible mouse pointer for picking in your game, you have to redefine the camera rotation mappings.

## Exercises

After a hit was registered, the closest object is identified as target, and marked with a red dot. Modify the code sample to solve these exercises:

## Exercise 1: Magic Spell

Change the color of the closest clicked target!

Here are some tips:

1. Go to the line where the closest target is identified, and add your changes after that.
2. To change an object's color, you must first know its Geometry. Identify the node by identifying the target's name.
  - Use `Geometry g = closest.getGeometry();`
3. Create a new color material and set the node's Material to this color.
  - Look inside the `makeCube()` method for an example of how to set random colors.

## Exercise 2: Shoot a Character

Shooting boxes isn't very exciting – can you add code that loads and positions a model in the scene, and shoot at it?

- Tip: You can use `Spatial golem = assetManager.loadModel("Models/Oto/Oto.mesh.xml");` from the engine's jme3-test-data.jar.
- Tip: Models are shaded! You need some light!

## Exercise 3: Pick up into Inventory

Change the code as follows to simulate the player picking up objects into the inventory:

When you click once, the closest target is identified and detached from the scene. When you click a second time, the target is reattached at the location that you have clicked. Here are some tips:

1. Create an inventory node to store the detached nodes temporarily.
2. The inventory node is not attached to the rootNode.
3. You can make the inventory visible by attaching the inventory node to the guiNode (which attaches it to the HUD). Note the following caveats:
  - If your nodes use a lit Material (not "Unshaded.j3md"), also add a light to the guiNode.
  - Size units are pixels in the HUD, therefor a 2-wu cube is displayed only 2 pixels wide in the HUD. – Scale it bigger!
  - Position the nodes: The bottom left corner of the HUD is (0f,0f), and the top right corner is at (`settings.getWidth(), settings.getHeight()`).

Link to user-proposed solutions: <http://jmonkeyengine.org/wiki/doku.php/jm3:solutions> Be sure to try to solve them for yourself first!

</div>

# Conclusion

You have learned how to use ray casting to solve the task of determining what object a user selected on the screen. You learned that this can be used for a variety of interactions, such as shooting, opening, picking up and dropping items, pressing a button or lever, etc.

Use your imagination from here:

- In your game, the click can trigger any action on the identified Geometry: Detach it and put it into the inventory, attach something to it, trigger an animation or effect, open a door or crate, – etc.
- In your game, you could replace the red mark with a particle emitter, add an explosion effect, play a sound, calculate the new score after each hit depending on what was hit – etc.

Now, wouldn't it be nice if those targets and the floor were solid objects and you could walk around between them? Let's continue to learn about [Collision Detection](#).

---

See also:

- [Hello Input](#)
- [Mouse Picking](#)
- [Collision and Intersection](#)

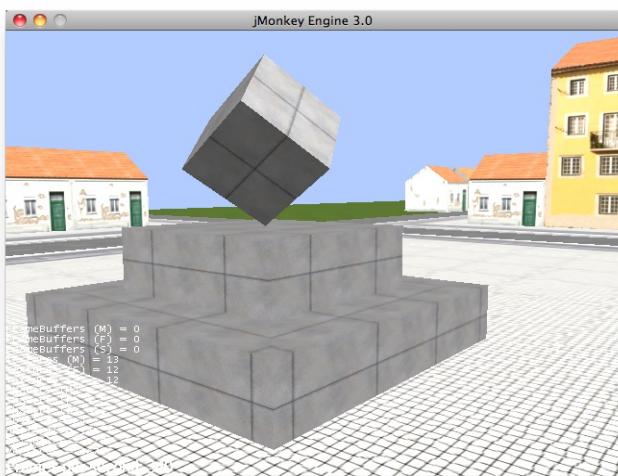
[beginner](#), [documentation](#), [intro](#), [node](#), [ray](#), [click](#), [collision](#), [keyinput](#), [input](#)

</div>

# jMonkeyEngine 3 Tutorial (9) - Hello Collision

Previous: [Hello Picking](#), Next: [Hello Terrain](#)

This tutorial demonstrates how you load a scene model and give it solid walls and floors for a character to walk around. You use a `RigidBodyControl` for the static collidable scene, and a `characterControl` for the mobile first-person character. You also learn how to set up the default first-person camera to work with physics-controlled navigation. You can use the solution shown here for first-person shooters, mazes, and similar games.



## Sample Code

If you don't have it yet, [download the town.zip sample scene](#).

```
jMonkeyProjects$ ls -1 BasicGame
assets/
build.xml
town.zip
src/
```

Place town.zip in the root directory of your JME3 project. Here is the code:

```
package jme3test.helloworld;
```

```

import com.jme3.app.SimpleApplication;
import com.jme3.asset.plugins.ZipLocator;
import com.jme3.bullet.BulletAppState;
import com.jme3.bullet.collision.shapes.CapsuleCollisionShape;
import com.jme3.bullet.collision.shapes.CollisionShape;
import com.jme3.bullet.control.CharacterControl;
import com.jme3.bullet.control.RigidBodyControl;
import com.jme3.bullet.util.CollisionShapeFactory;
import com.jme3.input.KeyInput;
import com.jme3.input.controls.ActionListener;
import com.jme3.input.controls.KeyTrigger;
import com.jme3.light.AmbientLight;
import com.jme3.lightDirectionalLight;
import com.jme3.math.ColorRGBA;
import com.jme3.math.Vector3f;
import com.jme3.scene.Node;
import com.jme3.scene.Spatial;

/**
 * Example 9 - How to make walls and floors solid.
 * This collision code uses Physics and a custom Action Listener.
 * @author normen, with edits by Zathras
 */
public class HelloCollision extends SimpleApplication
 implements ActionListener {

 private Spatial sceneModel;
 private BulletAppState bulletAppState;
 private RigidBodyControl landscape;
 private CharacterControl player;
 private Vector3f walkDirection = new Vector3f();
 private boolean left = false, right = false, up = false, down = f

 //Temporary vectors used on each frame.
 //They here to avoid instanciating new vectors on each frame
 private Vector3f camDir = new Vector3f();
 private Vector3f camLeft = new Vector3f();

 public static void main(String[] args) {
 HelloCollision app = new HelloCollision();

```

```
 app.start();
}

public void simpleInitApp() {
 /** Set up Physics */
 bulletAppState = new BulletAppState();
 stateManager.attach(bulletAppState);
 //bulletAppState.getPhysicsSpace().enableDebug(assetManager);

 // We re-use the flyby camera for rotation, while positioning it
 viewPort.setBackgroundColor(new ColorRGBA(0.7f, 0.8f, 1f, 1f));
 flyCam.setMoveSpeed(100);
 setUpKeys();
 setUpLight();

 // We load the scene from the zip file and adjust its size.
 assetManager.registerLocator("town.zip", ZipLocator.class);
 sceneModel = assetManager.loadModel("main.scene");
 sceneModel.setLocalScale(2f);

 // We set up collision detection for the scene by creating a
 // compound collision shape and a static RigidBodyControl with
 CollisionShape sceneShape =
 CollisionShapeFactory.createMeshShape((Node) sceneModel);
 landscape = new RigidBodyControl(sceneShape, 0);
 sceneModel.addControl(landscape);

 // We set up collision detection for the player by creating
 // a capsule collision shape and a CharacterControl.
 // The CharacterControl offers extra settings for
 // size, stepheight, jumping, falling, and gravity.
 // We also put the player in its starting position.
 CapsuleCollisionShape capsuleShape = new CapsuleCollisionShape(
 player = new CharacterControl(capsuleShape, 0.05f);
 player.setJumpSpeed(20);
 player.setFallSpeed(30);
 player.setGravity(30);
 player.setPhysicsLocation(new Vector3f(0, 10, 0));

 // We attach the scene and the player to the rootnode and the p
```

```

 // to make them appear in the game world.
 rootNode.attachChild(sceneModel);
 bulletAppState.getPhysicsSpace().add(landscape);
 bulletAppState.getPhysicsSpace().add(player);
 }

 private void setUpLight() {
 // We add light so we see the scene
 AmbientLight al = new AmbientLight();
 al.setColor(ColorRGBA.White.mult(1.3f));
 rootNode.addLight(al);

 DirectionalLight dl = new DirectionalLight();
 dl.setColor(ColorRGBA.White);
 dl.setDirection(new Vector3f(2.8f, -2.8f, -2.8f).normalizeLocal);
 rootNode.addLight(dl);
 }

 /**
 * We over-write some navigational key mappings here, so we can
 * add physics-controlled walking and jumping:
 */
 private void setUpKeys() {
 inputManager.addMapping("Left", new KeyTrigger(KeyInput.KEY_A));
 inputManager.addMapping("Right", new KeyTrigger(KeyInput.KEY_D));
 inputManager.addMapping("Up", new KeyTrigger(KeyInput.KEY_W));
 inputManager.addMapping("Down", new KeyTrigger(KeyInput.KEY_S));
 inputManager.addMapping("Jump", new KeyTrigger(KeyInput.KEY_SPACE));
 inputManager.addListener(this, "Left");
 inputManager.addListener(this, "Right");
 inputManager.addListener(this, "Up");
 inputManager.addListener(this, "Down");
 inputManager.addListener(this, "Jump");
 }

 /**
 * These are our custom actions triggered by key presses.
 * We do not walk yet, we just keep track of the direction the user
 * wants to move.
 */
 public void onAction(String binding, boolean isPressed, float tpf) {
 if (binding.equals("Left")) {
 left = isPressed;
 } else if (binding.equals("Right")) {
 right = isPressed;
 }
 }
}

```

```

 } else if (binding.equals("Up")) {
 up = isPressed;
 } else if (binding.equals("Down")) {
 down = isPressed;
 } else if (binding.equals("Jump")) {
 if (isPressed) { player.jump(); }
 }
 }

/**
 * This is the main event loop--walking happens here.
 * We check in which direction the player is walking by interpret
 * the camera direction forward (camDir) and to the side (camLeft)
 * The setWalkDirection() command is what lets a physics-controll
 * We also make sure here that the camera moves with player.
 */
@Override
public void simpleUpdate(float tpf) {
 camDir.set(cam.getDirection()).multLocal(0.6f);
 camLeft.set(cam.getLeft()).multLocal(0.4f);
 walkDirection.set(0, 0, 0);
 if (left) {
 walkDirection.addLocal(camLeft);
 }
 if (right) {
 walkDirection.addLocal(camLeft.negate());
 }
 if (up) {
 walkDirection.addLocal(camDir);
 }
 if (down) {
 walkDirection.addLocal(camDir.negate());
 }
 player.setWalkDirection(walkDirection);
 cam.setLocation(player.getPhysicsLocation());
}
}

```

Run the sample. You should see a town square with houses and a monument. Use the WASD keys and the mouse to navigate around with a first-person perspective. Run forward and jump by pressing W and Space. Note how you step over the sidewalk, and up the steps to the monument. You can walk in the alleys between the houses, but the walls are solid.

Don't walk over the edge of the world! 

## Understanding the Code

Let's start with the class declaration:

```
public class HelloCollision extends SimpleApplication
 implements ActionListener { ... }
```

You already know that `SimpleApplication` is the base class for all jME3 games. You make this class implement the `ActionListener` interface because you want to customize the navigational inputs later.

```
private Spatial sceneModel;
private BulletAppState bulletAppState;
private RigidBodyControl landscape;
private CharacterControl player;
private Vector3f walkDirection = new Vector3f();
private boolean left = false, right = false, up = false, down = f

//Temporary vectors used on each frame.
//They here to avoid instanciating new vectors on each frame
private Vector3f camDir = new Vector3f();
private Vector3f camLeft = new Vector3f();
```

You initialize a few private fields:

- The `BulletAppState` gives this `SimpleApplication` access to physics features (such as collision detection) supplied by jME3's `jBullet` integration
- The `Spatial sceneModel` is for loading an `OgreXML` model of a town.
- You need a `RigidBodyControl` to make the town model solid.
- The (invisible) first-person player is represented by a `CharacterControl` object.
- The fields `walkDirection` and the four Booleans are used for physics-controlled navigation.

- `camDir` and `camLeft` are temporary vectors used later when computing the walkingDirection from the cam position and rotation

Let's have a look at all the details:

## Initializing the Game

As usual, you initialize the game in the `simpleInitApp()` method.

```
viewPort.setBackgroundColor(new ColorRGBA(0.7f, 0.8f, 1f, 1f));
flyCam.setMoveSpeed(100);
setUpKeys();
setUpLight();
```

1. You set the background color to light blue, since this is a scene with a sky.
2. You repurpose the default camera control “flyCam” as first-person camera and set its speed.
3. The auxiliary method `setUpLights()` adds your light sources.
4. The auxiliary method `setUpKeys()` configures input mappings—we will look at it later.

## The Physics-Controlled Scene

The first thing you do in every physics game is create a `BulletAppState` object. It gives you access to jME3's jBullet integration which handles physical forces and collisions.

```
bulletAppState = new BulletAppState();
stateManager.attach(bulletAppState);
```

For the scene, you load the `sceneModel` from a zip file, and adjust the size.

```
assetManager.registerLocator("town.zip", ZipLocator.class);
sceneModel = assetManager.loadModel("main.scene");
sceneModel.setLocalScale(2f);
```

The file `town.zip` is included as a sample model in the JME3 sources – you can [download it here](#). (Optionally, use any OgreXML scene of your own.) For this sample, place the zip file in the application's top level directory (that is, next to `src/`, `assets/`, `build.xml`).

```

CollisionShape sceneShape =
 CollisionShapeFactory.createMeshShape((Node) sceneModel);
landscape = new RigidBodyControl(sceneShape, 0);
sceneModel.addControl(landscape);
rootNode.attachChild(sceneModel);

```

To use collision detection, you add a `RigidBodyControl` to the `sceneModel` `Spatial`. The `RigidBodyControl` for a complex model takes two arguments: A Collision Shape, and the object's mass.

- JME3 offers a `CollisionShapeFactory` that precalculates a mesh-accurate collision shape for a `Spatial`. You choose to generate a `CompoundCollisionShape` (which has `MeshCollisionShapes` as its children) because this type of collision shape is optimal for immobile objects, such as terrain, houses, and whole shooter levels.
- You set the mass to zero since a scene is static and its mass is irrelevant.
- Add the control to the `Spatial` to give it physical properties.
- As always, attach the `sceneModel` to the `rootNode` to make it visible.

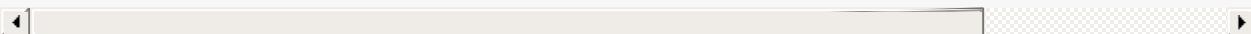
**Tip:** Remember to add a light source so you can see the scene.

## The Physics-Controlled Player

A first-person player is typically invisible. When you use the default `flyCam` as first-person cam, it does not even test for collisions and runs through walls. This is because the `flyCam` control does not have any physical shape assigned. In this code sample, you represent the first-person player as an (invisible) physical shape. You use the WASD keys to steer this physical shape around, while the physics engine manages for you how it walks along solid walls and on solid floors and jumps over solid obstacles. Then you simply make the camera follow the walking shape's location – and you get the illusion of being a physical body in a solid environment seeing through the camera.

So let's set up collision detection for the first-person player.

```
CapsuleCollisionShape capsuleShape = new CapsuleCollisionShape(
```



Again, you create a `CollisionShape`: This time you choose a `CapsuleCollisionShape`, a cylinder with a rounded top and bottom. This shape is optimal for a person: It's tall and the roundness helps to get stuck less often on obstacles.

- Supply the `CapsuleCollisionShape` constructor with the desired radius and height of the

bounding capsule to fit the shape of your character. In this example the character is 2\*1.5f units wide, and 6f units tall.

- The final integer argument specifies the orientation of the cylinder: 1 is the Y-axis, which fits an upright person. For animals which are longer than high you would use 0 or 2 (depending on how it is rotated).

```
player = new CharacterControl(capsuleShape, 0.05f);
```

“Does that CollisionShape make me look fat?” If you ever get confusing physics behaviour, remember to have a look at the collision shapes. Add the following line after the bulletAppState initialization to make the shapes visible:

```
bulletAppState.getPhysicsSpace().enableDebug(assetManager);
```

Now you use the CollisionShape to create a `CharacterControl` that represents the first-person player. The last argument of the `CharacterControl` constructor (here `.05f`) is the size of a step that the character should be able to surmount.

```
player.setJumpSpeed(20);
player.setFallSpeed(30);
player.setGravity(30);
```

Apart from step height and character size, the `CharacterControl` lets you configure jumping, falling, and gravity speeds. Adjust the values to fit your game situation.

```
player.setPhysicsLocation(new Vector3f(0, 10, 0));
```

Finally we put the player in its starting position and update its state – remember to use `setPhysicsLocation()` instead of `setLocalTranslation()` now, since you are dealing with a physical object.

</div>

## PhysicsSpace

Remember, in physical games, you must register all solid objects (usually the characters and the scene) to the PhysicsSpace!

```
bulletAppState.getPhysicsSpace().add(landscape);
bulletAppState.getPhysicsSpace().add(player);
```

The invisible body of the character just sits there on the physical floor. It cannot walk yet – you will deal with that next.

## Navigation

The default camera controller `cam` is a third-person camera. JME3 also offers a first-person controller, `flyCam`, which we use here to handle camera rotation. The `flyCam` control moves the camera using `setLocation()`.

However, you must redefine how walking (camera movement) is handled for physics-controlled objects: When you navigate a non-physical node (e.g. the default flyCam), you simply specify the *target location*. There are no tests that prevent the flyCam from getting stuck in a wall! When you move a PhysicsControl, you want to specify a *walk direction* instead. Then the PhysicsSpace can calculate for you how far the character can actually move in the desired direction – or whether an obstacle prevents it from going any further.

In short, you must re-define the flyCam's navigational key mappings to use `setWalkDirection()` instead of `setLocalTranslation()`. Here are the steps:

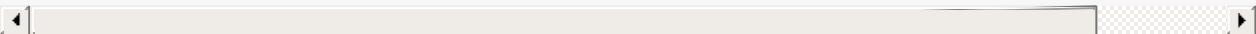
### 1. inputManager

In the `simpleInitApp()` method, you re-configure the familiar WASD inputs for walking, and Space for jumping.

```

private void setUpKeys() {
 inputManager.addMapping("Left", new KeyTrigger(KeyInput.KEY_A))
 inputManager.addMapping("Right", new KeyTrigger(KeyInput.KEY_D))
 inputManager.addMapping("Up", new KeyTrigger(KeyInput.KEY_W));
 inputManager.addMapping("Down", new KeyTrigger(KeyInput.KEY_S))
 inputManager.addMapping("Jump", new KeyTrigger(KeyInput.KEY_SPACE))
 inputManager.addListener(this, "Left");
 inputManager.addListener(this, "Right");
 inputManager.addListener(this, "Up");
 inputManager.addListener(this, "Down");
 inputManager.addListener(this, "Jump");
}

```



You can move this block of code into an auxiliary method `setUpKeys()` and call this method from `simpleInitApp()` – to keep the code more readable.

## 2. onAction()

Remember that this class implements the `ActionListener` interface, so you can customize the flyCam inputs. The `ActionListener` interface requires you to implement the `onAction()` method: You re-define the actions triggered by navigation key presses to work with physics.

```

public void onAction(String binding, boolean value, float tpf) {
 if (binding.equals("Left")) {
 if (value) { left = true; } else { left = false; }
 } else if (binding.equals("Right")) {
 if (value) { right = true; } else { right = false; }
 } else if (binding.equals("Up")) {
 if (value) { up = true; } else { up = false; }
 } else if (binding.equals("Down")) {
 if (value) { down = true; } else { down = false; }
 } else if (binding.equals("Jump")) {
 player.jump();
 }
}

```



The only movement that you do not have to implement yourself is the jumping action. The call `player.jump()` is a special method that handles a correct jumping motion for your `PhysicsCharacterNode`.

For all other directions: Every time the user presses one of the WASD keys, you *keep track* of the direction the user wants to go, by storing this info in four directional Booleans. No actual walking happens here yet. The update loop is what acts out the directional info stored in the booleans, and makes the player move, as shown in the next code snippet:

### 3. setWalkDirection()

Previously in the `onAction()` method, you have collected the info in which direction the user wants to go in terms of “forward” or “left”. In the update loop, you repeatedly poll the current rotation of the camera. You calculate the actual vectors to which “forward” or “left” corresponds in the coordinate system.

This last and most important code snippet goes into the `simpleUpdate()` method.

```
public void simpleUpdate(float tpf) {
 camDir.set(cam.getDirection()).multLocal(0.6f);
 camLeft.set(cam.getLeft()).multLocal(0.4f);
 walkDirection.set(0, 0, 0);
 if (left) {
 walkDirection.addLocal(camLeft);
 }
 if (right) {
 walkDirection.addLocal(camLeft.negate());
 }
 if (up) {
 walkDirection.addLocal(camDir);
 }
 if (down) {
 walkDirection.addLocal(camDir.negate());
 }
 player.setWalkDirection(walkDirection);
 cam.setLocation(player.getPhysicsLocation());
}
```

This is how the walking is triggered:

1. Initialize the vector `walkDirection` to zero. This is where you want to store the

- calculated walk direction.
2. Add to `walkDirection` the recent motion vectors that you polled from the camera. This way it is possible for a character to move forward and to the left simultaneously, for example!
  3. This one last line does the “walking magic”:

```
player.setWalkDirection(walkDirection);
```

Always use `setWalkDirection()` to make a physics-controlled object move continuously, and the physics engine handles collision detection for you.

4. Make the first-person camera object follow along with the physics-controlled player:

```
cam.setLocation(player.getPhysicsLocation());
```

**Important:** Again, do not use `setLocalTranslation()` to walk the player around. You will get it stuck by overlapping with another physical object. You can put the player in a start position with `setPhysicalLocation()` if you make sure to place it a bit above the floor and away from obstacles.

## Conclusion

You have learned how to load a “solid” physical scene model and walk around in it with a first-person perspective. You learned to speed up the physics calculations by using the `CollisionShapeFactory` to create efficient `CollisionShapes` for complex Geometries. You know how to add `PhysicsControls` to your collidable geometries and you register them to the `PhysicsSpace`. You also learned to use `player.setWalkDirection(walkDirection)` to move collision-aware characters around, and not `setLocalTranslation()`.

Terrains are another type of scene in which you will want to walk around. Let's proceed with learning [how to generate terrains](#) now.

---

Related info:

- How to load models and scenes: [Hello Asset](#), [Scene Explorer](#), [Scene Composer](#)
- [Terrain Collision](#)
- To learn more about complex physics scenes, where several mobile physical objects bump into each other, read [Hello Physics](#).
- FYI, there are simpler collision detection solutions without physics, too. Have a look at [jme3test.collision.TestTriangleCollision.java](#).

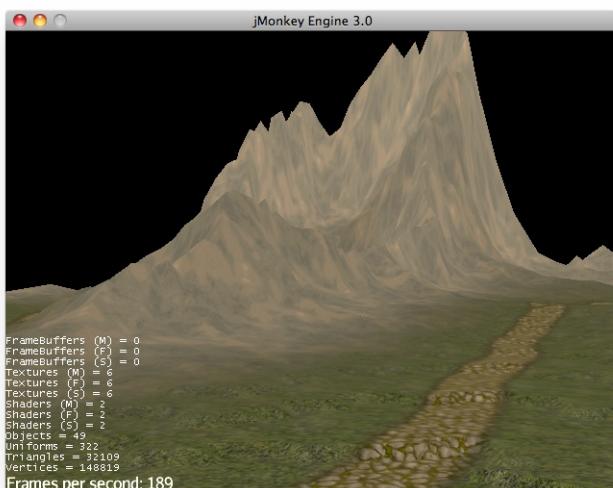
[beginner](#), [collision](#), [control](#), [intro](#), [documentation](#), [model](#), [physics](#)  
</div>

# title: jMonkeyEngine 3 Tutorial (10) - Hello Terrain

## jMonkeyEngine 3 Tutorial (10) - Hello Terrain

Previous: [Hello Collision](#), Next: [Hello Audio](#)

One way to create a 3D landscape is to sculpt a huge terrain model. This gives you a lot of artistic freedom – but rendering such a huge model can be quite slow. This tutorial explains how to create fast-rendering terrains from heightmaps, and how to use texture splatting to make the terrain look good.



Note: If you get an error when trying to create your `ImageBasedHeightMap` object, you may need to update the SDK, click on “Help” / “Check for updates”

To use the example assets in a new jMonkeyEngine SDK project, right-click your project, select “Properties”, go to “Libraries”, press “Add Library” and add the “jme3-test-data” library.  
`</div>`

## Sample Code

```
package jme3test.helloworld;
```

```
import com.jme3.app.SimpleApplication;
import com.jme3.material.Material;
import com.jme3.renderer.Camera;
import com.jme3.terrain.geomipmap.TerrainLodControl;
import com.jme3.terrain.heightmap.AbstractHeightMap;
import com.jme3.terrain.geomipmap.TerrainQuad;
import com.jme3.terrain.geomipmap.lodcalc.DistanceLodCalculator;
import com.jme3.terrain.heightmap.HillHeightMap; // for exercise 2
import com.jme3.terrain.heightmap.ImageBasedHeightMap;
import com.jme3.texture.Texture;
import com.jme3.texture.Texture.WrapMode;
import java.util.ArrayList;
import java.util.List;

/** Sample 10 - How to create fast-rendering terrains from heightmaps
 * and how to use texture splatting to make the terrain look good. */
public class HelloTerrain extends SimpleApplication {

 private TerrainQuad terrain;
 Material mat_terrain;

 public static void main(String[] args) {
 HelloTerrain app = new HelloTerrain();
 app.start();
 }

 @Override
 public void simpleInitApp() {
 flyCam.setMoveSpeed(50);

 /** 1. Create terrain material and load four textures into it.
 mat_terrain = new Material(assetManager,
 "Common/MatDefs/Terrain/Terrain.j3md");

 /** 1.1) Add ALPHA map (for red-blue-green coded splat textures)
 mat_terrain.setTexture("Alpha", assetManager.loadTexture(
 "Textures/Terrain/splat/alphamap.png"));

 /** 1.2) Add GRASS texture into the red layer (Tex1). */
 }
}
```

```

 Texture grass = assetManager.loadTexture(
 "Textures/Terrain/splat/grass.jpg");
 grass.setWrap(WrapMode.Repeat);
 mat_terrain.setTexture("Tex1", grass);
 mat_terrain.setFloat("Tex1Scale", 64f);

 /** 1.3) Add DIRT texture into the green layer (Tex2) */
 Texture dirt = assetManager.loadTexture(
 "Textures/Terrain/splat/dirt.jpg");
 dirt.setWrap(WrapMode.Repeat);
 mat_terrain.setTexture("Tex2", dirt);
 mat_terrain.setFloat("Tex2Scale", 32f);

 /** 1.4) Add ROAD texture into the blue layer (Tex3) */
 Texture rock = assetManager.loadTexture(
 "Textures/Terrain/splat/road.jpg");
 rock.setWrap(WrapMode.Repeat);
 mat_terrain.setTexture("Tex3", rock);
 mat_terrain.setFloat("Tex3Scale", 128f);

 /** 2. Create the height map */
 AbstractHeightMap heightmap = null;
 Texture heightMapImage = assetManager.loadTexture(
 "Textures/Terrain/splat/mountains512.png");
 heightmap = new ImageBasedHeightMap(heightMapImage.getImage());
 heightmap.load();

 /** 3. We have prepared material and heightmap.
 * Now we create the actual terrain:
 * 3.1) Create a TerrainQuad and name it "my terrain".
 * 3.2) A good value for terrain tiles is 64x64 -- so we supply
 * 3.3) We prepared a heightmap of size 512x512 -- so we supply
 * 3.4) As LOD step scale we supply Vector3f(1,1,1).
 * 3.5) We supply the prepared heightmap itself.
 */
 int patchSize = 65;
 terrain = new TerrainQuad("my terrain", patchSize, 513, heightmap);

 /** 4. We give the terrain its material, position & scale it, and
 terrain.setMaterial(mat_terrain);

```

```

 terrain.setLocalTranslation(0, -100, 0);
 terrain.setLocalScale(2f, 1f, 2f);
 rootNode.attachChild(terrain);

 /** 5. The LOD (level of detail) depends on were the camera is:
TerrainLodControl control = new TerrainLodControl(terrain, getO
 terrain.addControl(control);
 }
}

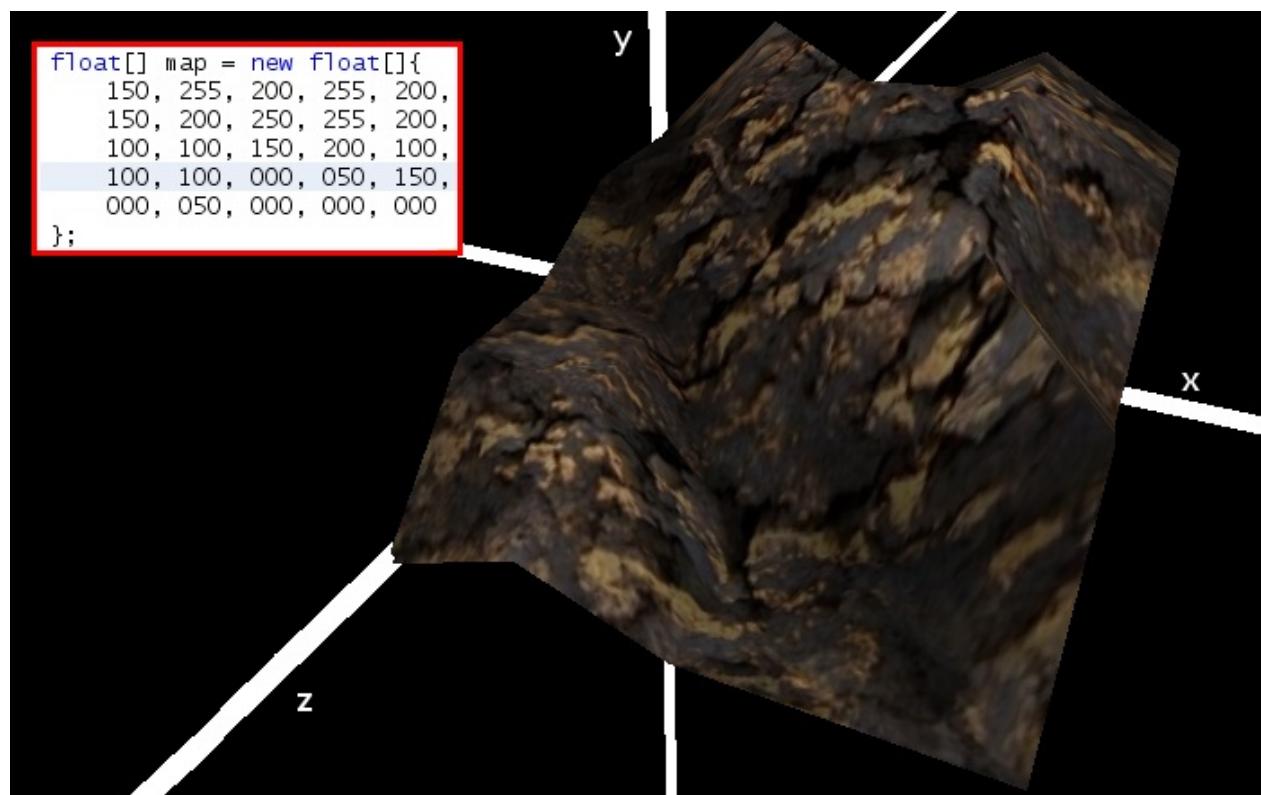
```

When you run this sample you should see a landscape with dirt mountains, grass plains, plus some winding roads in between.

## What is a Heightmap?

Heightmaps are an efficient way of representing the shape of a hilly landscape. Not every pixel of the landscape is stored, instead, a grid of sample values is used to outline the terrain height at certain points. The heights between the samples is interpolated.

In Java, a heightmap is a float array containing height values between 0f and 255f. Here is a very simple example of a terrain generated from a heightmap with  $5 \times 5 = 25$  height values.



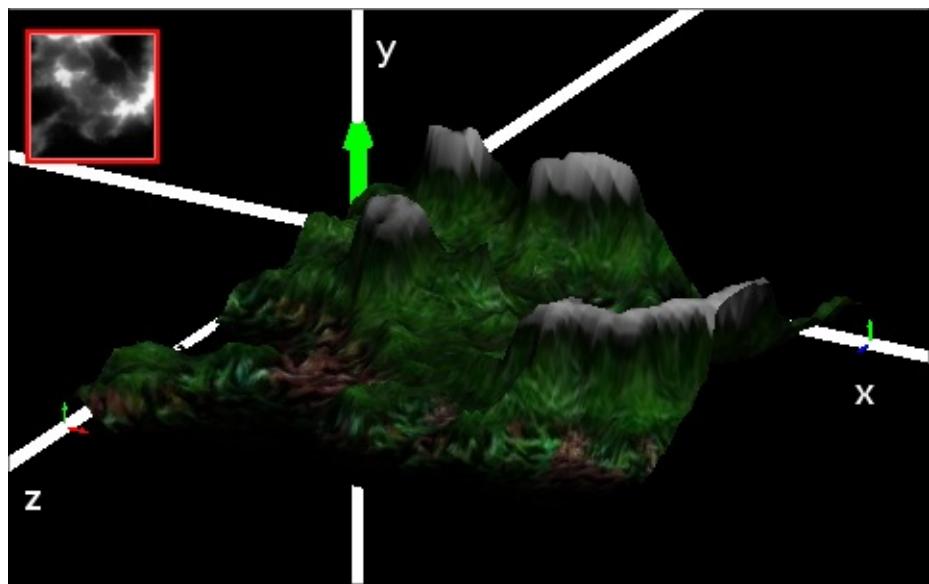
Important things to note:

- Low values (e.g. 0 or 50) are valesys.
- High values (e.g. 200, 255) are hills.
- The heightmap only specifies a few points, and the engine interpolates the rest.  
Interpolation is more efficient than creating a model with several millions vertices.

When looking at Java data types to hold an array of floats between 0 and 255, the Image class comes to mind. Storing a terrain's height values as a grayscale image has one big advantage: The outcome is a very userfriendly, like a topographical map:

- Low values (e.g. 0 or 50) are dark gray – these are valleys.
- High values (e.g. 200, 255) are light grays – these are hills.

Look at the next screenshot: In the top left you see a 128×128 grayscale image (heightmap) that was used as a base to generate the depicted terrain. To make the hilly shape better visible, the mountain tops are colored white, valleys brown, and the areas inbetween green:



In a real game, you will want to use more complex and smoother terrains than the simple heightmaps shown here. Heightmaps typically have square sizes of 512×512 or 1024×1024, and contain hundred thousands to 1 million height values. No matter which size, the concept is the same as described here.

## Looking at the Heightmap Code



The first step of terrain creation is the heightmap. You can create one yourself in any standard graphic application. Make sure it has the following properties:

- The size must be square, and a power of two.
  - Examples: 128×128, 256×256, 512×512, 1024×1024
- Color mode must be 255 grayscales.
  - Don't supply a color image, it will be interpreted as grayscale, with possibly weird results.
- Save the map as a .jpg or .png image file

The file `mountains512.png` that you see here is a typical example of an image heightmap.

Here is how you create the heightmap object in your jME code:

1. Create a Texture object.
2. Load your prepared heightmap image into the texture object.
3. Create an AbstractHeightMap object from an ImageBasedHeightMap.  
It requires an image from a JME Texture.
4. Load the heightmap.

```
AbstractHeightMap heightmap = null;
Texture heightMapImage = assetManager.loadTexture(
 "Textures/Terrain/splat/mountains512.png");
heightmap = new ImageBasedHeightMap(heightMapImage.getImage());
heightmap.load();
```

## What is Texture Splatting?

Previously you learned how to create a material for a simple shape such as a cube. All sides of the cube have the same color. You can apply the same material to a terrain, but then you have one big meadow, one big rock desert, etc. This is not always what you want.

Texture splatting allows you create a custom material, and “paint” textures on it like with a “paint brush”. This is very useful for terrains: As you see in the example here, you can paint a grass texture into the valleys, a dirt texture onto the mountains, and free-form roads inbetween.

The jMonkeyEngine SDK comes with a [TerrainEditor plugin](#). Using the TerrainEditor plugin, you can sculpt the terrain with the mouse, and save the result as heightmap. You can paint textures on the terrain and the plugin saves the resulting splat textures as alphamap(s). The following paragraphs describe the manual process for you. You can choose to create the

terrain by hand, or using the [TerrainEditor plugin](#).

Splat textures are based on the `Terrain.j3md` material definition. If you open the `Terrain.j3md` file, and look in the Material Parameters section, you see that you have several texture layers to paint on: `Tex1`, `Tex2`, `Tex3`, etc.

Before you can start painting, you have to make a few decisions:

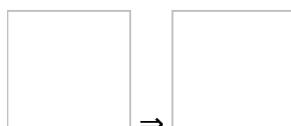


1. Choose three textures. For example grass.jpg, dirt.jpg, and road.jpg.

2. You “paint” three texture layers by using three colors: Red, blue and, green. You arbitrarily decide that...
  - i. Red is grass – red is layer `Tex1`, so put the grass texture into Tex1.
  - ii. Green is dirt – green is layer `Tex2`, so put the dirt texture into Tex2.
  - iii. Blue is roads – blue is layer `Tex3`, so put the roads texture into Tex3.

Now you start painting the texture:

1. Make a copy of your terrains heightmap, `mountains512.png`. You want it as a reference for the shape of the landscape.
2. Name the copy `alphamap.png`.
3. Open `alphamap.png` in a graphic editor and switch the image mode to color image.
  - i. Paint the black valleys red – this will be the grass.
  - ii. Paint the white hills green – this will be the dirt of the mountains.
  - iii. Paint blue lines where you want roads to criss-cross the landscape.
4. The end result should look similar to this:

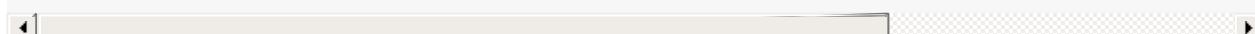


</div>

## Looking at the Texturing Code

As usual, you create a Material object. Base it on the Material Definition `Terrain.j3md` that is included in the jME3 framework.

```
Material mat_terrain = new Material(assetManager, "Common/MatDefs/Terrain/StandardTerrain.j3md")
```



Load four textures into this material. The first one, `Alpha`, is the alphamap that you just created.

```
mat_terrain.setTexture("Alpha",
 assetManager.loadTexture("Textures/Terrain/splat/alphamap.png")
```

The three other textures are the layers that you have previously decided to paint: grass, dirt, and road. You create texture objects and load the three textures as usual. Note how you assign them to their respective texture layers (Tex1, Tex2, and Tex3) inside the Material!

```
/** 1.2) Add GRASS texture into the red layer (Tex1). */
Texture grass = assetManager.loadTexture(
 "Textures/Terrain/splat/grass.jpg");
grass.setWrap(WrapMode.Repeat);
mat_terrain.setTexture("Tex1", grass);
mat_terrain.setFloat("Tex1Scale", 64f);

/** 1.3) Add DIRT texture into the green layer (Tex2) */
Texture dirt = assetManager.loadTexture(
 "Textures/Terrain/splat/dirt.jpg");
dirt.setWrap(WrapMode.Repeat);
mat_terrain.setTexture("Tex2", dirt);
mat_terrain.setFloat("Tex2Scale", 32f);

/** 1.4) Add ROAD texture into the blue layer (Tex3) */
Texture rock = assetManager.loadTexture(
 "Textures/Terrain/splat/road.jpg");
rock.setWrap(WrapMode.Repeat);
mat_terrain.setTexture("Tex3", rock);
mat_terrain.setFloat("Tex3Scale", 128f);
```

The individual texture scales (e.g. `mat_terrain.setFloat("Tex3Scale", 128f);`) depend on the size of the textures you use.

- You can tell you picked too small a scale if, for example, your road tiles appear like tiny grains of sand.
- You can tell you picked too big a scale if, for example, the blades of grass look like twigs.

Use `setWrap(WrapMode.Repeat)` to make the small texture fill the wide area. If the repetition is too visible, try adjusting the respective `Tex*Scale` value.

## What is a Terrain?

Internally, the generated terrain mesh is broken down into tiles and blocks. This is an optimization to make culling easier. You do not need to worry about “tiles and blocks” too much, just use recommended values for now – 64 is a good start.

Let's assume you want to generate a 512×512 terrain. You already have created the heightmap object. Here are the steps that you perform everytime you create a new terrain.

Create a `TerrainQuad` with the following arguments:

1. Specify a name: E.g. `my terrain`.
2. Specify tile size: You want to terrain tiles of size 64×64, so you supply  $64+1 = 65$ .
  - In general, 64 is a good starting value for terrain tiles.
3. Specify block size: Since you prepared a heightmap of size 512×512, you supply  $512+1 = 513$ .
  - If you supply a block size of 2x the heightmap size ( $1024+1=1025$ ), you get a stretched out, wider, flatter terrain.
  - If you supply a block size 1/2 the heightmap size ( $256+1=257$ ), you get a smaller, more detailed terrain.
4. Supply the 512×512 heightmap object that you created.

## Looking at the Terrain Code

Here's the code:

```
terrain = new TerrainQuad(
 "my terrain", // name
 65, // tile size
 513, // block size
 heightmap.getHeightMap()); // heightmap
```

You have created the terrain object.

1. Remember to apply the created material:

```
terrain.setMaterial(mat_terrain);
```

2. Remember to attach the terrain to the rootNode.

```
rootNode.attachChild(terrain);
```

3. If needed, scale and translate the terrain object, just like any other Spatial.

**Tip:** Terrain.j3md is an unshaded material definition, so you do not need a light source. You can also use TerrainLighting.j3md plus a light, if you want a shaded terrain.

## What is LOD (Level of Detail)?

JME3 includes an optimization that adjusts the level of detail (LOD) of the rendered terrain depending on how close or far the camera is.

```
TerrainLodControl control = new TerrainLodControl(terrain, getControl());
terrain.addControl(control);
```

Close parts of the terrain are rendered in full detail. Terrain parts that are further away are not clearly visible anyway, and JME3 improves performance by rendering them less detailed. This way you can afford to load huge terrains with no penalty caused by invisible details.

## Exercises

### Exercise 1: Texture Layers

What happens when you swap two layers, for example `Tex1` and `Tex2` ?

```
...
mat_terrain.setTexture("Tex2", grass);
...
mat_terrain.setTexture("Tex1", dirt);
```

You see it's easier to swap layers in the code, than to change the colors in the alphamap.

### Exercise 2: Randomized Terrains

The following three lines generate the heightmap object based on your user-defined image:

```

AbstractHeightMap heightmap = null;
Texture heightMapImage = assetManager.loadTexture(
 "Textures/Terrain/splat/mountains512.png");
heightmap = new ImageBasedHeightMap(heightMapImage.getImage());

```

Instead, you can also let JME3 generate a random landscape for you:

1. What result do you get when you replace the above three heightmap lines by the following lines and run the sample?

```

HillHeightMap heightmap = null;
HillHeightMap.NORMALIZE_RANGE = 100; // optional
try {
 heightmap = new HillHeightMap(513, 1000, 50, 100, (byte) 3)
} catch (Exception ex) {
 ex.printStackTrace();
}

```

2. Change one parameter at a time, and then run the sample again. Note the differences. Can you find out which of the values has which effect on the generated terrain (look at the javadoc also)?

- Which value controls the size?
  - What happens if the size is not a square number +1 ?
- Which value controls the number of hills generated?
- Which values control the size and steepness of the hills?
  - What happens if the min is bigger than or equal to max?
  - What happens if both min and max are small values (e.g. 10/20)?
  - What happens if both min and max are large values (e.g. 1000/1500)?
  - What happens if min and max are very close(e.g. 1000/1001, 20/21)? Very far apart (e.g. 10/1000)?

You see the variety of hilly landscapes that can be generated using this method.

For this exercise, you can keep using the splat Material from the sample code above. Just don't be surprised that the Material does not match the shape of the newly randomized landscape. If you want to generate real matching splat textures for randomized heightmaps, you need to write a custom method that, for example, creates an alphamap from the heightmap by replacing certain grayscales with certain RGB values.

</div>

## Exercise 3: Solid Terrains

Can you combine what you learned here and in [Hello Collision](#), and [make the terrain solid](#)?

## Conclusion

You have learned how to create terrains that are more efficient than loading one giant model. You know how to generate random or create handmade heightmaps. You can add a LOD control to render large terrains faster. You are aware that you can combine what you learned about collision detection to make the terrain solid to a physical player. You are also able to texture a terrain “like a boss” using layered Materials and texture splatting. You are aware that the jMonkeyEngine SDK provides a TerrainEditor that helps with most of these manual tasks.

Do you want to hear your players say “ouch!” when they bump into a wall or fall off a hill? Continue with learning [how to add sound](#) to your game.

---

See also:

- [Terrain Collision](#)

[beginner](#), [heightmap](#), [documentation](#), [terrain](#), [texture](#)

</div>

# title: jMonkeyEngine 3 Tutorial (11) - Hello Audio

## jMonkeyEngine 3 Tutorial (11) - Hello Audio

Previous: [Hello Terrain](#), Next: [Hello Effects](#)

This tutorial explains how to add 3D sound to a game, and how to make sounds play together with events, such as clicking. You learn how to use an Audio Listener and Audio Nodes. You also make use of an Action Listener and a MouseButtonTrigger from the previous [Hello Input](#) tutorial to make a mouse click trigger a gun shot sound.

To use the example assets in a new jMonkeyEngine SDK project, right-click your project, select “Properties”, go to “Libraries”, press “Add Library” and add the “jme3-test-data” library.  
</div>

## Sample Code

```
package jme3test.helloworld;

import com.jme3.app.SimpleApplication;
import com.jme3.audio.AudioNode;
import com.jme3.input.MouseInput;
import com.jme3.input.controls.ActionListener;
import com.jme3.input.controls.MouseButtonTrigger;
import com.jme3.material.Material;
import com.jme3.math.ColorRGBA;
import com.jme3.scene.Geometry;
import com.jme3.scene.shape.Box;

/** Sample 11 - playing 3D audio. */
public class HelloAudio extends SimpleApplication {

 private AudioNode audio_gun;
```

```
private AudioNode audio_nature;
private Geometry player;

public static void main(String[] args) {
 HelloAudio app = new HelloAudio();
 app.start();
}

@Override
public void simpleInitApp() {
 flyCam.setMoveSpeed(40);

 /** just a blue box floating in space */
 Box box1 = new Box(1, 1, 1);
 player = new Geometry("Player", box1);
 Material mat1 = new Material(assetManager, "Common/MatDefs/Misc/
mat1.setColor("Color", ColorRGBA.Blue);
player.setMaterial(mat1);
rootNode.attachChild(player);

/** custom init methods, see below */
initKeys();
initAudio();
}

/** We create two audio nodes. */
private void initAudio() {
 /* gun shot sound is to be triggered by a mouse click. */
 audio_gun = new AudioNode(assetManager, "Sound/Effects/Gun.wav");
 audio_gun.setPositional(false);
 audio_gun.setLooping(false);
 audio_gun.setVolume(2);
 rootNode.attachChild(audio_gun);

 /* nature sound - keeps playing in a loop. */
 audio_nature = new AudioNode(assetManager, "Sound/Environment/C
audio_nature.setLooping(true); // activate continuous playing
audio_nature.setPositional(true);
audio_nature.setVolume(3);
rootNode.attachChild(audio_nature);
```

```

 audio_nature.play(); // play continuously!
 }

/** Declaring "Shoot" action, mapping it to a trigger (mouse left
private void initKeys() {
 inputManager.addMapping("Shoot", new MouseButtonTrigger(MouseIr
 inputManager.addListener(actionListener, "Shoot");
}

/** Defining the "Shoot" action: Play a gun sound. */
private ActionListener actionListener = new ActionListener() {
 @Override
 public void onAction(String name, boolean keyPressed, float tpf
 if (name.equals("Shoot") && !keyPressed) {
 audio_gun.playInstance(); // play each instance once!
 }
 }
};

/** Move the listener with the a camera - for 3D audio. */
@Override
public void simpleUpdate(float tpf) {
 listener.setLocation(cam.getLocation());
 listener.setRotation(cam.getRotation());
}

}

```

When you run the sample, you should see a blue cube. You should hear a nature-like ambient sound. When you click, you hear a loud shot.

## Understanding the Code Sample

In the `initSimpleApp()` method, you create a simple blue cube geometry called `player` and attach it to the scene – this is just arbitrary sample content, so you see something when running the audio sample.

Let's have a closer look at `initAudio()` to learn how to use `AudioNode`s.

# AudioNodes

Adding sound to your game is quite simple: Save your audio files into your `assets/Sound` directory. JME3 supports both Ogg Vorbis (.ogg) and Wave (.wav) file formats.

For each sound, you create an AudioNode. You can use an AudioNode like any node in the JME scene graph, e.g. attach it to other Nodes. You create one node for a gunshot sound, and one node for a nature sound.

```
private AudioNode audio_gun;
private AudioNode audio_nature;
```

Look at the custom `initAudio()` method: Here you initialize the sound objects and set their parameters.

```
audio_gun = new AudioNode(assetManager, "Sound/Effects/Gun.wav", false);
...
audio_nature = new AudioNode(assetManager, "Sound/Environment/Natur
```

These two lines create new sound nodes from the given audio files in the AssetManager. The `false` flag means that you want to buffer these sounds before playing. (If you set this flag to true, the sound will be streamed, which makes sense for really long sounds.)

You want the gunshot sound to play *once* (you don't want it to loop). You also specify its volume as gain factor (at 0, sound is muted, at 2, it is twice as loud, etc.).

```
audio_gun.setPositional(false);
audio_gun.setLooping(false);
audio_gun.setVolume(2);
rootNode.attachChild(audio_gun);
```

Note that `setPositional(false)` is pretty important when you use stereo sounds. Positional sounds must always be mono audio files, otherwise the engine will remind it to you with a crash.

The nature sound is different: You want it to loop *continuously* as background sound. This is why you set looping to true, and immediately call the `play()` method on the node. You also choose to set its volume to 3.

```

 audio_nature.setLooping(true); // activate continuous playing
 ...
 audio_nature.setVolume(3);
 rootNode.attachChild(audio_nature);
 audio_nature.play(); // play continuously!
 }

```

Here you make `audio_nature` a positional sound that comes from a certain place. For that you give the node an explicit translation, in this example, you choose `Vector3f.ZERO` (which stands for the coordinates `0.0f, 0.0f, 0.0f`, the center of the scene.) Since jME supports 3D audio, you are now able to hear this sound coming from this particular location. Making the sound positional is optional. If you don't use these lines, the ambient sound comes from every direction.

```

 ...
 audio_nature.setPositional(true);
 audio_nature.setLocalTranslation(Vector3f.ZERO.clone());
 ...

```

**Tip:** Attach AudioNodes into the scene graph like all nodes, to make certain moving nodes stay up-to-date. If you don't attach them, they are still audible and you don't get an error message but 3D sound will not work as expected. AudioNodes can be attached directly to the root node or they can be attached inside a node that is moving through the scene and both the AudioNode and the 3d position of the sound it is generating will move accordingly.

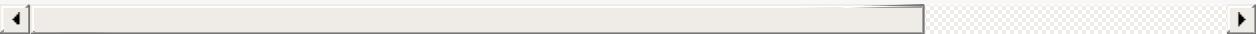
**Tip:** `playInstance` always plays the sound from the position of the AudioNode so multiple gunshots from one gun (for example) can be generated this way, however if multiple guns are firing at once then an AudioNode is needed for each one.

</div>

## Triggering Sound

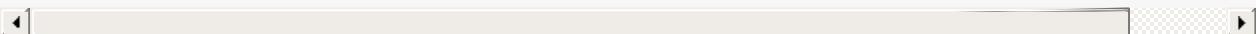
Let's have a closer look at `initKeys()` : As you learned in previous tutorials, you use the `inputManager` to respond to user input. Here you add a mapping for a left mouse button click, and name this new action `Shoot`.

```
/** Declaring "Shoot" action, mapping it to a trigger (mouse left
private void initKeys() {
 inputManager.addMapping("Shoot", new MouseButtonTrigger(MouseInput.BUTTON_LEFT));
 inputManager.addListener(actionListener, "Shoot");
}
```



Setting up the ActionListener should also be familiar from previous tutorials. You declare that, when the trigger (the mouse button) is pressed and released, you want to play a gun sound.

```
/** Defining the "Shoot" action: Play a gun sound. */
private ActionListener actionListener = new ActionListener() {
 @Override
 public void onAction(String name, boolean keyPressed, float tpf) {
 if (name.equals("Shoot") && !keyPressed) {
 audio_gun.playInstance(); // play each instance once!
 }
 }
};
```



Since you want to be able to shoot fast repeatedly, so you do not want to wait for the previous gunshot sound to end before the next one can start. This is why you play this sound using the `playInstance()` method. This means that every click starts a new instance of the sound, so two instances can overlap. You set this sound not to loop, so each instance only plays once. As you would expect it of a gunshot.

## Ambient or Situational?

The two sounds are two different use cases:

- A gunshot is situational. You want to play it only once, right when it is triggered.
  - This is why you `setLooping(false)` .
- The nature sound is an ambient, background noise. You want it to start playing from the start, as long as the game runs.
  - This is why you `setLooping(true)` .

Now every sound knows whether it should loop or not.

Apart from the looping boolean, another difference is where `play().playInstance()` is called on those nodes:

- You start playing the background nature sound right after you have created it, in the `initAudio()` method.

```
audio_nature.play(); // play continuously!
```

- The gunshot sound, however, is triggered situationally, once, only as part of the `shoot` input action that you defined in the `ActionListener`.

```
/** Defining the "Shoot" action: Play a gun sound. */
private ActionListener actionListener = new ActionListener()
 @Override
 public void onAction(String name, boolean keyPressed, float
 if (name.equals("Shoot") && !keyPressed) {
 audio_gun.playInstance(); // play each instance once!
 }
 }
};
```

## Buffered or Streaming?

The Boolean in the `AudioNode` constructor defines whether the audio is buffered (false) or streamed (true). For example:

```
audio_gunshot = new AudioNode(assetManager, "Sound/Effects/Gun.wav"
...
audio_nature = new AudioNode(assetManager, "Sound/Environment/Natur
```

Typically, you stream long sounds, and buffer short sounds.

Note that streamed sounds can not loop (i.e. `setLooping` will not work as you expect). Check the `getStatus` on the node and if it has stopped recreate the node.

## Play() or PlayInstance()?

| <code>audio.play()</code>                          | <code>audio.playInstance()</code>                    |
|----------------------------------------------------|------------------------------------------------------|
| Plays buffered sounds.                             | Plays buffered sounds.                               |
| Plays streamed sounds.                             | Cannot play streamed sounds.                         |
| The same sound cannot play twice at the same time. | The same sounds can play multiple times and overlap. |

## Your Ear in the Scene

To create a 3D audio effect, JME3 needs to know the position of the sound source, and the position of the ears of the player. The ears are represented by an 3D Audio Listener object. The `listener` object is a default object in a SimpleApplication.

In order to make the most of the 3D audio effect, you must use the `simpleUpdate()` method to move and rotate the listener (the player's ears) together with the camera (the player's eyes).

```
public void simpleUpdate(float tpf) {
 listener.setLocation(cam.getLocation());
 listener.setRotation(cam.getRotation());
}
```

If you don't do that, the results of 3D audio will be quite random.

## Global, Directional, Positional?

In this example, you defined the nature sound as coming from a certain position, but not the gunshot sound. This means your gunshot is global and can be heard everywhere with the same volume. JME3 also supports directional sounds which you can only hear from a certain direction.

It makes equal sense to make the gunshot positional, and let the ambient sound come from every direction. How do you decide which type of 3D sound to use from case to case?

- In a game with moving enemies you may want to make the gun shot or footsteps positional sounds. In these cases you must move the AudioNode to the location of the enemy before `playInstance()` ing it. This way a player with stereo speakers hears from which direction the enemy is coming.
- Similarly, you may have game levels where you want one background sound to play globally. In this case, you would make the AudioNode neither positional nor directional

(set both to false).

- If you want sound to be “absorbed by the walls” and only broadcast in one direction, you would make this AudioNode directional. This tutorial does not discuss directional sounds, you can read about [Advanced Audio](#) here.

In short, you must choose in every situation whether it makes sense for a sound to be global, directional, or positional.

## Conclusion

You now know how to add the two most common types of sound to your game: Global sounds and positional sounds. You can play sounds in two ways: Either continuously in a loop, or situationally just once. You know the difference between buffering short sounds and streaming long sounds. You know the difference between playing overlapping sound instances, and playing unique sounds that cannot overlap with themselves. You also learned to use sound files that are in either .ogg or .wav format.

**Tip:** JME's Audio implementation also supports more advanced effects such as reverberation and Doppler effect. Use these “pro” features to make audio sound different depending on whether it's in the hallway, in a cave, outdoors, or in a carpeted room. Find out more about environmental effects from the sample code included in the jme3test directory and from the advanced [Audio](#) docs.

Want some fire and explosions to go with your sounds? Read on to learn more about [effects](#).

---

See also:

- [Audio](#)

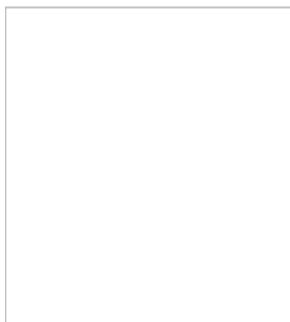
[sound](#), [documentation](#), [beginner](#), [intro](#)

</div>

## **title: jMonkeyEngine 3 Tutorial (12) - Hello Effects**

# **jMonkeyEngine 3 Tutorial (12) - Hello Effects**

Previous: [Hello Audio](#), Next: [Hello Physics](#)



When you see one of the following in a game, then a particle system is likely behind it:

- Fire, flames, sparks;
- Rain, snow, waterfalls, leaves;
- Explosions, debris, shockwaves;
- Dust, fog, clouds, smoke;
- Insects swarms, meteor showers;
- Magic spells.

These scene elements cannot be modeled by meshes. In very simple terms:

- The difference between an explosion and a dust cloud is the speed of the particle effect.
- The difference between flames and a waterfall is the direction and the color of the particle effect.

Particle effects can be animated (e.g. sparks, drops) and static (strands of grass, hair). Non-particle effects include bloom/glow, and motion blur/afterimage. In this tutorial you learn how to make animated particles (`com.jme3.effect`).

To use the example assets in a new jMonkeyEngine SDK project, right-click your project, select “Properties”, go to “Libraries”, press “Add Library” and add the “jme3-test-data” library.  
</div>

# Sample Code

```
package jme3test.helloworld;

import com.jme3.app.SimpleApplication;
import com.jme3.effect.ParticleEmitter;
import com.jme3.effect.ParticleMesh;
import com.jme3.material.Material;
import com.jme3.math.ColorRGBA;
import com.jme3.math.Vector3f;

/** Sample 11 - how to create fire, water, and explosion effects. */
public class HelloEffects extends SimpleApplication {

 public static void main(String[] args) {
 HelloEffects app = new HelloEffects();
 app.start();
 }

 @Override
 public void simpleInitApp() {

 ParticleEmitter fire =
 new ParticleEmitter("Emitter", ParticleMesh.Type.Triangulated);
 Material mat_red = new Material(assetManager,
 "Common/MatDefs/Misc/Particle.j3md");
 mat_red.setTexture("Texture", assetManager.loadTexture(
 "Effects/Explosion/flame.png"));
 fire.setMaterial(mat_red);
 fire.setImagesX(2);
 fire.setImagesY(2); // 2x2 texture animation
 fire.setEndColor(new ColorRGBA(1f, 0f, 0f, 1f)); // red
 fire.setStartColor(new ColorRGBA(1f, 1f, 0f, 0.5f)); // yellow
 fire.getParticleInfluencer().setInitialVelocity(new Vector3f(0,
 0, 0));
 fire.setStartSize(1.5f);
 fire.setEndSize(0.1f);
 fire.setGravity(0, 0, 0);
 fire.setLowLife(1f);
 fire.setHighLife(3f);
 }
}
```

```
 fire.getParticleInfluencer().setVelocityVariation(0.3f);
 rootNode.attachChild(fire);

 ParticleEmitter debris =
 new ParticleEmitter("Debris", ParticleMesh.Type.Triangl
Material debris_mat = new Material(assetManager,
 "Common/MatDefs/Misc/Particle.j3md");
debris_mat.setTexture("Texture", assetManager.loadTexture(
 "Effects/Explosion/Debris.png"));
debris.setMaterial(debris_mat);
debris.setImagesX(3);
debris.setImagesY(3); // 3x3 texture animation
debris.setRotateSpeed(4);
debris.setSelectRandomImage(true);
debris.getParticleInfluencer().setInitialVelocity(new Vector3f(
debris.setStartColor(ColorRGBA.White);
debris.setGravity(0, 6, 0);
debris.getParticleInfluencer().setVelocityVariation(.60f);
rootNode.attachChild(debris);
debris.emitAllParticles();
}
}
```

You should see an explosion that sends debris flying, and a fire. [More example code is here.](#)

## Texture Animation and Variation

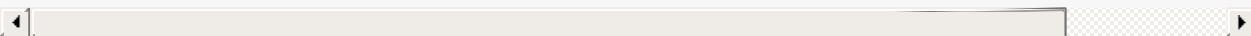


Start by choosing a material texture for your effect. If you provide the emitter with a set of textures (see image), it can use them either for variation (random order), or as animation steps (fixed order).

Setting emitter textures works just as you have already learned in previous chapters. This time you base the material on the `Particle.j3md` material definition. Let's have a closer look at the material for the Debris effect.

```
ParticleEmitter debris =
 new ParticleEmitter("Debris", ParticleMesh.Type.Triangl
Material debris_mat = new Material(assetManager,
 "Common/MatDefs/Misc/Particle.j3md");
debris_mat.setTexture("Texture", assetManager.loadTexture(
 "Effects/Explosion/Debris.png"));
debris.setMaterial(debris_mat);
debris.setImagesX(3);
debris.setImagesY(3); // 3x3 texture animation
debris.setSelectRandomImage(true);
...

```



1. Create a material and load the texture.
2. Tell the Emitter into how many animation steps ( $x^y$ ) the texture is divided.  
The debris texture has  $3 \times 3$  frames.
3. Optionally, tell the Emitter whether the animation steps are to be at random, or in order.  
For the debris, the frames play at random.

As you see in the debris example, texture animations improve effects because each “flame” or “piece of debris” now looks different. Also think of electric or magic effects, where you can create very interesting animations by using an ordered morphing series of lightning bolts; or flying leaves or snow flakes, for instance.

The fire material is created the same way, just using “Effects/Explosion/flame.png” texture, which has with  $2 \times 2$  ordered animation steps.

## Default Particle Textures

The following particle textures included in `test-data.jar`. You can copy and use them in your own effects.

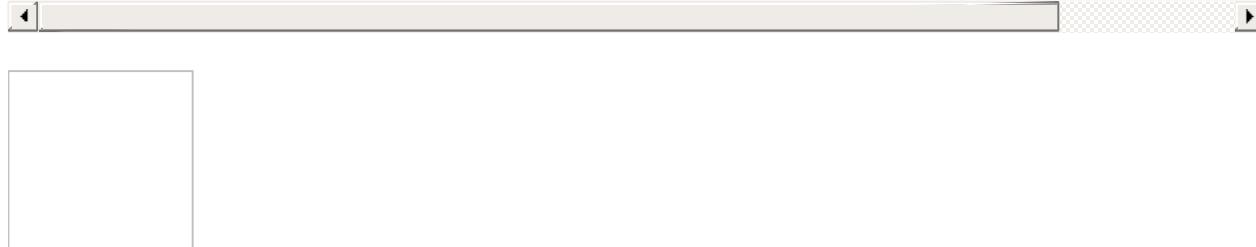
| Texture Path                     | Dimension | Preview |
|----------------------------------|-----------|---------|
| Effects/Explosion/Debris.png     | 3*3       |         |
| Effects/Explosion/flame.png      | 2*2       |         |
| Effects/Explosion/shockwave.png  | 1*1       |         |
| Effects/Explosion/smoketrail.png | 1*3       |         |
| Effects/Smoke/Smoke.png          | 1*15      |         |

Copy them into your `assets/Effects` directory to use them.

## Creating Custom Textures

For your game, you will likely create custom particle textures. Look at the fire example again.

```
ParticleEmitter fire =
 new ParticleEmitter("Emitter", ParticleMesh.Type.Triangle);
Material mat_red = new Material(assetManager,
 "Common/MatDefs/Misc/Particle.j3md");
mat_red.setTexture("Texture", assetManager.loadTexture(
 "Effects/Explosion/flame.png"));
fire.setMaterial(mat_red);
fire.setImagesX(2);
fire.setImagesY(2); // 2x2 texture animation
fire.setEndColor(new ColorRGBA(1f, 0f, 0f, 1f)); // red
fire.setStartColor(new ColorRGBA(1f, 1f, 0f, 0.5f)); // yellow
```



Compare the texture with the resulting effect.

- Black parts of the image become fully transparent.
- White/gray parts of the image are translucent and get colorized.

- You set the color using `setStartColor()` and `setEndColor()`.  
For fire, it's a gradient from yellow to red.
- By default, the animation is played in order and loops.

Create a grayscale texture in a graphic editor, and save it to your `assets/Effects` directory.  
If you split up one image file into  $x \times y$  animation steps, make sure each square is of equal size—just as you see in the examples here.

## Emitter Parameters

A particle system is always centered around an emitter.

Use the `setShape()` method to change the `EmitterShape`:

- `EmitterPointShape(Vector3f.ZERO)` – particles emit from a point (default)
- `EmitterSphereShape(Vector3f.ZERO, 2f)` – particles emit from a sphere-sized area
- `EmitterBoxShape(new Vector3f(-1f, -1f, -1f), new Vector3f(1f, 1f, 1f))` – particles emit from a box-sized area

Example:

```
emitter.setShape(new EmitterPointShape(Vector3f.ZERO));
```

You create different effects by changing the emitter parameters:

| Parameter     | Method                                                                                                                                                                                            | Default                                                     | Description                                                                                                                                                 |
|---------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| number        | <code>setNumParticles()</code>                                                                                                                                                                    | N/A                                                         | The maximum number of particles visible at the same time. Value is specified by user in constructor. This influences the density and length of the “trail”. |
| velocity      | <code>getParticleInfluencer().setInitialVelocity()</code>                                                                                                                                         | <code>Vector3f.ZERO</code>                                  | Specify a vector how fast particles move and in which start direction.                                                                                      |
| direction     | <code>getParticleInfluencer().setVelocityVariation()</code><br><code>setFacingVelocity()</code><br><code>setRandomAngle()</code><br><code>setFaceNormal()</code><br><code>setRotateSpeed()</code> | 0.2f<br>false<br>false<br><code>Vector3f.NAN</code><br>0.0f | Optional accessors that control in which direction particles face while flying.                                                                             |
| lifetime      | <code>setLowLife()</code><br><code>setHighLife()</code>                                                                                                                                           | 3f<br>7f                                                    | Minimum and maximum time period before particles fade.                                                                                                      |
| emission rate | <code>setParticlesPerSec()</code>                                                                                                                                                                 | 20                                                          | How many new particles are emitted per second.                                                                                                              |
| color         | <code>setStartColor()</code><br><code>setEndColor()</code>                                                                                                                                        | gray                                                        | Set to the same colors, or to two different colors for a gradient effect.                                                                                   |
| size          | <code>setStartSize()</code><br><code>setEndSize()</code>                                                                                                                                          | 0.2f<br>2f                                                  | Set to two different values for shrink/grow effect, or to same size for constant effect.                                                                    |
| gravity       | <code>setGravity()</code>                                                                                                                                                                         | 0,1,0                                                       | Whether particles fall down (positive) or fly up (negative). Set to 0f for a zero-g effect where particles keep flying.                                     |

You can find details about [effect parameters](#) here. Add and modify one parameter at a time, and try different values until you get the effect you want.

**Tip:** Use the SceneComposer in the jMonkeyEngine SDK to create effects more easily. Create an empty scene and add an emitter object to it. Change the emitter properties and watch the outcome live. You can save created effects as .j3o file and load them like scenes or models.

</div>

## Exercise

Can you “invert” the fire effect into a small waterfall? Here some tips:

- Change the Red and Yellow color to Cyan and Blue
- Invert the velocity vector (direction) by using a negative number
- Swap start and end size
- Activate gravity by setting it to 0,1,0

## Conclusion

You have learned that many different effects can be created by changing the parameters and textures of one general emitter object.

Now you move on to another exciting chapter – the simulation of [physical objects](#). Let's shoot some cannon balls at a brick wall!

---

[beginner](#), [documentation](#), [intro](#), [transparency](#), [effect](#)

</div>

# **title: jMonkeyEngine 3 Tutorial (13) - Hello Physics**

## **jMonkeyEngine 3 Tutorial (13) - Hello Physics**

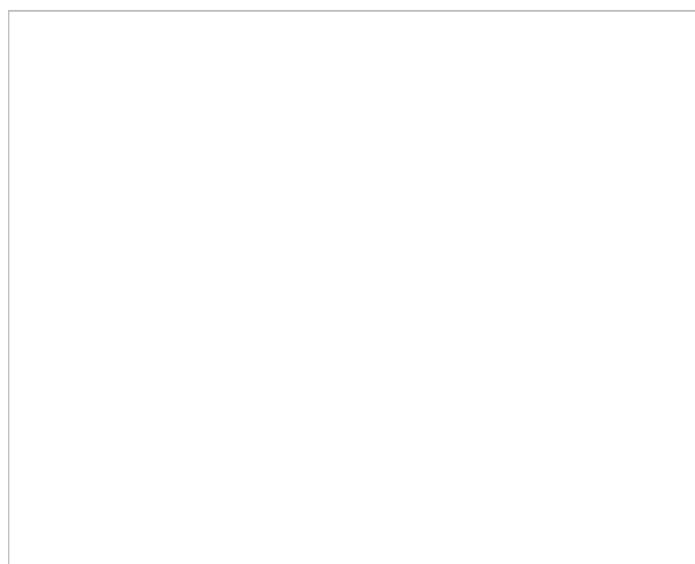
Previous: [Hello Effects](#), Next: [JME 3 documentation](#)

Do you remember the [Hello Collision](#) tutorial where you made the model of a town solid and walked through it in a first-person perspective? Then you may remember that, for the simulation of physical forces, jME3 integrates the [jBullet](#) library.

Apart from making models “solid”, the most common use cases for physics in 3D games are:

- Driving vehicles with suspensions, tyre friction, ramp jumping, drifting – Example: car racers
- Rolling and bouncing balls – Example: pong, pool billiard, bowling
- Sliding and falling boxes – Example: Breakout, Arkanoid
- Exposing objects to forces and gravity – Example: spaceships or zero-g flight
- Animating ragdolls – Example: “realistic” character simulations
- Swinging pendulums, rope bridges, flexible chains, and much more...

All these physical properties can be simulated in JME3. Let's have a look at a simulation of physical forces in this example where you shoot cannon balls at a brick wall.



To use the example assets in a new jMonkeyEngine SDK project, right-click your project, select “Properties”, go to “Libraries”, press “Add Library” and add the “jme3-test-data” library.  
</div>

## Sample Code

Thanks to double1984 for contributing this fun sample!

```
package jme3test.helloworld;

import com.jme3.app.SimpleApplication;
import com.jme3.asset.TextureKey;
import com.jme3.bullet.BulletAppState;
import com.jme3.bullet.control.RigidBodyControl;
import com.jme3.font.BitmapText;
import com.jme3.input.MouseInput;
import com.jme3.input.controls.ActionListener;
import com.jme3.input.controls.MouseButtonTrigger;
import com.jme3.material.Material;
import com.jme3.math.Vector2f;
import com.jme3.math.Vector3f;
import com.jme3.scene.Geometry;
import com.jme3.scene.shape.Box;
import com.jme3.scene.shape.Sphere;
import com.jme3.scene.shape.Sphere.TextureMode;
import com.jme3.texture.Texture;
import com.jme3.texture.Texture.WrapMode;

/**
 * Example 12 - how to give objects physical properties so they boun
 * @author base code by double1984, updated by zathras
 */
public class HelloPhysics extends SimpleApplication {

 public static void main(String args[]) {
 HelloPhysics app = new HelloPhysics();
 app.start();
 }
}
```

```
/** Prepare the Physics Application State (jBullet) */
private BulletAppState bulletAppState;

/** Prepare Materials */
Material wall_mat;
Material stone_mat;
Material floor_mat;

/** Prepare geometries and physical nodes for bricks and cannon ball */
private RigidBodyControl brick_phy;
private static final Box box;
private RigidBodyControl ball_phy;
private static final Sphere sphere;
private RigidBodyControl floor_phy;
private static final Box floor;

/** dimensions used for bricks and wall */
private static final float brickLength = 0.48f;
private static final float brickWidth = 0.24f;
private static final float brickHeight = 0.12f;

static {
 /** Initialize the cannon ball geometry */
 sphere = new Sphere(32, 32, 0.4f, true, false);
 sphere.setTextureMode(TextureMode.Projected);
 /** Initialize the brick geometry */
 box = new Box(brickLength, brickHeight, brickWidth);
 box.scaleTextureCoordinates(new Vector2f(1f, .5f));
 /** Initialize the floor geometry */
 floor = new Box(10f, 0.1f, 5f);
 floor.scaleTextureCoordinates(new Vector2f(3, 6));
}

@Override
public void simpleInitApp() {
 /** Set up Physics Game */
 bulletAppState = new BulletAppState();
 stateManager.attach(bulletAppState);
 //bulletAppState.getPhysicsSpace().enableDebug(assetManager);
```

```

 /** Configure cam to look at scene */
 cam.setLocation(new Vector3f(0, 4f, 6f));
 cam.lookAt(new Vector3f(2, 2, 0), Vector3f.UNIT_Y);
 /** Add InputManager action: Left click triggers shooting. */
 inputManager.addMapping("shoot",
 new MouseButtonTrigger(MouseInput.BUTTON_LEFT));
 inputManager.addListener(actionListener, "shoot");
 /** Initialize the scene, materials, and physics space */
 initMaterials();
 initWall();
 initFloor();
 initCrossHairs();
}

/**
 * Every time the shoot action is triggered, a new cannon ball is
 * The ball is set up to fly from the camera position in the same
 */
private ActionListener actionListener = new ActionListener() {
 public void onAction(String name, boolean keyPressed, float tpf
 if (name.equals("shoot") && !keyPressed) {
 makeCannonBall();
 }
 }
};

/** Initialize the materials used in this scene. */
public void initMaterials() {
 wall_mat = new Material(assetManager, "Common/MatDefs/Misc/Unsh
 TextureKey key = new TextureKey("Textures/Terrain/BrickWall/Bri
 key.setGenerateMips(true);
 Texture tex = assetManager.loadTexture(key);
 wall_mat.setTexture("ColorMap", tex);

 stone_mat = new Material(assetManager, "Common/MatDefs/Misc/Uns
 TextureKey key2 = new TextureKey("Textures/Terrain/Rock/Rock.PN
 key2.setGenerateMips(true);
 Texture tex2 = assetManager.loadTexture(key2);
 stone_mat.setTexture("ColorMap", tex2);
}

```

```

floor_mat = new Material(assetManager, "Common/MatDefs/Misc/Unshaded.j3m");
TextureKey key3 = new TextureKey("Textures/Terrain/Pond/Pond.jpg");
key3.setGenerateMips(true);
Texture tex3 = assetManager.loadTexture(key3);
tex3.setWrap(WrapMode.Repeat);
floor_mat.setTexture("ColorMap", tex3);
}

/** Make a solid floor and add it to the scene. */
public void initFloor() {
 Geometry floor_geo = new Geometry("Floor", floor);
 floor_geo.setMaterial(floor_mat);
 floor_geo.setLocalTranslation(0, -0.1f, 0);
 this.rootNode.attachChild(floor_geo);
 /* Make the floor physical with mass 0.0f! */
 floor_phy = new RigidBodyControl(0.0f);
 floor_geo.addControl(floor_phy);
 bulletAppState.getPhysicsSpace().add(floor_phy);
}

/** This loop builds a wall out of individual bricks. */
public void initWall() {
 float startpt = brickLength / 4;
 float height = 0;
 for (int j = 0; j < 15; j++) {
 for (int i = 0; i < 6; i++) {
 Vector3f vt =
 new Vector3f(i * brickLength * 2 + startpt, brickHeight +
 makeBrick(vt));
 }
 startpt = -startpt;
 height += 2 * brickHeight;
 }
}

/** This method creates one individual physical brick. */
public void makeBrick(Vector3f loc) {
 /** Create a brick geometry and attach to scene graph. */
 Geometry brick_geo = new Geometry("brick", box);
 brick_geo.setMaterial(wall_mat);
}

```

```

rootNode.attachChild(brick_geo);
/** Position the brick geometry */
brick_geo.setLocalTranslation(loc);
/** Make brick physical with a mass > 0.0f. */
brick_phy = new RigidBodyControl(2f);
/** Add physical brick to physics space. */
brick_geo.addControl(brick_phy);
bulletAppState.getPhysicsSpace().add(brick_phy);
}

/** This method creates one individual physical cannon ball.
 * By default, the ball is accelerated and flies
 * from the camera position in the camera direction.*/
public void makeCannonBall() {
 /** Create a cannon ball geometry and attach to scene graph. */
 Geometry ball_geo = new Geometry("cannon ball", sphere);
 ball_geo.setMaterial(stone_mat);
 rootNode.attachChild(ball_geo);
 /** Position the cannon ball */
 ball_geo.setLocalTranslation(cam.getLocation());
 /** Make the ball physical with a mass > 0.0f */
 ball_phy = new RigidBodyControl(1f);
 /** Add physical ball to physics space. */
 ball_geo.addControl(ball_phy);
 bulletAppState.getPhysicsSpace().add(ball_phy);
 /** Accelerate the physical ball to shoot it. */
 ball_phy.setLinearVelocity(cam.getDirection().mult(25));
}

/** A plus sign used as crosshairs to help the player with aiming
protected void initCrossHairs() {
 guiNode.detachAllChildren();
 guiFont = assetManager.loadFont("Interface/Fonts/Default.fnt");
 BitmapText ch = new BitmapText(guiFont, false);
 ch.setSize(guiFont.getCharSet().getRenderedSize() * 2);
 ch.setText("+"); // fake crosshairs :)
 ch.setLocalTranslation(// center
 settings.getWidth() / 2 - guiFont.getCharSet().getRenderedSize()
 settings.getHeight() / 2 + ch.getLineHeight() / 2, 0);
 guiNode.attachChild(ch);
}

```

```
 }
}
```

You should see a brick wall. Click to shoot cannon balls. Watch the bricks fall and bounce off one another!

## A Basic Physics Application

In the previous tutorials, you used static Geometries (boxes, spheres, and models) that you placed in the scene. Depending on their translation, Geometries can “float in mid-air” and even overlap – they are not affected by “gravity” and have no physical mass. This tutorial shows how to add physical properties to Geometries.

As always, start with a standard com.jme3.app.SimpleApplication. To activate physics, create a com.jme3.bullet.BulletAppState, and attach it to the SimpleApplication's AppState manager.

```
public class HelloPhysics extends SimpleApplication {
 private BulletAppState bulletAppState;

 public void simpleInitApp() {
 bulletAppState = new BulletAppState();
 stateManager.attach(bulletAppState);
 ...
 }
 ...
}
```

The BulletAppState gives the game access to a PhysicsSpace. The PhysicsSpace lets you use com.jme3.bullet.control.PhysicsControls that add physical properties to Nodes.

## Creating Bricks and Cannon Balls

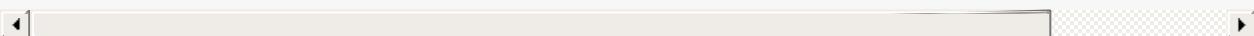
### Geometries

In this “shoot at the wall” example, you use Geometries such as cannon balls and bricks. Geometries contain meshes, such as Shapes. Let's create and initialize some Shapes: Boxes and Spheres.

```

/** Prepare geometries and physical nodes for bricks and cannon ball */
private static final Box box;
private static final Sphere sphere;
private static final Box floor;
/** dimensions used for bricks and wall */
private static final float brickLength = 0.48f;
private static final float brickWidth = 0.24f;
private static final float brickHeight = 0.12f;
static {
 /** Initialize the cannon ball geometry */
 sphere = new Sphere(32, 32, 0.4f, true, false);
 sphere.setTextureMode(TextureMode.Projected);
 /** Initialize the brick geometry */
 box = new Box(brickLength, brickHeight, brickWidth);
 box.scaleTextureCoordinates(new Vector2f(1f, .5f));
 /** Initialize the floor geometry */
 floor = new Box(10f, 0.1f, 5f);
 floor.scaleTextureCoordinates(new Vector2f(3, 6));
}

```



## RigidBodyControl: Brick

We want to create brick Geometries from those boxes. For each Geometry with physical properties, you create a RigidBodyControl.

```
private RigidBodyControl brick_phy;
```

The custom `makeBrick(loc)` methods creates individual bricks at the location `loc`. A brick has the following properties:

- It has a visible Geometry `brick_geo` (Box Shape Geometry).
- It has physical properties `brick_phy` (RigidBodyControl)

```

public void makeBrick(Vector3f loc) {
 /** Create a brick geometry and attach to scene graph. */
 Geometry brick_geo = new Geometry("brick", box);
 brick_geo.setMaterial(wall_mat);
 rootNode.attachChild(brick_geo);
 /** Position the brick geometry */
 brick_geo.setLocalTranslation(loc);
 /** Make brick physical with a mass > 0.0f. */
 brick_phy = new RigidBodyControl(2f);
 /** Add physical brick to physics space. */
 brick_geo.addControl(brick_phy);
 bulletAppState.getPhysicsSpace().add(brick_phy);
}

```

This code sample does the following:

1. You create a brick Geometry `brick_geo`. A Geometry describes the shape and look of an object.
  - `brick_geo` has a box shape
  - `brick_geo` has a brick-colored material.
2. You attach `brick_geo` to the `rootNode`
3. You position `brick_geo` at `loc`.
4. You create a `RigidBodyControl` `brick_phy` for `brick_geo`.
  - `brick_phy` has a mass of `2f`.
  - You add `brick_phy` to `brick_geo`.
  - You register `brick_phy` to the `PhysicsSpace`.

## RigidBodyControl: Cannonball

You notice that the cannon ball is created in the same way, using the custom `makeCannonBall()` method. The cannon ball has the following properties:

- It has a visible Geometry `ball_geo` (Sphere Shape Geometry)
- It has physical properties `ball_phy` (RigidBodyControl)

```

/** Create a cannon ball geometry and attach to scene graph. */
Geometry ball_geo = new Geometry("cannon ball", sphere);
ball_geo.setMaterial(stone_mat);
rootNode.attachChild(ball_geo);
/** Position the cannon ball */
ball_geo.setLocalTranslation(cam.getLocation());
/** Make the ball physical with a mass > 0.0f */
ball_phy = new RigidBodyControl(1f);
/** Add physical ball to physics space. */
ball_geo.addControl(ball_phy);
bulletAppState.getPhysicsSpace().add(ball_phy);
/** Accelerate the physical ball to shoot it. */
ball_phy.setLinearVelocity(cam.getDirection().mult(25));

```



This code sample does the following:

1. You create a ball Geometry `ball_geo`. A Geometry describes the shape and look of an object.
  - `ball_geo` has a sphere shape
  - `ball_geo` has a stone-colored material.
2. You attach `ball_geo` to the `rootNode`
3. You position `ball_geo` at the camera location.
4. You create a `RigidBodyControl` `ball_phy` for `ball_geo`.
  - `ball_phy` has a mass of 1f.
  - You add `ball_phy` to `ball_geo`.
  - You register `ball_phy` to the `PhysicsSpace`.

Since you are shooting cannon balls, the last line accelerates the ball in the direction the camera is looking, with a speed of 25f.

## RigidBodyControl: Floor

The (static) floor has one important difference compared to the (dynamic) bricks and cannonballs: **Static objects have a mass of zero**. As before, you write a custom `initFloor()` method that creates a flat box with a rock texture that you use as floor. The floor has the following properties:

- It has a visible Geometry `floor_geo` (Box Shape Geometry)
- It has physical properties `floor_phy` (RigidBodyControl)

```

public void initFloor() {
 Geometry floor_geo = new Geometry("Floor", floor);
 floor_geo.setMaterial(floor_mat);
 floor_geo.setLocalTranslation(0, -0.1f, 0);
 this.rootNode.attachChild(floor_geo);
 /* Make the floor physical with mass 0.0f! */
 floor_phy = new RigidBodyControl(0.0f);
 floor_geo.addControl(floor_phy);
 bulletAppState.getPhysicsSpace().add(floor_phy);
}

```

This code sample does the following:

1. You create a floor Geometry `floor_geo`. A Geometry describes the shape and look of an object.
  - `floor_geo` has a box shape
  - `floor_geo` has a pebble-colored material.
2. You attach `floor_geo` to the `rootNode`
3. You position `floor_geo` a bit below `y=0` (to prevent overlap with other PhysicControl'ed Spatials).
4. You create a `RigidBodyControl` `floor_phy` for `floor_geo`.
  - `floor_phy` has a mass of 0f
  - You add `floor_phy` to `floor_geo`.
  - You register `floor_phy` to the `PhysicsSpace`.

## Creating the Scene

Let's have a quick look at the custom helper methods:

- `initMaterial()` – This method initializes all the materials we use in this demo.
- `initWall()` – A double loop that generates a wall by positioning brick objects: 15 rows high with 6 bricks per row. It's important to space the physical bricks so they do not overlap.
- `initCrossHairs()` – This method simply displays a plus sign that you use as crosshairs for aiming. Note that screen elements such as crosshairs are attached to the `guiNode`, not the `rootNode` !
- `initInputs()` – This method sets up the click-to-shoot action.

These methods are each called once from the `simpleInitApp()` method at the start of the game. As you see, you can write any number of custom methods to set up your game's scene.

## The Cannon Ball Shooting Action

In the `initInputs()` method, you add an input mapping that triggers a shoot action when the left mouse button is pressed.

```
private void initInputs() {
 inputManager.addMapping("shoot",
 new MouseButtonTrigger(MouseInput.BUTTON_LEFT));
 inputManager.addListener(actionListener, "shoot");
}
```

You define the actual action of shooting a new cannon ball as follows:

```
private ActionListener actionListener = new ActionListener() {
 public void onAction(String name, boolean keyPressed, float tpf) {
 if (name.equals("shoot") && !keyPressed) {
 makeCannonBall();
 }
 }
};
```

In the moment the cannonball appears in the scene, it flies off with the velocity (and in the direction) that you specified using `setLinearVelocity()` inside `makeCannonBall()`. The newly created cannon ball flies off, hits the wall, and exerts a *physical force* that impacts individual bricks.

## Moving a Physical Spatial

The location of the dynamic Spatial is controlled by its RigidBodyControl. Move the RigidBodyControl to move the Spatial. If it's a dynamic PhysicsControl, you can use `setLinearVelocity()` and apply forces and torques to it. Other RigidBodyControl'led objects can push the dynamic Spatial around (like pool/billiard balls).

You can make Spatialss that are not dynamic: Switch the RigidBodyControl to setKinematic(true) to have it move along with its Spatial.

- A kinematic is unaffected by forces or gravity, which means it can float in mid-air and cannot be pushed away by dynamic “cannon balls” etc.
- A kinematic RigidBody has a mass.
- A kinematic can be moved and can exert forces on dynamic RigidBodys. This means you can use a kinematic node as a billiard cue or a remote-controlled battering ram.

Learn more about static versus kinematic versus dynamic in the [advanced physics doc](#).

## Excercises

### Exercise 1: Debug Shapes

Add the following line after the bulletAppState initialization.

```
bulletAppState.getPhysicsSpace().enableDebug(assetManager);
```

Now you see the collisionShapes of the bricks and spheres, and the floor highlighted.

### Exercise 2: No Mo' Static

What happens if you give a static node, such as the floor, a mass of more than 0.0f?

### Exercise 3: Behind the Curtain

Fill your scene with walls, bricks, and cannon balls. When do you begin to see a performance impact?

Popular AAA games use a clever mix of physics, animation and prerendered graphics to give you the illusion of a real, “physical” world. Think of your favorite video games and try to spot where and how the game designers trick you into believing that the whole scene is physical. For example, think of a building “breaking” into 4-8 parts after an explosion. The pieces most likely fly on predefined (so called kinematic) paths and are only replaced by dynamic Spatialss after they touch the ground... Now that you start to implement game physics yourself, look behind the curtain!

Using physics everywhere in a game sounds like a cool idea, but it is easily overused. Although the physics nodes are put to “sleep” when they are not moving, creating a world solely out of dynamic physics nodes will quickly bring you to the limits of your computer’s

capabilities.

## Conclusion

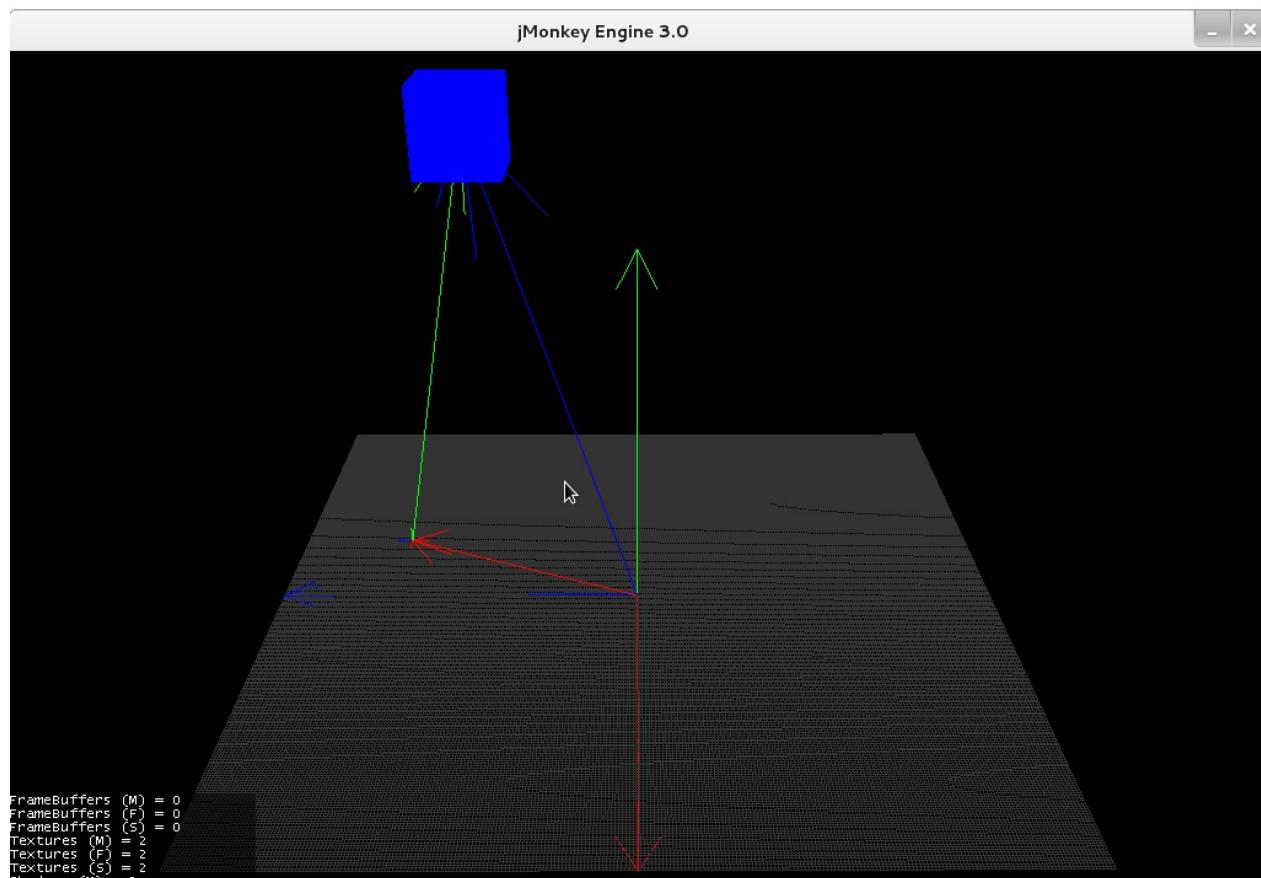
You have learned how to activate the jBullet PhysicsSpace in an application by adding a `BulletAppState`. You have created PhysicsControls for simple Shape-based Geometries (for more complex shapes, read up on [CollisionShapes](#)). You have learned that physical objects are not only attached to the rootNode, but also registered to the PhysicsSpace. You know that it makes a difference whether a physical object has a mass (dynamic) or not (static). You are aware that overusing physics has a huge performance impact.

Congratulations! – You have completed the last beginner tutorial. Now you are ready to start [combining what you have learned](#), to create a cool 3D game of your own. Show us what you can do, and feel free to share your demos, game videos, and screenshots on the [Free Announcements Forum!](#)

[beginner](#), [intro](#), [physics](#), [documentation](#), [input](#), [model](#), [control](#)

</div>

## title:



### HelloVectorSumm.java

```
package org.jmonkey.chapter2.hellovector;

import com.jme3.app.SimpleApplication;
import com.jme3.material.Material;
import com.jme3.math.ColorRGBA;
import com.jme3.math.Vector3f;
import com.jme3.renderer.RenderManager;
import com.jme3.scene.Geometry;
import com.jme3.scene.Node;
import com.jme3.scene.shape.Box;
import org.jmonkey.utils.Debug;
import org.jmonkey.utils.MaterialUtils;
import org.jmonkey.utils.SpatialUtils;
```

```
/*
Example Vector Summ

@author Alex Cham aka Jcrypto
*/
public class HelloVectorSumm extends SimpleApplication
{

 private Node vctrNode = SpatialUtils.makeNode("vectorNode");
 //
 private Vector3f vctrNodeLoc = new Vector3f(64.0f, 64.0f, 64.0f);
 private Vector3f camLocVctr = new Vector3f(512.0f, 64.0f, 0.0f);
 //
 private Vector3f vctrNodeSpatLoc = new Vector3f(64.0f, 128.0f,
 //
 private Vector3f vctrSumm = null;
 private Vector3f scale = new Vector3f(8, 8, 8);

 public static void main(String[] args)
 {
 HelloVectorSumm app = new HelloVectorSumm();
 //app.setShowSettings(false);
 app.start();
 }

 @Override
 public void simpleInitApp()
 {
 cam.setLocation(camLocVctr);
 cam.lookAt(Vector3f.ZERO, cam.getUp());
 flyCam.setMoveSpeed(100.0f);
 //
 Debug.showNodeAxes(assetManager, rootNode, 128);
 Debug.attachWireFrameDebugGrid(assetManager, rootNode, Vect

 //
 Box box = new Box(1, 1, 1);
 //
 Material mat = MaterialUtils.makeMaterial(assetManager, "Co
```

```
 Geometry geom = SpatialUtils.makeGeometry(vctrNodeSpatLoc,
vctrNode.attachChild(geom);
vctrNode.setLocalTranslation(vctrNodeLoc);
vctrSumm = vctrNodeLoc.add(vctrNodeSpatLoc);
//
Debug.showNodeAxes(assetManager, vctrNode, 4.0f);
Debug.showVector3fArrow(assetManager, rootNode, vctrNodeLoc
Debug.showVector3fArrow(assetManager, vctrNode, vctrNodeSpatLoc);
Debug.showVector3fArrow(assetManager, rootNode, vctrSumm, Color.CYAN);
//
rootNode.attachChild(vctrNode);
}

@Override
public void simpleUpdate(float tpf)
{
 //n.move(tpf * 10, 0, 0);
}

@Override
public void simpleRender(RenderManager rm)
{
 //TODO: add render code
}
}
```



```
package org.jmonkey.utils;

import com.jme3.asset.AssetManager;
import com.jme3.material.Material;
import com.jme3.math.ColorRGBA;

/**
 * Example Vector Summ
 * @author Alex Cham aka Jcrypto
 */

public class MaterialUtils
{

 public MaterialUtils()
 {
 }

 //Common/MatDefs/Misc/Unshaded.j3md"
 public static Material makeMaterial(AssetManager am, String nam
 {
 Material mat = new Material(am, name);
 mat.setColor("Color", color);
 return mat;
 }
}
```

```
package org.jmonkey.utils;

import com.jme3.material.Material;
import com.jme3.math.Vector3f;
import com.jme3.scene.Geometry;
import com.jme3.scene.Mesh;
import com.jme3.scene.Node;

/**
```

```

 * Example Vector Summ
 * @author Alex Cham aka Jcrypto
 */
public class SpatialUtils
{
 //
 public static Node makeNode(String name)
 {
 Node n = new Node(name);
 return n;
 }

 //
 public static Geometry makeGeometry(Mesh mesh, Material mat, St
 {
 Geometry geom = new Geometry(name, mesh);
 geom.setMaterial(mat);
 return geom;
 }

 //
 public static Geometry makeGeometry(Vector3f loc, Vector3f scl,
 {
 Geometry geom = new Geometry(name, mesh);
 geom.setMaterial(mat);
 geom.setLocalTranslation(loc);
 geom.setLocalScale(scl);
 return geom;
 }
}

```

— Debug.java

```

package org.jmonkey.utils;

import com.jme3.animation.AnimControl;
import com.jme3.asset.AssetManager;
import com.jme3.material.Material;
import com.jme3.math.ColorRGBA;

```

```

import com.jme3.math.Vector3f;
import com.jme3.scene.Geometry;
import com.jme3.scene.Node;
import com.jme3.scene.debug.Arrow;
import com.jme3.scene.debug.Grid;
import com.jme3.scene.debug.SkeletonDebugger;
import com.jme3.scene.shape.Line;
import static org.jmonkey.utils.SpatialUtils.makeGeometry;

/**
 * Example Vector Summ

@author Alex Cham aka Jcrypto
*/
public class Debug
{

 public static void showNodeAxes(AssetManager am, Node n, float
 {
 Vector3f v = new Vector3f(axisLen, 0, 0);
 Arrow a = new Arrow(v);
 Material mat = new Material(am, "Common/MatDefs/Misc/Unshad
 mat.setColor("Color", ColorRGBA.Red);
 Geometry geom = new Geometry(n.getName() + "XAxis", a);
 geom.setMaterial(mat);
 n.attachChild(geom);

 //
 v = new Vector3f(0, axisLen, 0);
 a = new Arrow(v);
 mat = new Material(am, "Common/MatDefs/Misc/Unshaded.j3md")
 mat.setColor("Color", ColorRGBA.Green);
 geom = new Geometry(n.getName() + "YAxis", a);
 geom.setMaterial(mat);
 n.attachChild(geom);

 //
 v = new Vector3f(0, 0, axisLen);
 }
}

```

```
a = new Arrow(v);
mat = new Material(am, "Common/MatDefs/Misc/Unshaded.j3md")
mat.setColor("Color", ColorRGBA.Blue);
geom = new Geometry(n.getName() + "ZAxis", a);
geom.setMaterial(mat);
n.attachChild(geom);

}

//

public static void showVector3fArrow(AssetManager am, Node n, V
{
 Arrow a = new Arrow(v);
 Material mat = MaterialUtils.makeMaterial(am, "Common/MatDe
 Geometry geom = makeGeometry(a, mat, name);
 n.attachChild(geom);
}

public static void showVector3fLine(AssetManager am, Node n, Ve
{
 Line l = new Line(v.subtract(v), v);
 Material mat = MaterialUtils.makeMaterial(am, "Common/MatDe
 Geometry geom = makeGeometry(l, mat, name);
 n.attachChild(geom);
}

//Skeleton Debugger
public static void attachSkeleton(AssetManager am, Node player,
{
 SkeletonDebugger skeletonDebug = new SkeletonDebugger("skele
 Material mat2 = new Material(am, "Common/MatDefs/Misc/Unsha
 mat2.setColor("Color", ColorRGBA.Yellow);
 mat2.getAdditionalRenderState().setDepthTest(false);
 skeletonDebug.setMaterial(mat2);
 player.attachChild(skeletonDebug);
}

///

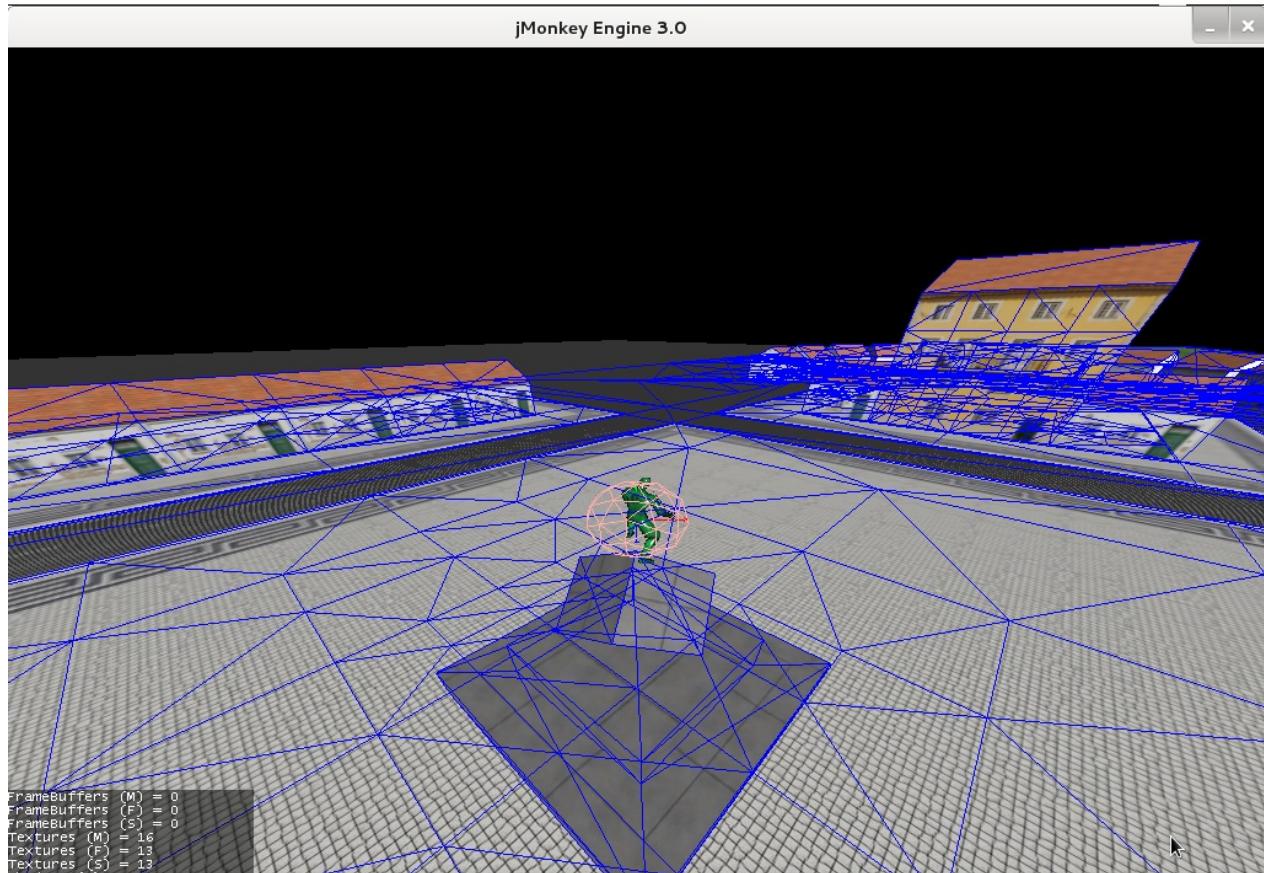
public static void attachWireFrameDebugGrid(AssetManager assetM
{
 Geometry g = new Geometry("wireFrameDebugGrid", new Grid(si
```

```
 Material mat = new Material(assetManager, "Common/MatDefs/M
 mat.getAdditionalRenderState().setWireframe(true);
 mat.setColor("Color", color);
 g.setMaterial(mat);
 g.center().move(pos);
 n.attachChild(g);
}
}
```



## title:

[Youtube video](#)



- [Ninja Mesh](#)
- [Town Scene](#)
  
- [Walk Bugs](#)
- [Jump Bugs](#)

Memo:

1. jMonkeyEngine calls the update() methods of all AppState objects in the order in which you attached them.
2. jMonkeyEngine calls the controlUpdate() methods of all controls in the order in which you added them.
3. jMonkeyEngine calls the simpleUpdate() method of the main SimpleApplication class.

—AbstractPhysicsContext

```
package org.jmonkey3.chasecam;

import com.jme3.app.state.AbstractAppState;
import com.jme3.app.state.AppStateManager;
import com.jme3.bullet.BulletAppState;

/*
Chase camera (aka 3rd person camera) example
Based on official TestQ3.java

@author Alex Cham aka Jcrypto
*/
public abstract class AbstractPhysicBodyContext extends AbstractApp
{

 private AppStateManager stateManager = null;
 //
 private static final BulletAppState bulletAppState;

 static
 {
 bulletAppState = new BulletAppState();
 }

 public AbstractPhysicBodyContext()
 {
 }

 /**
 * @return the bulletAppState
 */
 public BulletAppState getBulletAppState()
 {
 return bulletAppState;
 }

 /**
 * @param stateManager the stateManager to set
 * Attaching BulletAppstate to Initialize PhysicsSpace
 */
}
```

```
public void attachBulletAppState(AppStateManager stateManager)
{
 this.stateManager = stateManager;
 stateManager.attach(bulletAppState);
}
```

—AbstractSpatialBodyContext.java

```
package org.jmonkey3.chasecam;

import com.jme3.app.state.AbstractAppState;

/*
Chase camera (aka 3rd person camera) example
Based on official TestQ3.java

@author Alex Cham aka Jcrypto
*/
public abstract class AbstractSpatialBodyContext extends AbstractAp
{

 public AbstractSpatialBodyContext()
 {
 }

}
```

—ApplicationContext.java

```
package org.jmonkey3.chasecam;

import com.jme3.app.Application;
import com.jme3.app.state.AbstractAppState;
import com.jme3.app.state.AppStateManager;
import com.jme3.asset.AssetManager;
import com.jme3.input.FlyByCamera;
import com.jme3.input.InputManager;
```

```
import com.jme3.math.ColorRGBA;
import com.jme3.math.Vector3f;
import com.jme3.renderer.Camera;
import com.jme3.scene.Node;
import com.jme3.system.AppSettings;
import org.jmonkey.utils.Debug;

/*
Chase camera (aka 3rd person camera) example
Based on official TestQ3.java

@author Alex Cham aka Jcrypto
*/
public class ApplicationContext extends AbstractAppState
{

 private final Node rootNode;
 //

 private final CameraContext sceneCameraContext;
 private final AvatarBodyManager avatarBodyManager;
 private final SceneBodyManager sceneBodyManager;

 /**
 * @param stateManager
 * @param am
 * @param settings
 * @param inputManager
 * @param rootNode
 * @param cam
 * @param flyByCam
 */
 public ApplicationContext(AppStateManager stateManager, AssetMa
 {

 this.rootNode = rootNode;
 this.sceneCameraContext = new CameraContext(settings, input
 this.sceneBodyManager = new SceneBodyManager(stateManager,
 this.avatarBodyManager = new AvatarBodyManager(am, rootNode
 }
}
```

```

@Override
public void initialize(AppStateManager stateManager, Application app)
{
 //super.initialize(stateManager, app);
 //TODO: initialize your AppState, e.g. attach spatials to rootNode
 //this is called on the OpenGL thread after the AppState has been
 //initialized
 //
 stateManager.attach(this.sceneCameraContext);
 stateManager.attach(this.sceneBodyManager); //initialize physics
 stateManager.attach(this.avatarBodyManager);
 //
 Debug.showNodeAxes(app.getAssetManager(), this.rootNode, 10);
 Debug.attachWireFrameDebugGrid(app.getAssetManager(), rootNode);
}

@Override
public void update(float tpf)
{
}

```

—AvatarAnimationEventListener.java

```

package org.jmonkey3.chasecam;

import com.jme3.animation.AnimChannel;
import com.jme3.animation.AnimControl;
import com.jme3.animation.AnimEventListener;
import com.jme3.app.Application;
import com.jme3.app.state.AbstractAppState;
import com.jme3.app.state.AppStateManager;
import com.jme3.bullet.objects.PhysicsCharacter;
import com.jme3.scene.Spatial;

/*
 * Chase camera (aka 3rd person camera) example
 */

```

Based on official TestQ3.java

```
@author Alex Cham aka Jcrypto
*/
public class AvatarAnimationEventListener extends AbstractAppState
{

 private final AnimChannel channel;
 private final AnimControl control;
 private final PlayerInputActionListener pial;
 private final AvatarAnimationHelper animHelper;
 private final PhysicsCharacter physicBody;
 /**
 * @param pial
 * @param pc
 * @param avatarMesh
 */
 public AvatarAnimationEventListener(PlayerInputActionListener p
 {
 this.pial = pial;
 this.control = avatarMesh.getControl(AnimControl.class);
 assert (this.control != null);
 this.channel = this.control.createChannel();
 this.physicBody = pc;
 this.animHelper = new AvatarAnimationHelper(this.physicBody)
 }

 @Override
 public void initialize(AppStateManager stateManager, Application
 {
 this.control.addListener(this);
 this.channel.setAnim("Idle1");
 this.channel.setSpeed(0.5f);
 }

 public void onAnimCycleDone(AnimControl control, AnimChannel ch
 {
 //throw new UnsupportedOperationException("Not supported ye
 }
}
```

```
public void onAnimChange(AnimControl control, AnimChannel chan
{
 //throw new UnsupportedOperationException("Not supported ye

}

/**
 * @return the channel
 */
protected AnimChannel getChannel()
{
 return channel;
}

/**
 * @return the control
 */
protected AnimControl getControl()
{
 return control;
}

/**
 * @return the animHelper
 */
protected AvatarAnimationHelper getAnimHelper()
{
 return animHelper;
}
}
```

—AvatarAnimationHelper.java

```
package org.jmonkey3.chasecam;

import com.jme3.animation.AnimChannel;
import com.jme3.animation.LoopMode;
import com.jme3.bullet.objects.PhysicsCharacter;
```

```
/*
Chase camera (aka 3rd person camera) example
Based on official TestQ3.java

@author Alex Cham aka Jcrypto
*/
public class AvatarAnimationHelper
{

 private final AnimChannel animChannel;
 private final PhysicsCharacter physicBody;
 /**

 * @param pc
 * @param ac
 */
 public AvatarAnimationHelper(PhysicsCharacter pc, AnimChannel ac)
 {
 this.animChannel = ac;
 this.physicBody = pc;
 }

 protected void idle()
 {
 animChannel.setAnim("Idle1");
 animChannel.setSpeed(0.5f);
 }

 protected boolean forward(boolean pressed)
 {
 if (pressed)
 {
 if (this.physicBody.onGround())
 {
 animChannel.setAnim("Walk");
 animChannel.setSpeed(AvatarConstants.FORWARD_MOVE_SPEED);
 animChannel.setLoopMode(LoopMode.Loop);
 }
 return true;
 }
 }
}
```

```
 } else
 {
 idle();
 return false;
 }
 //throw new UnsupportedOperationException("Not supported yet");
 }

protected boolean backward(boolean pressed)
{
 if (pressed)
 {
 return true;
 } else
 {
 return false;
 }
}

protected boolean rightward(boolean pressed)
{
 if (pressed)
 {
 return true;
 } else
 {
 return false;
 }
}

protected boolean leftward(boolean pressed)
{
 if (pressed)
 {
 return true;
 } else
 {
 return false;
 }
}
```

```

protected boolean jump(boolean pressed)
{
 if (pressed)
 {
 if (this.physicBody.onGround())
 {
 animChannel.setAnim("HighJump");
 animChannel.setSpeed(AvatarConstants.FORWARD_MOTION_SPEED);
 animChannel.setLoopMode(LoopMode.DontLoop);
 //
 this.physicBody.jump();
 }
 return true;
 } else
 {
 return false;
 }
}

```

## —AvatarBodyManager.java

```

package org.jmonkey3.chasecam;

import com.jme3.app.Application;
import com.jme3.app.state.AppStateManager;
import com.jme3.asset.AssetManager;
import com.jme3.bullet.control.BetterCharacterControl;
import com.jme3.bullet.objects.PhysicsCharacter;
import com.jme3.input.ChaseCamera;
import com.jme3.input.InputManager;
import com.jme3.renderer.Camera;
import com.jme3.scene.Node;
import org.jmonkey.utils.Debug;

/*
Chase camera (aka 3rd person camera) example
Based on official TestQ3.java

```

```
@author Alex Cham aka Jcrypto
*/
public class AvatarBodyManager extends AbstractPhysicBodyContext
{

 private InputManager inputManager;
 private final Node rootNode;
 //
 private final CameraContext cc;
 private final Camera cam;
 private final ChaseCamera chaseCam;
 //
 private final AvatarPhysicBodyContext apbc;
 private final AvatarSpatialBodyContext asbc;
 //
 private final PhysicsCharacter physicBody;
 private final Node avatar;
 private final BetterCharacterControl bcc;
 //

 private final PlayerInputActionListener playerInputListener;

 /**
 * @param am
 * @param rootNode
 * @param cc
 */
 public AvatarBodyManager(AssetManager am, Node rootNode, Camera
 {

 //
 this.rootNode = rootNode;
 //
 this.asbc = new AvatarSpatialBodyContext(am, rootNode);
 this.apbc = new AvatarPhysicBodyContext();
 //
 this.physicBody = apbc.getPhysicBody();

 }
}
```

```
 this.avatar = asbc.getAvatar();
 this.bcc = new BetterCharacterControl(AvatarConstants.COLLISION_TYPE);
 // ...
 this.playerInputListener = new PlayerInputActionListener(this);
 // ...
 this.cc = cc;
 this.cam = cc.getCam();
 this.chaseCam = cc.getChaseCam();
 }

@Override
public void initialize(AppStateManager stateManager, Application app)
{
 //TODO: initialize your AppState, e.g. attach spatlials to resources
 //this is called on the OpenGL thread after the AppState has been created

 stateManager.attach(this.asbc);
 stateManager.attach(this.apbc);
 stateManager.attach(this.playerInputListener);

 // ...
 this.avatar.addControl(new AvatarBodyMoveControl(playerInputListener));
 this.avatar.addControl(chaseCam);
 this.avatar.addControl(bcc);

 //DEBUG
 Debug.showNodeAxes(app.getAssetManager(), avatar, 4);
 getBulletAppState().getPhysicsSpace().enableDebug(app.getAssetManager());
}

@Override
public void update(float tpf)
{
 //assert (sceneCameraContext != null);

 //correctDirectionVectors(cam.getDirection(), cam.getLeft());
}
```

```
 }
}
```

—AvatarBodyMoveControl.java

```
package org.jmonkey3.chasecam;

import com.jme3.bullet.control.BetterCharacterControl;
import com.jme3.bullet.objects.PhysicsCharacter;
import com.jme3.math.Vector3f;
import com.jme3.renderer.Camera;
import com.jme3.renderer.RenderManager;
import com.jme3.renderer.ViewPort;
import com.jme3.scene.control.AbstractControl;

/*
Chase camera (aka 3rd person camera) example
Based on official TestQ3.java

@author Alex Cham aka Jcrypto
*/
public class AvatarBodyMoveControl extends AbstractControl
{
 private final Camera cam;
 private final PhysicsCharacter physicBody;
 private final PlayerInputActionListener pial;
 /**

 * @param pial
 * @param physicBody
 * @param cam
 */
 public AvatarBodyMoveControl(PlayerInputActionListener pial, Pr
 {
 this.pial = pial;
 this.physicBody = physicBody;
 this.cam = cam;
 }
 private final Vector3f walkDirection = new Vector3f();
```

```
@Override
protected void controlUpdate(float tpf)
{
 //throw new UnsupportedOperationException("Not supported yet");
 correctDirectionVectors();
}

@Override
protected void controlRender(RenderManager rm, ViewPort vp)
{
 //throw new UnsupportedOperationException("Not supported yet");
}

/**
 * @param camDir
 * @param camLeft
 */
public void correctDirectionVectors()
{
 //assert (camDir != null);
 //assert (camLeft != null);
 //assert (walkDirection != null);
 //Affect forward, backward move speed 0.6f lower - 1.0f faster
 Vector3f camDirVector = cam.getDirection().clone().multLocal(...);
 //Affect left, right move speed 0.6f lower - 1.0f faster
 Vector3f camLeftVector = cam.getLeft().clone().multLocal(...);

 walkDirection.set(0, 0, 0); //critical
 if (pial.isLeftward())
 {
 walkDirection.addLocal(camLeftVector);
 }
 if (pial.isRightward())
 {
 walkDirection.addLocal(camLeftVector.negate());
 }
 if (pial.isForward())
 {
 walkDirection.addLocal(...);
 }
}
```

```
{
 walkDirection.addLocal(camDirVector);
}
if (pial.isBackward())
{
 //@@TODO Bug if cam direction (0, -n, 0) - character fly
 walkDirection.addLocal(camDirVector.negate());
}
physicBody.setWalkDirection(walkDirection);//Critical

//Avoid vibration
spatial.setLocalTranslation(physicBody.getPhysicsLocation()
//Translate Node accordingly
spatial.getControl(BetterCharacterControl.class).warp(physi
//Rotate Node accordingly to camera
spatial.getControl(
 BetterCharacterControl.class).setViewDirection(
 cam.getDirection().negate());

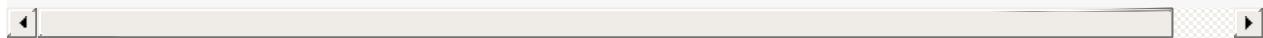
}
}
]
[<] [>]
```

—AvatarConstants.java

```
package org.jmonkey3.chasecam;

/*
Chase camera (aka 3rd person camera) example
Based on official TestQ3.java

@author Alex Cham aka Jcrypto
*/
public class AvatarConstants
{
 public static final float COLLISION_SHAPE_CENTRAL_POINT = 0.0f;
 public static final float COLLISION_SHAPE_RADIUS = 4.0f;
 //
 public static final float PHYSIC_BODY_MASS = 1.0f;
 public static float FORWARD_MOVE_SPEED = 0.8f;
 public static float SIDEWARD_MOVE_SPEED = 0.6f;
}
```



#### —AvatarPhysicBodyContext.java

```
package org.jmonkey3.chasecam;

import com.jme3.app.Application;
import com.jme3.app.state.AppStateManager;
import com.jme3.bullet.collision.shapes.SphereCollisionShape;
import com.jme3.bullet.objects.PhysicsCharacter;
import com.jme3.math.Vector3f;

/*
Chase camera (aka 3rd person camera) example
Based on official TestQ3.java

@author Alex Cham aka Jcrypto
*/
public class AvatarPhysicBodyContext extends AbstractPhysicBodyCont
{
```

```
private final PhysicsCharacter physicBody;

public AvatarPhysicBodyContext()
{
 this.physicBody = new PhysicsCharacter(new SphereCollisionShape());
}

@Override
public void initialize(AppStateManager stateManager, Application application)
{
 // ...
 assert (getBulletAppState() != null);
 System.out.println(this.getClass().getName() + ".getBulletAppState()");

 // ...
 this.physicBody.setJumpSpeed(32);
 this.physicBody.setFallSpeed(32);
 this.physicBody.setGravity(32);
 this.physicBody.setPhysicsLocation(new Vector3f(0, 10, 0));
 // ...
 getBulletAppState().getPhysicsSpace().add(this.physicBody);
}

@Override
public void update(float tpf)
{
}

@Override
public void cleanup()
{
 super.cleanup();
}
```

```

 /**
 * @return the physicBody
 */
 public PhysicsCharacter getPhysicBody()
 {
 return this.physicBody;
 }
}

```



## —AvatarSpatialBodyContext.java

```

package org.jmonkey3.chasecam;

import com.jme3.app.Application;
import com.jme3.app.state.AppStateManager;
import com.jme3.asset.AssetManager;
import com.jme3.math.Vector3f;
import com.jme3.scene.Node;
import com.jme3.scene.Spatial;

/*
Chase camera (aka 3rd person camera) example
Based on official TestQ3.java

@author Alex Cham aka Jcrypto
*/
public class AvatarSpatialBodyContext extends AbstractSpatialBodyC
{

 //
 private final Node rootNode;
 //
 private final Node avatar;
 private final Spatial avatarMesh;
 private final Vector3f correction;
 /**

 * @param am
 * @param rootNode

```

```
/*
public AvatarSpatialBodyContext(AssetManager am, Node rootNode)
{
 this.rootNode = rootNode;
 //
 this.avatar = new Node();
 this.avatarMesh = am.loadModel("Models/Ninja/Ninja.mesh.xml");
 this.correction = new Vector3f(
 0,
 AvatarConstants.COLLISION_SHAPE_CENTERAL_POINT - AvatarConstants.COLLISION_SHAPE_CENTERAL_POINT);
}

@Override
public void initialize(AppStateManager stateManager, Application app)
{

 this.avatarMesh.setLocalScale(new Vector3f(0.05f, 0.05f, 0.05f));
 this.avatarMesh.setLocalTranslation(this.correction);
 this.avatar.attachChild(this.avatarMesh);
 this.rootNode.attachChild(this.avatar);

 //super.initialize(stateManager, app); //To change body of
}

/**
 * @return the avatar
 */
public Node getAvatar()
{
 return avatar;
}

/**
 * @return the avatarMesh
 */
public Spatial getAvatarMesh()
{
 return avatarMesh;
}
```

```
 }
}
```

## —CameraContext.java

```
package org.jmonkey3.chasecam;

import com.jme3.app.Application;
import com.jme3.app.state.AbstractAppState;
import com.jme3.app.state.AppStateManager;
import com.jme3.input.ChaseCamera;
import com.jme3.input.FlyByCamera;
import com.jme3.input.InputManager;
import com.jme3.renderer.Camera;
import com.jme3.system.AppSettings;

/*
Chase camera (aka 3rd person camera) example
Based on official TestQ3.java

@author Alex Cham aka Jcrypto
*/
public class CameraContext extends AbstractAppState
{

 private final AppSettings settings;
 private final InputManager inputManager;
 /*
 * http://hub.jmonkeyengine.org/javadoc/com/jme3/renderer/Camera.
 * public class Camera
 * extends java.lang.Object
 * implements Savable, java.lang.Cloneable

 * Width and height are set to the current Application's settings
 * Frustum Perspective:
 * Frame of view angle of 45° along the Y axis
 * Aspect ratio of width divided by height
 * Near view plane of 1 wu
 * Far view plane of 1000 wu
 */
}
```

```
Start location at (0f, 0f, 10f).
Start direction is looking at the origin.
*/
private final Camera cam;
/*
http://hub.jmonkeyengine.org/javadoc/com/jme3/input/ChaseCamera.html
public class ChaseCamera
extends java.lang.Object
implements ActionListener, AnalogListener, Control

A camera that follows a spatial and can turn around it by drag
Constructs the chase camera, and registers inputs if you use t
constructor you have to attach the cam later to a spatial doir
spatial.addControl(chaseCamera);
*/
private final ChaseCamera chaseCam;
private final FlyByCamera flyByCam;

/**
@param settings
@param inputManager
@param cam
@param flyByCam
*/
public CameraContext(AppSettings settings, InputManager inputMa
{

 assert (settings != null);
 this.settings = settings;
 assert (inputManager != null);
 this.inputManager = inputManager;
 assert (cam != null);
 this.cam = cam;
 assert (flyByCam != null);
 this.flyByCam = flyByCam;
 this.chaseCam = new ChaseCamera(this.cam, this.inputManager
}

@Override
```

```

public void initialize(AppStateManager stateManager, Application app)
{
 super.initialize(stateManager, app);
 //TODO: initialize your AppState, e.g. attach spatlals to root
 //this is called on the OpenGL thread after the AppState has been
 //initialized

 this.cam.setFrustumPerspective(116.0f, (settings.getWidth() / settings.getHeight()));
 //this.flyByCam.setMoveSpeed(100);
 this.flyByCam.setEnabled(false);
}

/**
 * @return the cam
 */
public Camera getCam()
{
 return cam;
}

/**
 * @return the chaseCam
 */
public ChaseCamera getChaseCam()
{
 return chaseCam;
}
}

```

—PlayerInputActionListener.java

```

package org.jmonkey3.chasecam;

import com.jme3.app.Application;
import com.jme3.app.state.AbstractAppState;
import com.jme3.app.state.AppStateManager;
import com.jme3.bullet.objects.PhysicsCharacter;
import com.jme3.input.KeyInput;
import com.jme3.input.controls.ActionListener;
import com.jme3.input.controls.KeyTrigger;

```

```
import com.jme3.scene.Spatial;

/*
Chase camera (aka 3rd person camera) example
Based on official TestQ3.java

@author Alex Cham aka Jcrypto
*/
public class PlayerInputActionListener extends AbstractAppState implements InputListener {
 private final PhysicsCharacter physicBody;
 // ...
 private boolean leftward = false;
 private boolean rightward = false;
 private boolean forward = false;
 private boolean backward = false;
 private boolean jump = false;
 private final AvatarAnimationEventListerner aael;
 /** ...

 @param pc
 @param avatar
 */
 public PlayerInputActionListener(PhysicsCharacter pc, Spatial a) {
 this.physicBody = pc;
 this.aael = new AvatarAnimationEventListerner(this, this.phy
 }

 @Override
 public void initialize(AppStateManager stateManager, Application a) {
 stateManager.attach(this.aael);
 //
 app.getInputManager().addMapping("LEFTWARD", new KeyTrigger());
 app.getInputManager().addMapping("RIGHTWARD", new KeyTrigger());
 app.getInputManager().addMapping("FORWARD", new KeyTrigger());
 app.getInputManager().addMapping("BACKWARD", new KeyTrigger());
 app.getInputManager().addMapping("JUMP", new KeyTrigger(KeyTrigger.KEYBOARD));
 }
}
```

```
 app.getInputManager().addListener(this, "LEFTWARD");
 app.getInputManager().addListener(this, "RIGHTWARD");
 app.getInputManager().addListener(this, "FORWARD");
 app.getInputManager().addListener(this, "BACKWARD");
 app.getInputManager().addListener(this, "JUMP");
 //
}

/**
@param binding
@param keyPressed
@param tpf
*/
public void onAction(String binding, boolean keyPressed, float
{

 if (binding.equals("LEFTWARD"))
 {

 this.leftward = this.aael.getAnimHelper().leftward(keyP

 } else if (binding.equals("RIGHTWARD"))
 {

 this.rightward = this.aael.getAnimHelper().rightward(ke

 } else if (binding.equals("FORWARD"))
 {

 this.forward = this.aael.getAnimHelper().forward(keyPre

 } else if (binding.equals("BACKWARD"))
 {

 this.backward = this.aael.getAnimHelper().backward(ke

 } else if (binding.equals("JUMP"))
 {

 this.jump = this.aael.getAnimHelper().jump(keyPressed);
```

```
 }

 }

 /**
 * @return the leftward
 */
 public boolean isLeftward()
 {
 return this.leftward;
 }

 /**
 * @return the rightward
 */
 public boolean isRightward()
 {
 return this.rightward;
 }

 /**
 * @return the forward
 */
 public boolean isForward()
 {
 return this.forward;
 }

 /**
 * @return the backward
 */
 public boolean isBackward()
 {
 return this.backward;
 }

 /**
 * @return the jump
 */
 public boolean isJump()
```

```
{
 return this.jump;
}
}
```

—SceneBodyManager.java

```
package org.jmonkey3.chasecam;

import com.jme3.app.Application;
import com.jme3.app.state.AppStateManager;
import com.jme3.asset.AssetManager;
import com.jme3.scene.Node;

/*
Chase camera (aka 3rd person camera) example
Based on official TestQ3.java

@author Alex Cham aka Jcrypto
*/
public class SceneBodyManager extends AbstractPhysicsBodyContext
{

 private final ScenePhysicsBodyContext spbc;
 private final SceneSpatialBodyContext ssbc;

 /**
 * @param stateManager
 * @param am
 * @param rootNode
 */
 public SceneBodyManager(AppStateManager stateManager, AssetManager am, Node rootNode)
 {

 this.ssbc = new SceneSpatialBodyContext(am, rootNode);
 this.spbc = new ScenePhysicsBodyContext(ssbc.getScene());
 }
}
```

```
 @Override
 public void initialize(AppStateManager stateManager, Application application)
 {
 //PhysicsSpace Initialization
 attachBulletAppstate(stateManager);
 //
 stateManager.attach(this.ssbc);
 stateManager.attach(this.spbc);
 }
}
```

## —ScenePhysicBodyContext.java

```
package org.jmonkey3.chasecam;

import com.jme3.app.Application;
import com.jme3.app.state.AppStateManager;
import com.jme3.bullet.control.RigidBodyControl;
import com.jme3.scene.Node;

/*
Chase camera (aka 3rd person camera) example
Based on official TestQ3.java

@author Alex Cham aka Jcrypto
*/
public class ScenePhysicBodyContext extends AbstractPhysicBodyContext
{
 private final RigidBodyControl rigidBodyControl;
 private final Node scene;

 /**
 * @param scene
 */
 public ScenePhysicBodyContext(Node scene)
```

```

{
 this.scene = scene;
 this.rigidBodyControl = new RigidBodyControl(.0f);
}

@Override
public void initialize(AppStateManager stateManager, Application application) {
 //
 //Add scene to PhysicsSpace
 System.out.println(this.getClass().getName() + ".getBulletAppState()");
 scene.addControl(rigidBodyControl);
 getBulletAppState().getPhysicsSpace().addAll(scene);
}

}

```

### —SceneSpatialBodyContext.java

```

package org.jmonkey3.chasecam;

import com.jme3.app.Application;
import com.jme3.app.state.AppStateManager;
import com.jme3.asset.AssetManager;
import com.jme3.asset.plugins.ZipLocator;
import com.jme3.light.AmbientLight;
import com.jme3.light.DirectionalLight;
import com.jme3.math.Vector3f;
import com.jme3.scene.Node;

/*
Chase camera (aka 3rd person camera) example
Based on official TestQ3.java

@author Alex Cham aka Jcrypto
*/
public class SceneSpatialBodyContext extends AbstractSpatialBodyContext {

```

```
private final Node rootNode;
//
private final Node scene;
private AmbientLight ambient;
private DirectionalLight sun;
/**/

@param am
@param rootNode
*/
public SceneSpatialBodyContext(AssetManager am, Node rootNode)
{
 this.rootNode = rootNode;
 //
 am.registerLocator("town.zip", ZipLocator.class);
 this.scene = (Node) am.loadModel("main.scene");
 this.ambient = new AmbientLight();
 this.sun = new DirectionalLight();
}

@Override
public void initialize(AppStateManager stateManager, Application application)
{
 //Main Scene loading

 this.scene.setLocalScale(0.1f);
 this.scene.scale(32.0f);
 //
 this.sun.setDirection(new Vector3f(1.4f, -1.4f, -1.4f));
 this.scene.setLocalTranslation(Vector3f.ZERO);
 //

 rootNode.attachChild(this.scene);
 rootNode.addLight(this.ambient);
 rootNode.addLight(this.sun);
}

/***
@return the scene
*/
```

```
 */
public Node getScene()
{
 return scene;
}
```

—TheGame.java

```
package org.jmonkey3.chasecam;

import com.jme3.app.SimpleApplication;

/*
Chase camera (aka 3rd person camera) example
Based on official TestQ3.java

@author Alex Cham aka Jcrypto
*/
public class TheGame extends SimpleApplication
{

 private ApplicationContext applicationContext;

 public TheGame()
 {
 }

 //
 public static void main(String[] args)
 {
 TheGame game = new TheGame();
 game.setShowSettings(false);
 game.start();
 }

 @Override
 public void simpleInitApp()
 {
 this.applicationContext = new ApplicationContext(stateManager);
 //
 stateManager.attach(applicationContext);
 }
}
```

—Debug.java

```
package org.jmonkey.utils;
```

```
import com.jme3.animation.AnimControl;
import com.jme3.asset.AssetManager;
import com.jme3.material.Material;
import com.jme3.math.ColorRGBA;
import com.jme3.math.Vector3f;
import com.jme3.scene.Geometry;
import com.jme3.scene.Node;
import com.jme3.scene.debug.Arrow;
import com.jme3.scene.debug.Grid;
import com.jme3.scene.debug.SkeletonDebugger;
import com.jme3.scene.shape.Line;
import static org.jmonkey.utils.SpatialUtils.makeGeometry;

/*
Chase camera (aka 3rd person camera) example
Based on official TestQ3.java

@author Alex Cham aka Jcrypto
*/
public class Debug
{

 public static void showNodeAxes(AssetManager am, Node n, float
 {
 Vector3f v = new Vector3f(axisLen, 0, 0);
 Arrow a = new Arrow(v);
 Material mat = new Material(am, "Common/MatDefs/Misc/Unshad
 mat.setColor("Color", ColorRGBA.Red);
 Geometry geom = new Geometry(n.getName() + "XAxis", a);
 geom.setMaterial(mat);
 n.attachChild(geom);

 //
 v = new Vector3f(0, axisLen, 0);
 a = new Arrow(v);
 mat = new Material(am, "Common/MatDefs/Misc/Unshaded.j3md")
 mat.setColor("Color", ColorRGBA.Green);
 geom = new Geometry(n.getName() + "YAxis", a);
```

```

 geom.setMaterial(mat);
 n.attachChild(geom);

 //

 v = new Vector3f(0, 0, axisLen);
 a = new Arrow(v);
 mat = new Material(am, "Common/MatDefs/Misc/Unshaded.j3md")
 mat.setColor("Color", ColorRGBA.Blue);
 geom = new Geometry(n.getName() + "ZAxis", a);
 geom.setMaterial(mat);
 n.attachChild(geom);
 }

 //

 public static void showVector3fArrow(AssetManager am, Node n, V
 {
 Arrow a = new Arrow(v);
 Material mat = MaterialUtils.makeMaterial(am, "Common/MatDe
 Geometry geom = makeGeometry(a, mat, name);
 n.attachChild(geom);
 }

 public static void showVector3fLine(AssetManager am, Node n, Ve
 {
 Line l = new Line(v.subtract(v), v);
 Material mat = MaterialUtils.makeMaterial(am, "Common/MatDe
 Geometry geom = makeGeometry(l, mat, name);
 n.attachChild(geom);
 }

 //Skeleton Debugger
 public static void attachSkeleton(AssetManager am, Node player,
 {
 SkeletonDebugger skeletonDebug = new SkeletonDebugger("skel
 Material mat2 = new Material(am, "Common/MatDefs/Misc/Unsha
 mat2.setColor("Color", ColorRGBA.Yellow);
 mat2.getAdditionalRenderState().setDepthTest(false);
 skeletonDebug.setMaterial(mat2);
 player.attachChild(skeletonDebug);
 }
}

```

```
}

///
```

```
public static void attachWireFrameDebugGrid(AssetManager assetM
{
 Geometry g = new Geometry("wireFrameDebugGrid", new Grid(si
 Material mat = new Material(assetManager, "Common/MatDefs/M
 mat.getAdditionalRenderState().setWireframe(true);
 mat.setColor("Color", color);
 g.setMaterial(mat);
 g.center().move(pos);
 n.attachChild(g);
}
```

```
}
```

—MaterialUtils.java

```

package org.jmonkey.utils;

import com.jme3.asset.AssetManager;
import com.jme3.material.Material;
import com.jme3.math.ColorRGBA;

/*
Chase camera (aka 3rd person camera) example
Based on official TestQ3.java

@author Alex Cham aka Jcrypto
*/
public class MaterialUtils
{

 public MaterialUtils()
 {
 }

 //Common/MatDefs/Misc/Unshaded.j3md"
 public static Material makeMaterial(AssetManager am, String nam
 {
 Material mat = new Material(am, name);
 mat.setColor("Color", color);
 return mat;
 }
}

```

## —SpatialUtils.java

```

package org.jmonkey.utils;

import com.jme3.material.Material;
import com.jme3.math.Vector3f;
import com.jme3.scene.Geometry;
import com.jme3.scene.Mesh;
import com.jme3.scene.Node;

```

```
/**

 * @author java
 */

public class SpatialUtils
{
 //
 public static Node makeNode(String name)
 {
 Node n = new Node(name);
 return n;
 }

 //
 public static Geometry makeGeometry(Mesh mesh, Material mat, St
 {
 Geometry geom = new Geometry(name, mesh);
 geom.setMaterial(mat);
 return geom;
 }

 //
 public static Geometry makeGeometry(Vector3f loc, Vector3f scl,
 {
 Geometry geom = new Geometry(name, mesh);
 geom.setMaterial(mat);
 geom.setLocalTranslation(loc);
 geom.setLocalScale(scl);
 return geom;
 }
}
```



## title: What's an IDE?

# What's an IDE?

- IDE stands for Integrated Development Environment. It's a software tool for developers.
- NetBeans IDE, Eclipse, IntelliJ are examples of development environments (=IDEs) for developers who are writing Java applications.
- Java is a programming language.
  - “Java Beans” is just a funny way of naming a certain type of data object that you can write in Java.  
(Java = coffee... coffee beans, get it? No, developers aren't very funny.)  
*Java Beans have nothing to do with this topic or NetBeans!*
- jMonkeyEngine is a game engine written in Java. You use it by putting a JAR library on the class path, and calling its API from your Java code.
- The jMonkeyEngine [SDK](#) (Software Development Kit) is a customized NetBeans IDE that has special tools that help you develop 3D Java games.

The writing of the game (the actual game development) is still up to you, the person using these tools. There is an expectation that you, the person using these tools, already know how to program in Java.

</div>

## The Past: A World Without IDEs

Let's say you have no IDE. The typical stuff that you need for game development is:

1. Install the Java SDK ("JDK 6") from Sun/Oracle.
  - The JDK includes essential development tools, such as the Java compiler (`javac`) and Java runtime (`java`).
2. Install the jMonkeyEngine JAR libraries.
  - These libraries include the special classes that you need for game development.
  - As with all Java applications, you put JAR libraries on the classpath. (`java -cp "bla.jar;foo.jar" myapp ...`)
3. Get a text editor to write `.java` and `.xml` files.
4. Get a 3D model editor to preview 3D models and arrange them in scenes.
5. Create Java project:
  - i. create directories for Java packages,
  - ii. create more directories for textures and sound files and 3D models,
  - iii. write an Ant build script,
  - iv. move JAR files around and check the classpath...
6. Write code:
  - i. write code in text editor,
  - ii. look up javadoc in the web browser,
  - iii. compile and then run code in the terminal,
  - iv. when you get error output in the terminal, go find the file and line back in text editor...
7. Repeat.

Basically, you switch back and forth between terminal, 3D model editor, web browser, and text editor a lot. You have to repeat lots of manual fine-tuning for every new file and project.

Some people got annoyed by these maintenance tasks, and that's why they invented the IDE.

</div>

## The Present: The world with IDEs

An IDE is a source code editor that developers use to write applications and manage projects. It replaces the text editor – and all the switching between browser and terminal and text editors. The essential word here is **integrated**: An IDE integrates all development tools

(listed above) in one window.

- There are New Project and New File menu items that create directories and packages. It creates a professional Ant script. It manages the classpath for you. These formerly manual tasks are always the same for you – now they are automated in the “Project window”.
- The “Editor window” lets you edit Java and XML files. (more details below)
- You can drag and drop commonly used code snippets from a “Palette window” into your Java files.
- A “Navigator window” gives you a quick overview of long Java classes, and lets you jump to methods and fields.
- The Build button starts the JDK6 compiler (`ant` and `javac`), the Clean button runs `ant clean`, the Run button runs the application (`ant run` and `java`) – all these tasks are now once-click actions in a toolbar or context menu.
- After building and running, the IDE opens the “Application window”. This window is what end-users of your applications will see. Here you can test your gameplay.

The **Editor** is the heart of the IDE, and it has tons of great additional capabilities:

- The IDE tries to compile in the background what you write in the Editor. If you made a horrible, but obvious, mistake (forgot semicolon, mixed up data types, made a typo in a method call...) it tells you so immediately through warning colors and icons. This is called syntactic and semantic code highlighting.
  - You still get Terminal output for errors and warnings (in the “Output window” inside the IDE), but this way you eradicate tiny typos and compiletime errors immediately, and you can focus on serious runtime errors in the Output window.
- The number of commands in the Java API is limited. So while you type a method or class name, there is only a limited number of things it can be. If you temporarily forgot what a method was called, the Editor pops up a list of options (plus javadoc comments), and you can simply select it.
- Similarly, every class and method only has a limited number of arguments. Again, the IDE pops up a list of expected arguments, so you don't need to search for them in the javadoc.

## Your Future: A World With jMonkeyEngine SDK

The jMonkeyEngine SDK is the same as NetBeans IDE, plus

- The New Project Wizards automatically adds the jMonkeyEngine libraries on the classpath and creates a build script.
- The javadoc popup displays Standard Java and jMonkeyEngine APIs in the editor.

- The Palette contains special code snippets from the jMonkeyEngine [API](#) for loading and saving 3D objects, input handling, nodes, lights, materials, rotation constants, etc.
- The Projects, SceneComposer, and SceneExplorer windows let you convert, preview, and arrange 3D models before you load them in your Java code.
- And more...



You see how such a unique IDE can speed up your development process drastically, it's worth giving it a try!

- [Video: jMonkeyEngine3 - Intro](#)
- [jMonkeyEngine SDK - the Comic](#)

</div>

# Intermediate

Tutorials for intermediate users. (Maybe rename to "Essentials". It should basically be gamedev basics. Mostly other stuff that you can't do without, whereas the advanced section is more of a "cherry pick the topics specific to your game".

## title: jMonkeyEngine3 Supported File Types

# jMonkeyEngine3 Supported File Types

</div>

## jMonkeyEngine3 File Formats

| Suffix | Usage                                                                                                                                                                                                                                                             | Learn more                                                             |
|--------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------|
| .j3o   | Binary 3D model or scene. At the latest from the Beta release of your game on, you should convert all models to .j3o format. During alpha and earlier development phases (when models still change a lot) you can alternatively load OgreXML/OBJ models directly. | <a href="#">Model Loader and Viewer</a>                                |
| .j3m   | A custom Material. You can create a .j3m file to store a Material configuration for a Geometry (e.g. 3D model).                                                                                                                                                   | <a href="#">Materials Overview</a><br><a href="#">Material Editing</a> |
| .j3md  | A Material definition. These are pre-defined templates for shader-based Materials.<br>Each custom .j3m Material is based on a material definition.<br>Advanced users can create their own material definitions.                                                   | <a href="#">Materials Overview</a>                                     |
| .j3f   | A custom post-processor filter configuration. You can create a .j3f file to store a FilterPostProcessor with a set of preconfigured filters.                                                                                                                      | <a href="#">Filters Effects Overview</a>                               |

</div>

## Supported External File Types

| <b>File Suffix</b>  | <b>Type</b> | <b>Description</b>                               |
|---------------------|-------------|--------------------------------------------------|
| .mesh.xml, .meshxml | 3D model    | Ogre Mesh XML                                    |
| .scene              | 3D scene    | Ogre DotScene                                    |
| .OBJ, .MTL          | 3D model    | Wavefront                                        |
| .blend              | 3D model    | Blender version 2.49 onwards (tested up to 2.65) |
| COLLADA             | 3D model    | Imported via Blender bundled with the SDK        |
| 3DS                 | 3D model    | Imported via Blender bundled with the SDK        |

|                  |             |                                       |
|------------------|-------------|---------------------------------------|
| .JPG, .PNG, .GIF | image       | Textures, icons                       |
| .DDS             | image       | Direct Draw Surface texture           |
| .HDR             | image       | High Dynamic Range texture            |
| .TGA             | image       | Targa Image File texture              |
| .PFM             | image       | Portable Float Map texture            |
| .fnt             | bitmap font | AngelCode font for <u>GUI</u> and HUD |
| .WAV             | audio       | Wave music and sounds                 |
| .OGG             | audio       | OGG Vorbis music and sounds           |

&lt;/div&gt;

## title: jME3 Application Display Settings

# jME3 Application Display Settings

Every class that extends `jme3.app.SimpleApplication` has properties that can be configured by customizing a `com.jme3.system.AppSettings` object.

Configure application settings in `main()`, before you call `app.start()` on the application object. If you change display settings during runtime, for example in `simpleInitApp()`, you must call `app.restart()` to make them take effect.

**Note:** Other runtime settings are covered in [SimpleApplication](#).

</div>

## Code Samples

Specify settings for a game (here called `MyGame`, or whatever you called your `SimpleApplication` instance) in the `main()` method before the game starts:

```
public static void main(String[] args) {
 AppSettings settings = new AppSettings(true);
 settings.setResolution(640,480);
 // ... other properties, see below
 MyGame app = new MyGame();
 app.setSettings(settings);
 app.start();
}
```

Set the boolean in the `AppSettings` constructor to true if you want to keep the default settings for values that you do not specify. Set this parameter to false if you want the application to load user settings from previous launches. In either case you can still customize individual settings.

This example toggles the settings to fullscreen while the game is already running. Then it restarts the game context (not the whole game) which applies the changed settings.

```

public void toggleToFullscreen() {
 GraphicsDevice device = GraphicsEnvironment.getLocalGraphicsEnvironment().getLocalDisplayMode();
 int i=0; // note: there are usually several, let's pick the first
 settings.setResolution(modes[i].getWidth(), modes[i].getHeight());
 settings.setFrequency(modes[i].getRefreshRate());
 settings.setBitsPerPixel(modes[i].getBitDepth());
 settings.setFullscreen(device.isFullScreenSupported());
 app.setSettings(settings);
 app.restart(); // restart the context to apply changes
}

```



## Properties

| Settings Property (Video)                                                                                                                          | Description                                                                                                                                                                                                                                                                                                                                                                                                                      | De          |
|----------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------|
| <b>setRenderer</b> (AppSettings.LWJGL_OPENGL1)<br><b>setRenderer</b> (AppSettings.LWJGL_OPENGL2)<br><b>setRenderer</b> (AppSettings.LWJGL_OPENGL3) | Switch Video Renderer to OpenGL 1.1, OpenGL 2, or OpenGL 3.3. If your graphic card does not support all OpenGL2 features ( <code>UnsupportedOperationException</code> : GLSL and OpenGL2 is required for the LWJGL renderer ), then you can force your SimpleApplication to use OpenGL1 compatibility. (Then you still can't use special OpenGL2 features, but at least the error goes away and you can continue with the rest.) | Open        |
| <b>setBitsPerPixel</b> (32)                                                                                                                        | Set the color depth.<br>1 bpp = black and white, 2 bpp = gray,<br>4 bpp = 16 colors, 8 bpp = 256 colors, 24 or 32 bpp = "truecolor".                                                                                                                                                                                                                                                                                             | 24          |
| <b>setFramerate</b> (60)                                                                                                                           | How often per second the engine should try to refresh the frame. For the release, usually 60 fps. Can be lower (30) if you need to free up the CPU for other applications. No use setting it to a higher value than                                                                                                                                                                                                              | -1<br>(unli |

|                                                         |                                                                                                                                                                                                                                                                                                                                                                                         |                     |
|---------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------|
|                                                         | the screen frequency! If the framerate goes below 30 fps, viewers start to notice choppiness or flickering.                                                                                                                                                                                                                                                                             |                     |
| setFullscreen(true)                                     | <p>Set this to true to make the game window fill the whole screen; you need to provide a key that calls <code>app.stop()</code> to exit the fullscreen view gracefully (default: escape).</p> <p>Set this to false to play the game in a normal window of its own.</p>                                                                                                                  | False<br>(windowed) |
| setHeight(480), setWidth(640)<br>setResolution(640,480) | Two equivalent ways of setting the display resolution.                                                                                                                                                                                                                                                                                                                                  | 640×480 pixel       |
| setSamples(4)                                           | <p>Set multisampling to 0 to switch antialiasing off (harder edges, faster.)</p> <p>Set multisampling to 2 or 4 to activate antialiasing (softer edges, may be slower.)</p> <p>Depending on your graphic card, you may be able to set multisampling to higher values such as 8, 16, or 32 samples.</p>                                                                                  | 0                   |
| setVSync(true)<br>setFrequency(60)                      | <p>Set vertical syncing to true to time the frame buffer to coincide with the refresh frequency of the screen. VSync prevents ugly page tearing artefacts, but is a bit slower; recommended for release build.</p> <p>Set VSync to false to deactivate vertical syncing (faster, but possible page tearing artifacts); can remain deactivated during development or for slower PCs.</p> | false<br>60 fp      |
| setStencilBits(8)                                       | Set the number of stencil bits. This value is only relevant when the stencil buffer is being used. Specify 8 to indicate an 8-bit stencil buffer, specify 0 to disable the stencil buffer.                                                                                                                                                                                              | 0<br>(disabled)     |
| setDepthBits(16)                                        | Sets the number of depth bits to use.<br>The number of depth bits specifies the precision of the depth buffer. To increase precision, specify 32 bits. To                                                                                                                                                                                                                               | 24                  |

decrease precision, specify 16 bits. On some platforms 24 bits might not be supported, in that case, specify 16 bits.

| Settings Property (Input)          | Description                                                                                                                                                                                                | Default     |
|------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------|
| setUseInput(false)                 | Respond to user input by mouse and keyboard. Can be deactivated for use cases where you only display a 3D scene on the canvas without any interaction.                                                     | true        |
| setUseJoysticks(true)              | Activate optional joystick support                                                                                                                                                                         | false       |
| setEmulateMouse(true)              | Enable or disable mouse emulation for touchscreen-based devices. Setting this to true converts taps on the touchscreen to clicks, and finger swiping gestures over the touchscreen into mouse axis events. | false       |
| setEmulateMouseFlipAxis(true,true) | Flips the X or Y (or both) axes for the emulated mouse. Set the first parameter to true to flip the x axis, and the second to flip the y axis.                                                             | false,false |

| Settings Property (Audio)                  | Description                                                                                                                                                                                                      | Default |
|--------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------|
| setAudioRenderer(AppSettings.LWJGL_OPENAL) | Switch Audio Renderer. Currently there is only one option.                                                                                                                                                       | OpenAL  |
| setStereo3D(true)                          | Enable 3D stereo. This feature requires hardware support from the GPU driver. See <a href="#">Quad Buffering</a> . Currently, your everyday user's hardware does not support this, so you can ignore it for now. | false   |

| Settings Property (Branding)                                                      | Description                                                                                                                                                                                                                                                      |          |
|-----------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------|
| <code>setTitle("My Game")</code>                                                  | This string will be visible in the titlebar, unless the window is fullscreen.                                                                                                                                                                                    | "jMonke  |
| <code>setIcons(new BufferedImage[]{<br/>ImageIO.read(new File(""), ...)});</code> | This specifies the little application icon in the titlebar of the application (unused in MacOS?). You should specify the icon in various sizes (256,128,32,16) to look good on various operating systems. Note: This is not the application icon on the desktop. | null     |
| <code>setSettingsDialogImage("Interface/mysplashscreen.png")</code>               | A custom splashscreen image in the assets/Interface directory which is displayed when the settings dialog is shown.                                                                                                                                              | "/com/jr |

You can use `app.setShowSettings(true);` and `setSettingsDialogImage("Interface/mysplashscreen.png")` to present the user with jme3's default display settings dialog when starting the game. Use `app.setShowSettings(false);` to hide the default settings screen. Set this boolean before calling `app.start()` on the SimpleApplication.

</div>

## Toggling and Activating Settings

| SimpleApplication method     | Description                                                                                                                                                                                                                                                                              |
|------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| app.setShowSettings(boolean) | Activate or deactivate the default settings screen before start()ing the game. If you let users use this screen, you do not need to modify the settings object. Note: Most developers implement their own custom settings screen, but the default one is useful during the alpha stages. |
| app.setSettings(settings)    | After you have modified the properties on the settings object, you apply it to your application. Note that the settings are not automatically reloaded while the game is running.                                                                                                        |
| app.start()                  | Every game calls start() in the beginning to initialize the game and apply the settings. Modify and set your settings before calling start().                                                                                                                                            |
| app.restart()                | Restart()ing a running game restarts the game context and applies the updated settings object. (This does not restart or reinitialize the whole game.)                                                                                                                                   |

&lt;/div&gt;

## Saving and Loading Settings

An AppSettings object also supports the following methods to save your settings under a unique key (in this example “com.foo.MyCoolGame3”):

- Use `settings.save("com.foo.MyCoolGame3")` to save your settings via standard java.io serialization.
- Use `settings.load("com.foo.MyCoolGame3")` to load your settings.
- Use `settings2.copyFrom(settings)` to copy a settings object.

Usage:

Provide the unique name of your jME3 application as the String argument. For example  
`com.foo.MyCoolGame3` .

```
try { settings.save("com.foo.MyCoolGame3"); }
catch (BackingStoreException ex) { /* could not save settings */ }
```

- On Windows, the preferences are saved under the following registry key:  
`HKEY_CURRENT_USER\Software\JavaSoft\Prefs\com\foo\MyCoolGame3`
- On Linux, the preferences are saved in an XML file under:  
`$HOME/.java/.userPrefs/com/foo/MyCoolGame3`

- On Mac OS X, the preferences are saved as XML file under:  
\$HOME/Library/Preferences/com.foo.MyCoolGame3.plist

</div>

## **title: SimpleApplication**

# **SimpleApplication**

The base class of the jMonkeyEngine3 is `com.jme3.app.SimpleApplication`. Your first game's Main class extends SimpleApplication directly. When you feel confident you understand the features, you will typically extend SimpleApplication to create a custom base class for the type of games that you want to develop.

SimpleApplication gives you access to standard game features, such as a scene graph (rootNode), an asset manager, a user interface (guiNode), input manager, audio manager, a physics simulation, and a fly-by camera. You call `app.start()` and `app.stop()` on your game instance to start or quit the application.

For each game, you (directly or indirectly) extend SimpleApplication exactly once as the central class. If you need access to any SimpleApplication features from another game class, make the other class extend [AbstractAppState](#) (don't extend SimpleApplication once more).

The following code sample shows the typical base structure of a jME3 game:

```
import com.jme3.app.SimpleApplication;

public class MyBaseGame extends SimpleApplication {

 public static void main(String[] args){
 MyBaseGame app = new MyBaseGame();
 app.start();
 }

 @Override
 public void simpleInitApp() {
 /* Initialize the game scene here */
 }

 @Override
 public void simpleUpdate(float tpf) {
 /* Interact with game events in the main loop */
 }

 @Override
 public void simpleRender(RenderManager rm) {
 /* (optional) Make advanced modifications to frameBuffer and
 */
 }
}
```

Let's have a look at the [API](#) of the base class.

</div>

## Application Class

Internally, com.jme3.app.SimpleApplication extends com.jme3.app.Application. The Application class represents a generic real-time 3D rendering jME3 application (i.e., not necessarily a game). Typically, you do not extend com.jme3.app.Application directly to create a game.

| <b>Application class fields</b>     | <b>Purpose</b>                                                                                                                                                                                                                                                     |
|-------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| viewPort<br>getViewPort()           | The view object for the default camera. You can register advanced <a href="#">post-processor filters</a> here.                                                                                                                                                     |
| settings<br>setSettings()           | Use this <a href="#">AppSettings</a> object to specify the display width and height (by default 640×480), color bit depth, z-buffer bits, anti-aliasing samples, and update frequency, video and audio renderer, asset manager. See: <a href="#">AppSettings</a> . |
| cam<br>getCamera()                  | The default <a href="#">camera</a> provides perspective projection, 45° field of view, near plane = 1 wu, far plane = 1000 wu.                                                                                                                                     |
| assetManager<br>getAssetManager()   | An object that manages paths for loading models, textures, materials, sounds, etc. By default the <a href="#">Asset Manager</a> paths are relative to your project's root directory.                                                                               |
| audioRenderer<br>getAudioRenderer() | This object gives you access to the jME3 <a href="#">audio</a> system.                                                                                                                                                                                             |
| listener<br>getListener()           | This object represents the user's ear for the jME3 <a href="#">audio</a> system.                                                                                                                                                                                   |
| inputManager<br>getInputManager()   | Use the <a href="#">inputManager</a> to configure your custom inputs (mouse movement, clicks, key presses, etc) and set mouse pointer visibility.                                                                                                                  |
| stateManager<br>getStateManager()   | You use the Application's state manager to activate <a href="#">AppStates</a> , such as <a href="#">Physics</a> .                                                                                                                                                  |

| <b>Application methods</b> | <b>Purpose</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|----------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| setPauseOnLostFocus(true)  | Set this boolean whether the game loop should stop running when ever the window loses focus (typical for single-player game). Set this to false for real-time and multi-player games that keep running.                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| start()                    | Call this method to start a jME3 game. By default this opens a new jME3 window, initializes the scene, and starts the event loop.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| restart()                  | Loads modified <a href="#">AppSettings</a> into the current application context.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| stop()                     | Stops the running jME3 game and closes the jME3 window.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| start(Type.Headless) etc   | <p>Switch Context com.jme3.system.JmeContext.Type when starting the application:</p> <p>Type.Display – jME application runs in a window of its own. (This is the default.)</p> <p>Type.Canvas – jME application is embedded in a <a href="#">Swing Canvas</a>.</p> <p>Type.Headless – jME application runs its event loop without calculating any view and without opening any window. Can be used for a <a href="#">Headless Server</a> application.</p> <p>Type.OffscreenSurface – jME application view is not shown and no window opens, but everything calculated and cached as bitmap (back buffer) for use by other applications.</p> |

| Internal class field/method                                       | Purpose                                                                                                                                                                                                |
|-------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| context<br>getContext()                                           | The application context contains the renderer, <a href="#">AppSettings</a> , timer, etc. Typically, you do not directly access the context object.                                                     |
| inputEnabled                                                      | this internal boolean is true if you want the system to listen for user inputs, and false if you just want to play a non-interactive scene. You change the boolean using <a href="#">AppSettings</a> . |
| keyInput,<br>mouseInput<br>joyInput, touchInput                   | Default input contexts for keyboard, mouse, and joystick. Internally used to enable handling of joysticks or touch devices. The base classes contain key and mouse button enums.                       |
| renderManager<br>getRenderManager()<br>renderer<br>getRenderer(); | Low-level and high-level rendering interface. Mostly used internally.                                                                                                                                  |
| guiViewPort<br>getGuiViewPort()                                   | The view object for the orthogonal <a href="#">GUI</a> view. Only used internally for <a href="#">HUDs</a> .                                                                                           |
| timer                                                             | An internal update loop timer, don't use. See <code>tpf</code> in <code>simpleUpdate()</code> below to learn about timers.                                                                             |
| paused                                                            | Boolean is used only internally during runtime to pause/unpause a game. (You need to implement your own <code>isRunning</code> boolean or so.)                                                         |

## SimpleApplication Class

The com.jme3.app.SimpleApplication class extends the generic com.jme3.app.Application class. SimpleApplication makes it easy to start writing a game because it adds typical functionality:

- First-person (fly-by) camera
- Scene graph that manages your models in the rendered 3D scene.
- Useful default input mappings (details below.)

Additional to the functionality that Application brings, SimpleApplication offers the following methods and fields that can be used, for example, inside the `simpleInitApp()` method:

| <b>SimpleApplication Class Field</b> | <b>Purpose</b>                                                                                                                                                        |
|--------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| rootNode<br>get rootNode()           | The root node of the scene graph. Attach a <a href="#">Spatial</a> to the rootNode and it appears in the 3D scene.                                                    |
| guiNode<br>get GuiNode()             | Attach flat GUI elements (such as <a href="#">HUD</a> images and text) to this orthogonal <a href="#">GUI</a> node to make them appear on the screen.                 |
| flyCam<br>get FlyByCamera()          | The default first-person fly-by camera control. This default camera control lets you navigate the 3D scene using the preconfigured WASD and arrow keys and the mouse. |

| <b>SimpleApplication Method</b> | <b>Purpose</b>                                                                                                                                                                                                                              |
|---------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| loadStatsView();                | Call this method to print live statistic information to the screen, such as current frames-per-second and triangles/vertices counts. You use this info typically only during development or debugging.                                      |
| loadFPSText();                  | Call this method to print the current framerate (frames per second) to the screen.                                                                                                                                                          |
| setDisplayFps(false);           | A default SimpleApplication displays the framerate (frames per second) on the screen. You can choose to deactivate the FPS display using this command.                                                                                      |
| setDisplayStatView(false);      | A default SimpleApplication displays mesh statistics on the screen using the com.jme3.app.StatsView class. The information is valuable during the development and debugging phase, but for the release, you should hide the statistics HUD. |

| SimpleApplication Interface                             | Purpose                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|---------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>public void simpleInitApp()</code>                | Override this method to initialize the game scene. Here you load and create objects, attach Spatial to the rootNode, and bring everything in its starts position. See also <a href="#">Application States</a> for best practices.                                                                                                                                                                                                                                                                                                                                                                 |
| <code>public void simpleUpdate(float tpf)</code>        | Override this method to hook into the <a href="#">update loop</a> , all code you put here is repeated in a loop. Use this loop to poll the current game state and respond to changes, or to let the game mechanics generate encounters and initiate state changes. Use the float <code>tpf</code> as a factor to time actions relative to the <i>time per frame</i> in seconds: <code>tpf</code> is large on slow PCs, and small on fast PCs.<br>For more info on how to hook into the <a href="#">update loop</a> , see <a href="#">Application States</a> and <a href="#">Custom Controls</a> . |
| <code>public void simpleRender(RenderManager rm)</code> | <b>Optional:</b> Advanced developers can override this method if the need to modify the frameBuffer and scene graph directly.                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |

Use `app.setShowSettings(true);` to present the user with a splashscreen and the built-in display settings dialog when starting the game; or use `app.setShowSettings(false);` to hide the buil-in screen (in this case, you may want to provide a custom splashscreen and settings panel). Set this boolean before calling `app.start()` in the `main()` method of the SimpleApplication. See also [AppSettings](#).

</div>

## Default Input Mappings

The following default navigational input actions are mapped by the default `flyCam` control in a SimpleApplication: You can use these mappings for debugging and testing until you implement custom [input handling](#).

| Key                     | Action                                                                        |
|-------------------------|-------------------------------------------------------------------------------|
| <code>KEY_ESCAPE</code> | Quits the game by calling <code>app.stop()</code>                             |
| <code>KEY_C</code>      | Debug key: Prints camera position, rotation, and direction to the out stream. |
| <code>KEY_M</code>      | Debug key: Prints memory usage stats the out stream.                          |
| <code>F5</code>         | Hides or shows the statistics the bottom left.                                |

As long as the `flyCam` is enabled, the following so-called “WASD” inputs, including MouseLook, are available:

| Camera Motion          | Key or Mouse Input                                        |
|------------------------|-----------------------------------------------------------|
| Move Forward           | KEY_W                                                     |
| Move Left (Strafe)     | KEY_A                                                     |
| Move Backward          | KEY_S                                                     |
| Move Right (Strafe)    | KEY_D                                                     |
| Move Vertical Upward   | KEY_Q                                                     |
| Move Vertical Downward | KEY_Z                                                     |
| Rotate Left            | KEY_LEFT, or move mouse horizontally left (-x)            |
| Rotate Right           | KEY_RIGHT, or move mouse horizontally right (+x)          |
| Rotate Up              | KEY_UP, or move mouse vertically forward (+y)             |
| Rotate Down            | KEY_DOWN, or move mouse vertically backward (-y)          |
| Rotate                 | BUTTON_LEFT, or hold left mouse button and drag to rotate |
| Zoom In                | AXIS_WHEEL, or scroll mouse wheel backward                |
| Zoom Out               | AXIS_WHEEL, or scroll mouse wheel forward                 |

## Defaults and Customization

By default, a SimpleApplication displays Statistics (`new StatsAppState()`), has debug output keys configured (`new DebugKeysAppState()`), and enables the flyCam (`new FlyCamAppState()`). You can customize which you want to reuse in your SimpleApplication.

The following example shows how you can remove one of the default AppStates, in this case, the FlyCamAppState:

- Either, in your application's constructor, you create the SimpleApplication with only the AppStates you want to keep:

```
public MyAppliction() {
 super(new StatsAppState(), new DebugKeysAppState());
}
```

- Or, in the `simpleInitApp()` method, you remove the ones you do not want to keep:

```
public void simpleInitApp() {
 stateManager.detach(stateManager.getState(FlyCamAppState.c
 . . .
```



[display](#), [basegame](#), [documentation](#), [intro](#), [intermediate](#), [init](#), [input](#), [game](#), [loop](#), [rootnode](#),  
[application](#), [simpleapplication](#)

</div>

## **title: Best Practices For jME3 Developers**

# **Best Practices For jME3 Developers**

Every milestone of a game development project is made up of phases: Planning, development, testing, and release. Every milestone involves updates to multi-media assets and to code.

This “best practices” page is a collection of recommendations and expert tips. Feel free to add your own!

## **Requirements and Planning**

If you are a beginner, you should first [read some articles about game development](#). We cannot cover all general tips here.

## **Requirements Gathering**

As a quick overview, answer yourself the following questions:

- Motivation
  - Sum up your game idea in one catchy sentence. If you can't, it's too complicated.  
E.g. “Craft by day, fight by night!”
  - Who's the target group? Are you making it for your friends or are you trying to attract the masses?
- Game type
  - Point of view (first- or third-person camera)? What characters does the player control? (if applicable)
  - Time- or turn-based?
  - Genre (drama, horror, adventure, mystery, comedy, educational, documentary)?
  - Setting and background story? (historic, fantasy, anime, futuristic, utopia/dystopia, pirate, zombie, vampire...)?
- Gameplay
  - What is the start state, what is the end state? (if applicable)
  - What resources does the player manage? How are resources gained, transformed, spent?

- E.g. “points”, health, speed, gold, xp, mana.
- How does the player interact? Define rules, challenges, game mechanics.
- What state is considered winning, and what losing, or is it an open world?
- Multi-media assets
  - Which media will you need? How will you get this content?  
E.g. models, terrains; materials, textures; noises, music, voices; video, cutscenes; spoken/written dialog; level maps, quests, story; AI scripts.
- Interface
  - Can you achieve a high degree of input control? (Even minor navigation and interaction glitches make the game unsolvable.)
  - Decide how to reflect current status, and changes in game states. E.g. health/damage in HUD.
  - Decide how to reward good moves and discourage bad ones.

## Planning Development Milestones

Use an [issue and bug tracker](#) to outline what features you want and what components are needed.

1. Pre-Alpha Development
  - Artwork: Test asset loading and saving with mock-ups and stock art.
  - Lay out the overall application flow, i.e. switching between intro / options / game screen, etc.
  - Get one typical level working before you can announce the Alpha Release.  
E.g. if the game is a “Jump’n’Run”, jumping and running must work.
2. Alpha Release
3. Pre-Beta Development
  - Artwork: Replace all mock-ups with first drafts of real media and level maps.
  - Have your team members review and “alpha test” it on various systems, track bugs, debug, optimize.
  - Declare [Feature Freeze](#) before you announce the Beta Release to prevent a bottomless pit of new bugs.
4. Beta Release
5. Post-Beta Development
  - Artwork: Fill in the final media and level maps.
  - Have external people review and “beta test” it, make it easy to report bugs.
  - Fix high-priority bugs, even out the kinks in code and gameplay, don’t add new features for now!
6. Gamma Release, Delta Release... = Release Candidates
  - Think you’re done? Make test runs incl. packaging and distribution. (Order form? download?)

- Test the heck out of it. Last chance to find and fix a horrible bug.

## 7. Omega = Final Release

How you name or number these stages is fully up to your team. Development teams use numbered milestones (m1, m2, m3), Greek letters (e.g. alpha, beta, gamma, delta), "major.minor.patch-build" version numbering (e.g. "2.7.23-1328"), or combinations thereof.

## Use File Version Control

Whether you work in a team or alone, keeping a version controlled repository of your code will help you roll-back buggy changes, or recover old code that someone deleted and that is now needed again.

- Treat commit messages as messages to your future self. "Made some changes" is *not* a commit message.
- The jMonkeyEngine SDK supports Subversion, Mercurial, and Git.  
If you don't know which to choose, Subversion is a good choice for starters.
- Set up your own local server, or get free remote hosting space from various open-source dev portals like [Sourceforge](#), [Github](#), [bitbucket](#) (supports private projects), [Java.net](#), [Google Code](#)...

## Multi-Media Asset Pipeline

| DO                                                                | DON'T                                                                                                                      |
|-------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------|
| Save original models+textures into assets/Textures .              | Don't reference textures or models outside your JME project.                                                               |
| Save sounds into assets/Sounds .                                  | Don't reference audio files outside your JME project.                                                                      |
| Create simple, low-polygon models.                                | Don't create high-polygon models, they render too slow to be useful in games.                                              |
| Only use Diffuse Map, Normal Map, Glow Map, Specular Map.         | Don't use unsupported material properties that are not listed in the <a href="#">Materials Overview</a> .                  |
| Use UV texture / texture atlases / baking for each texture map.   | Don't create models based on multiple separate textures, it will break the model into separate meshes.                     |
| Convert Models to j3o format. Move j3o files into assets/Models . | Don't reference Blender/Ogre/OBJ files in your load() code, because these unoptimized files are not packaged into the JAR. |

Learn details about the [Multi-Media Asset Pipeline](#) here.

# Development Phase

Many game developers dream of creating their very own “MMORPG with full-physics, AI, post-rendering effects, multi-player networking, procedurally generated maps, and customizable characters”. So why aren’t there tons of MMORPGs out there?

Even for large experienced game producers, the creation of such a complex game is time-intensive and failure-prone. How familiar are you with multi-threading, persistence, optimization, client-server synchronization, ...? Unless your answer is “very!”, then start with a single-player desktop game, and work your way up – just as the pros did when they started.

</div>

## Extend SimpleApplication

Every jME3 game is centered around one main class that (directly or indirectly) extends com.jme3.app.[SimpleApplication](#).

Note that although the “SimpleApplication” name might be misleading, all jME3 applications, including very large projects, are based on this class. The name only implies that this class itself is a simple application already. You make it “non-simple” by extending it!

For your future game releases, you will want to rely on your own framework (based on jME): Your custom framework extends jME’s SimpleApplication, and includes your custom methods for loading, saving, and arranging your scenes, your custom navigation methods, your inputs for pausing and switching your custom screens, your custom user interface (options screen, HUD, etc), your custom NPC factory, your custom physics properties, your custom networking synchronization, etc.

Writing and reusing (extending) your own base framework saves you time. When you update your generic base classes, all your games that extend them benefit from improvements to the base (just as all jME-based games benefit of improvements to the jME framework). Also, your own framework gives all your games a common look and feel.

## Where to Start?

You have a list of features that you want in game, but which one do you implement first? You will keep adding features to a project that grows more and more complex, how can you minimize the amount of rewriting required?

1. Make sure the game’s high-level frame (screen switching, network sync, loading/saving) is sound and solid.
2. Start with implementing the most complex game feature first – the one that imposes

most constraints on the structure of your project (for example: multi-player networking, or physics.)

3. Add only one larger feature at a time. If there are complex interactions (such as “networking + physics”), start with a small test case (“one shared cube”) and work your way up. Starting with a whole scene introduces too many extra sources of error.
4. Implement low-complexity decorations (audio and visual effects) last.
5. Test for side-effects on existing code after you add a new feature (regression test).

Acknowledge whether you want a feature because it is necessary for gameplay, or simply because “everyone else has it”. Your goal should be to bring out the essence of your game idea. Don’t water down gameplay by attempting to make it “do everything, but better”. Successful high-performance games are the ones where someone made smart decisions what to keep and what to *drop*.

## The Smart Way to Add Custom Methods and Fields

**Avoid the Anti-Pattern:** Don't design complex role-based classes using Java inheritance, it will result in an unmaintainable mess.

Example: You start extending `Node` → `MyMobileNode` → `MyNPC`. Then you extend `MyFighterNPC` (defends, attacks) and `MyShopKeeperNPC` (trades) from `MyNPC`. What if you need an NPC that trades and defends itself, but doesn't attack? Do you extend `MyShopKeeperNPC` and copy and paste the defensive methods from `MyFighterNPC`? Or do you extend `MyFighterNPC` and override the attacking methods of its parent? Neither is a clean solution.

Wouldn't it be better if behaviours were a separate “system”, and attributes were separate “components” that you add to the “entity” that needs them?

You write Java classes named `Controls` to implement your Game Entities, and define an Entity's visuals, attributes, and behaviours. In jME, `Spatial`s (`Nodes` or `Geometry`s) are the visual representation of the game entity in the scene graph.

- Game entities have **attributes** – All Entities are neutral *things*, only their attributes define what an entity actually *is* (a person or a brick). In jME, we call these class fields of Spatial “user data”.

Example: Players have **class fields** for `id`, `health`, `coins`, `inventory`, `equipment`, `profession`.

- Game entities have **behaviours** – Behaviour systems communicate about the game state and modify attributes. In jME, these game mechanics are implemented in modular `update()` methods that all hook into the main update loop.

Example: Players have **methods** such as `walk()`, `addGold()`, `getHealth()`, `pickUpItem()`, `dropItem()`, `useItem()`, `attack()`.

**Follow the Best Practice:** In general, use composition over inheritance and keep “what an

entity does" (behaviour system) separate from "what this entity is" (attributes).

- Use `setUserData()` to add custom attributes to Spatial.
- Use [Controls](#) and [Application States](#) to define custom behaviour systems.

If your game is even more complex, you may want to learn about "real" Entity Systems, which form a quite different programming paradigm from object oriented coding but are scalable to very large proportions. Note however that this topic is very unintuitive to handle for an OOP programmer and you should really decide on a case basis if you really need this or not and gather some experiences before diving head first into a MMO project 

- <http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/>
- [http://www.gamasutra.com/blogs/MeganFox/20101208/88590/Game\\_Engines\\_101\\_The\\_EntityComponent\\_Model.php](http://www.gamasutra.com/blogs/MeganFox/20101208/88590/Game_Engines_101_The_EntityComponent_Model.php)
- <http://gamedev.stackexchange.com/questions/28695/variants-of-entity-component-systems>
- <http://louisstowasser.com/post/19279778476/entity-component-systems-inheritance-vs-composition>
- <http://t-machine.org/index.php/2012/03/16/entity-systems-what-makes-good-components-good-entities/>
- <http://entity-systems.wikidot.com/>

## The Smart Way to Access Game Features

[SimpleApplication](#) gives you access to game features such as the `rootNode`, `assetManager`, `guiNode`, `inputManager`, `audioManager`, `physicsSpace`, `viewPort`, and the camera. But what if you need this access also from another class? Don't extend `SimpleApplication` a second time, and don't pass around tons of object references in constructors! Needing access to application level objects is a sign that this class should be designed as an [AppState](#) (read details there).

An `AppState` has access to all game features in the `SimpleApplication` via the `this.app` and `this.stateManager` objects. Examples:

```
Spatial sky = SkyFactory.createSky(this.app.getAssetManager(), "sky
...
this.app.getRootNode().attachChild(sky);
```

## The Smart Way to Implement Game Logic

As your SimpleApplication-based game grows more advanced, you find yourself putting more and more interactions in the `simpleUpdate()` loop, and your `simpleInitApp()` methods grows longer and longer. It's a best practice to move blocks of game mechanics into reusable component classes of their own. In jME3, these reusable classes are called `Controls` and `AppStates`.

- Use [AppStates](#) to implement *global game mechanics*.
  - Each AppState calls its own `initialize()` and `cleanup()` methods when it is attached to or detached from the game.
  - Each AppState runs its own *thread-safe* `update()` loop that hooks into the main `simpleUpdate()` loop.
  - You specify what happens if an AppState is paused/unpaused.
  - You can use an AppState to switch between sets of AppStates.
  - An AppState has access to everything in the SimpleApplication (`rootNode`, `AssetManager`, `StateManager`, `InputListener`, `ViewPort`, etc).
- Use [Controls](#) to implement the *behaviour of game entities*.
  - Controls add a type of behaviour (methods and fields) to an individual Spatial (a player, an NPC).
  - Each Control runs its own *thread-safe* `controlUpdate()` loop that hooks into the main `simpleUpdate()` loop.
  - One Spatial can be influenced by several Controls. (!)
  - Each Spatial needs its own instance of the Control.
  - A Control only has control over and access to the spatial that it is attached to (and its sub-spatial).

A game contains algorithms that do not directly affect spatial (for example, AI pathfinding code that calculates and chooses paths, but does not actually move spatial). You do not need to put such non-spatial code in controls, you can run these things in a new thread. Only the transformation code that actually modifies the spatial must be called from a control, or must be `enqueue()`ed.

Controls and AppStates often work together: An AppState can reach up to the application and `get` all Spatial from the `rootNode` that carry a specific Control, and perform a global action on them. Example: In BulletPhysics, all physical Spatial that carry RigidBodyControls are steered by the overall BulletAppState.

AppStates and Controls are extensions to a SimpleApplication. They are your modular building blocks to build a more complex game. In the ideal case, you move all init/update code into Controls and AppStates, and your `simpleInitApp()` and `simpleUpdate()` could end up virtually empty. This powerful and modular approach cleans up your code considerably. Read all about [Custom Controls](#) and [Application States](#) here.

</div>

## Optimize Application Performance

- [Optimization](#) – How to avoid wasting cycles
- [Multithreading](#) – Use concurrency for long-running background tasks, but don't manipulate the scene graph from outside the main thread (update loop)!
- You can add a [Java Profiler](#) to the jMonkeyEngine SDK via Tools → Plugins → Available. The profiler presents statistics on the lifecycle of methods and objects. Performance problems may be caused by just a few methods that take long, or are called too often (try to cache values to avoid this). If object creation and garbage collection counts keep increasing, you are looking at a memory leak.

</div>

## Don't Mess With Geometric State

**These tips are especially important for users who already know jME2.** Automatic handling of the Geometric State has improved in jME3, and it is now a best practice to *not* mess with it.

- Do not call `updateGeometricState()` on anything but the root node!
- Do not override or mess with `updateGeometricState()` at all.
- Do not use `getLocalTranslation().set()` to move a spatial in jME3, always use  
`setLocalTranslation()`.

## Maintain Internal Documentation

It's unlikely you will fully document *every* class you write, we hear you. However, you should at least write meaningful javadoc to provide context for your most crucial methods/parameters.

- What is this? How does it solve its task (input, algorithm used, output, side-effects)?
- Write down implicit limits (e.g. min/max values) and defaults while you still remember.
- In which situation do I want to use this, is this part of a larger process? Is this step required, or what are the alternatives?

Treat javadoc as messages to your future self. “`genNextVal()` generates the next value” and “`@param float factor A factor influencing the result`” do *not* count as documentation.

</div>

## Debugging and Test Phase

**A Java Debugger** is included in the jMonkeyEngine SDK. It allows you to set a break point in your code near the line of code where an exception happens. Then you step through the execution line by line and watch object and variable states live, to detect where the bug starts.

**Use the Logger** to print status messages during the development and debugging phase, instead of `System.out.println()`. The logger can be switched off with one line of code, whereas commenting out all your `println()`s takes a while.

**Unit Testing (Java Assertions)** has a different status in 3D graphics development than in other types of software. You cannot write assertions that automatically test whether the rendered image *looks* correct, or whether interactions are *intuitive*. Still you should [create simple test cases](#) for individual game features such as loaders, content generators, effects. Run the test cases now and then to see whether they still work as intended – or whether they are suffering from regressions or side-effects. Keep the test classes in the `test` directory of your project, don't include them in the distribution.

**Quality Assurance (QA)** means repeatedly checking a certain set of features that must work, but that might be unexpectedly broken as a side-effect. Every game has some crazy bugs somewhere – but basic tasks *must work*, no excuse. This includes installing and de-installing; saving and loading; changing options; starting, pausing, quitting; basic actions such as walking, fighting, etc. After every milestone, you go through your QA list again and systematically look for regressions or newly introduced bugs. Check the application *on every supported operating system and hardware* (!) because not all graphic cards support the same features. If you don't find the obvious bugs, your users will, and carelessness will put them off.

**Alpha and Beta Testing** means that you ask someone to try to install and run your game. It should be a real user situation, where they are left to figure out the installation and gameplay by themselves—you only can include the usual read-me and help docs. Provide the testers with an easy method to report back what problems they encountered, what they liked best, or why they gave up. Evaluate whether reported problems are one-off glitches, or whether they must be fixed for the game to be playable for everyone.

</div>

## Release Phase

</div>

### Pre-Release To-Do List

- Prepare a web page, a cool slogan, advertisements, etc
- Verify that all assets are up-to-date and converted to .j3o.
- Verify that your code loads the optimized .j3o files, and not the original model formats.
- Prepare licenses of assets that you use for inclusion. (You *did* obtain permission to use them, right...?)
- Switch off fine [logging](#) output.
- Prepare promotional art: The most awesome screenshots (in thumbnail, square, vertical, horizontal, and fullscreen formats) and video clips. Include name, contact info, slogan, etc., so future customers can find you.
- Prepare a readme.txt file, or installation guide, or handbook – if applicable.
- Get a certificate if one is required for your distribution method (see below).
- Specify a [classification rating](#) (needed for e.g. app stores).

</div>

## Distributing the Executables

The [jMonkeyEngine SDK helps you with deployment](#): You specify your branding and deployment options in the Project Properties dialog, and then choose Clean and Build from the context menu. **If you use another IDE, consult this IDE's documentation.**

Decide whether you want to release your game as WebStart, desktop JAR, mobile APK, or browser Applet – Each has its pros and cons.

| Distribution                            | Pros                                                                                                                                                                       | Cons                                                                                                                                                            |
|-----------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Desktop Launcher (.EXE, .app, .jar+.sh) | This is the standard way of distributing desktop applications. The jMonkeyEngine SDK can be configured to automatically create zipped launchers for each operating system. | You need to offer three separate, platform-dependent downloads.                                                                                                 |
| Desktop Application (.JAR)              | Platform independent desktop application.                                                                                                                                  | User must have Java configured to run JARs when they are opened; or user must know how to run JARs from command line; or you must provide a custom JAR wrapper. |
| Web Start (.JNLP)                       | The user accesses a <u>URL</u> , saves the game as one executable file. Easy process, no installer required. You can allow the game to be played offline.                  | Users need network connection to install the game. Downloading bigger games takes a while as opposed to running them from a CD.                                 |
| Browser Applet (.HTML+.JAR)             | Easy to access and play game via most web browsers. Userfriendly solution for quick small games.                                                                           | Game only runs in the browser. Game or settings cannot be saved to disk. Some restrictions in default camera navigation (jME cannot capture mouse.)             |
| Android (.APK)                          | Game runs on Android devices.                                                                                                                                              | Android devices do not support post-procesor effects.                                                                                                           |

Which ever method you choose, a Java-Application works on the main operating systems: Windows, Mac OS, Linux, Android.

The distribution appears in a newly generated `dist` directory inside your project directory. These are the files that you upload or burn to CD to distribute to your customers.

---

See also:

- [gamedev.net: Developing Your Game Concept By Making A Design Document](#)

</div>

## **title:**

Overview of Jmonkey Engine Structure and Function

## title: How to Use Materials

# How to Use Materials

A Geometry (mesh) is just the shape of the object. jMonkeyEngine cannot render a shape without knowing anything about its surface properties. You need to apply a color or texture to the surface of your Geometries to make them visible. In jMonkeyEngine, colors and textures are represented as Material objects. (An alternative would also be to load a model that includes materials generated by a mesh editor, such as Blender.)

- All Geometries must have Materials that defines color or texture.
- Each Material is based on a Material Definition file.

Examples of included Material Definitions: Lighting.j3md, Unshaded.j3md

You want to make the most of your 3D models by specifying good looking material parameters. The developers must be in contact with the graphic designers regarding which of the [Material properties](#) they intend to use in their 3D models. You must have an understanding what [texture maps](#) are, to be able to use texture-mapped materials.

Don't forget to add a [Light Source](#) to the scene! All Materials (except "Unshaded" ones) are **invisible** without a light source.

If you want more advanced background info: You can learn more about [Material Definitions](#) in general here. You can find the full list of Material Parameters in the [Material Definitions Properties](#) overview. The following sections introduce you to the most commonly used cases. You typically initialize Material objects in the `simpleInitApp()` method, and configure them using the setters described here. Then load the Materials using

```
myGeometry.setMaterial(mat).
</div>
```

## Code Sample

The following samples assume that you loaded a Geometry called `myGeometry`, and want to assign a material to it.

This example creates a simple unshaded blue material: Use it for abstract objects that need no illumination/shading, e.g. sky, [GUI](#) and billboard nodes, tiles/cards, or toons.

```
Spatial myGeometry = assetManager.loadModel("Models/Teapot/Teapot.j3o");
Material mat = new Material(assetManager, // Create new material and
 "Common/MatDefs/Misc/Unshaded.j3md"); // ... specify .j3md file
mat.setColor("Color", ColorRGBA.Blue); // Set some parameters,
myGeometry.setMaterial(mat); // Use new material on the geometry
```

This example creates a [Phong](#)-illuminated blue material. Use it for illuminated, naturalistic objects, such as characters, buildings, terrains, vehicles. Needs a light source, otherwise it will be invisible.

```
Spatial myGeometry = assetManager.loadModel("Models/Teapot/Teapot.j3o");
Material mat = new Material(assetManager, // Create new material and
 "Common/MatDefs/Light/Lighting.j3md"); // ... specify .j3md file
mat.setBoolean("UseMaterialColors",true); // Set some parameters,
mat.setColor("Ambient", ColorRGBA.Blue); // ... color of this object
mat.setColor("Diffuse", ColorRGBA.Blue); // ... color of light being emitted
myGeometry.setMaterial(mat); // Use new material on the geometry
```

Do you reuse Materials? You can [store Material properties in a .j3m file](#) and load all properties in one line using

```
myGeometry.setMaterial(assetManager.loadMaterial("Materials/myMaterial.j3m"));
```

</div>

## Colored or Textured

Every Material must have at least Material Colors or Textures. Some optional material features also require a combination of both.

### Colored

To give an unshaded material a color:

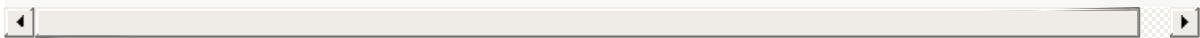
1. Specify the color property

```
mat.setColor("Color", ColorRGBA.Blue); // with Unshaded.j3md
```

To give an Phong-illuminated material a color:

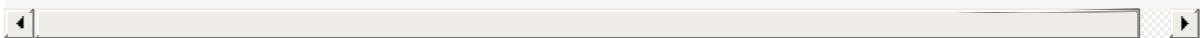
1. Activate material colors:

```
mat.setBoolean("UseMaterialColors", true); // with Lighting.j3md
```



2. Specify at least Diffuse and Ambient colors. Set both to the same color in the standard case.

```
mat.setColor("Diffuse", ColorRGBA.Blue); // with Lighting.j3md
mat.setColor("Ambient", ColorRGBA.Blue); // with Lighting.j3md
```

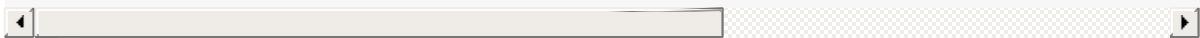


## Textured

To give an unshaded material a texture:

- Specify at least a ColorMap:

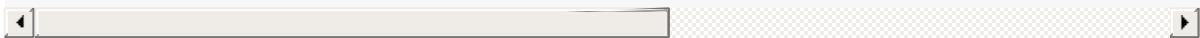
```
mat.setTexture("ColorMap", assetManager.loadTexture("Textures/m
```



To give a Phong-illuminated material a texture:

- Specify at least the DiffuseMap texture:

```
mat.setTexture("DiffuseMap", assetManager.loadTexture("Textures.
```



It can happen that you load textures at different scales, for example, your blades of grass may look as big as twigs, or your spaceship's heat tiles may look like small bathroom tiles. Then you need to adjust the texture scale, either bigger (< 1.0f) or smaller (< 1.0f).

```
geometry.scaleTextureCoordinates(new Vector2f(0.5f, 0.5f));
```

All other Texture Maps or Material settings are optional. If used skillfully, they make your model look really spiffy.

</div>

## (Optional) Bumpy

A NormalMap (also called BumpMap) is an extra colored texture that describes the fine bumpy details of the Material surface. E.g. fine cracks, pores, creases, notches. Using a BumpMap is more efficient than trying to shape the mesh to be bumpy.

To add a BumpMap (this only makes sense for illuminated Materials):

1. Generate normal vectors information for the Mesh (not for the Geometry!) using

```
com.jme3.util.TangentBinormalGenerator .
```

```
TangentBinormalGenerator.generate(mesh);
```

2. Specify the `NormalMap` texture for the Material.

```
mat.setTexture("NormalMap", assetManager.loadTexture("Textures/\\
```

[Learn more about creating and using NormalMaps and BumpMaps here.](#)

## (Optional) Shiny

To activate Shininess (this only makes sense for illuminated Materials):

1. Specify the `Shininess` intensity the Material.

Shininess is a float value between 1 (rough surface with blurry shininess) and 128 (very smooth surface with focused shininess)

```
mat.setFloat("Shininess", 5f);
```

2. Activate material colors:

```
mat.setBoolean("UseMaterialColors", true);
```

3. Specify the `Specular` and `Diffuse` colors of the shiny spot.

Typically you set Specular to the ColorRGBA value of the light source, often RGBA.White.

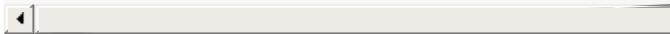
```
mat.setColor("Specular", ColorRGBA.White);
mat.setColor("Diffuse", ColorRGBA.White);
```

4. (Optional) Specify a `SpecularMap` texture.

You optionally hand-draw this grayscale texture to outline in detail where the surface should be more shiny (whiter grays) and where less (blacker grays). If you don't supply

a SpecularMap, the whole material is shiny everywhere.

```
mat.setTexture("SpecularMap", assetManager.loadTexture("Textures/alien_glow_glowmap.jpg"));
```



To deactivate shininess

- Set the `Specular` color to `ColorRGBA.Black`. Do not just set `Shininess` to 0.

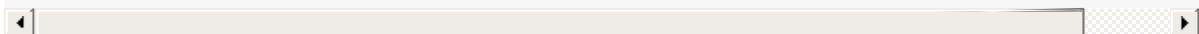
```
mat.setColor("Specular", ColorRGBA.Black);
```

## (Optional) Glow

To activate glow:

- Add one [BloomFilter PostProcessor](#) in your `simpleInitApp()` method (only once, it is used by all glowing objects).

```
FilterPostProcessor fpp=new FilterPostProcessor(assetManager);
BloomFilter bloom = new BloomFilter(BloomFilter.GlowMode.Objects);
fpp.addFilter(bloom);
viewPort.addProcessor(fpp);
```



- Specify a `Glow` color.

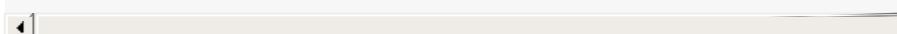
A `ColorRGBA` value of your choice, e.g. choose a warm or cold color for different effects, or white for a neutral glow.

```
mat.setColor("GlowColor", ColorRGBA.White);
```

- (Optional) Specify a `GlowMap` texture.

This texture outlines in detail where the `DiffuseMap` texture glows. If you don't supply a `GlowMap`, the whole material glows everywhere.

```
mat.setTexture("GlowMap", assetManager.loadTexture("Textures/alien_glow_glowmap.jpg"));
```



To deactivate glow:

- Set the `Glow` color to `ColorRGBA.Black`.

```
mat.setColor("GlowColor", ColorRGBA.Black);
```

Learn more about [Bloom and Glow](#).

## (Optional) Transparent

Most Material Definitions support an alpha channel to make a model opaque, translucent, or transparent.

- Alpha=1.0f makes the color opaque (default),
- Alpha=0.0f make the color fully transparent
- Alpha between 0f and 1f makes the color more or less translucent.

To make a Geometry transparent or translucent:

1. Specify which areas you want to be transparent or translucent by specifying the alpha channel:

- (For colored Materials) In any RGBA color, the first three are Red-Green-Blue, and the last float is the Alpha channel. For example, to replace ColorRGBA.Red with a translucent red:

```
mat.setColor("Color", new ColorRGBA(1,0,0,0.5f));
```

- (For textured Materials) Supply an AlphaMap that outlines which areas are transparent.

```
mat.setTexture("AlphaMap", assetManager.loadTexture("Texture
```

- (For textured Materials) If the DiffuseMap has an alpha channel, use:

```
mat.setBoolean("UseAlpha", true);
```

2. Specify BlendMode Alpha for the Material.

```
mat.getAdditionalRenderState().setBlendMode(BlendMode.Alpha);
```

3. Put the Geometry (not the Material!) in the appropriate render queue bucket.

Objects in the translucent bucket (e.g. particles) are not affected by SceneProcessors (e.g. shadows). Objects in the transparent bucket (e.g. foliage) are affected by SceneProcessors (e.g. shadows).

```
geo.setQueueBucket(Bucket.Translucent);
```

```
geo.setQueueBucket(Bucket.Transparent);
```

4. (Optional) Specify other material settings.

Standard Material Transparency	Description
getAdditionalRenderState().setBlendMode(BlendMode.Off);	This is the default, no transparency.
getAdditionalRenderState().setBlendMode(BlendMode.Alpha);	Interpolates the background pixel with the current pixel by using the current pixel's alpha.
getAdditionalRenderState().setDepthWrite(false);	Disables writing of the pixel's depth value to the depth buffer.
getAdditionalRenderState().setAlphaTest(true) getAdditionalRenderState().setAlphaFallOff(0.5f);	Enables Alpha Testing and uses an AlphaDiscardThreshold as alpha fall-off value. This means that gradients in the AlphaMap are no longer interpreted as soft translucency, but parts of the texture become either fully opaque or fully transparent. Only pixels above the alpha threshold (e.g. 0.5f) are rendered.

It is possible to load a DiffuseMap texture that has an Alpha channel, and combine it with an underlying Material Color.

```
mat.setBoolean("UseAlpha", true);
```

The Material Color bleeds through the transparent areas of the top-layer DiffuseMap texture. In this case you do not need BlendMode Alpha – because it's not the whole Material that is transparent, but only one of the texture layers. You use this bleed-through effect, for example, to generate differently colored uniforms, animals, or plants, where each Material uses the same “template” DiffuseMap texture but combines it with a different color.

</div>

## (Optional) Wireframe

Additionally to the above settings, you can switch off and on a wireframe rendering of the mesh. Since a wireframe has no faces, this temporarily disables the other Texture Maps.

Material Property	Description	Example
<code>getAdditionalRenderState().setWireframe(true);</code>	Switch to showing the (textured) Material in wireframe mode. The wireframe optionally uses the Material's <code>Color</code> value.	Use wireframes to debug meshes, or for a “matrix” or “holodeck” effect.

[material](#), [texture](#), [effect](#), [wireframe](#), [light](#), [documentation](#)

</div>

## **title: Math Cheat Sheet**

# **Math Cheat Sheet**

</div>

## **Formulas**

Note: Typically you have to string these formulas together. Look in the table for what you want, and what you have. If the two are not the same line, than you need conversion steps inbetween. E.g. if you have an angle in degrees, but the formula expects radians: 1) convert degrees to radians, 2) use the radians formula on the result.

I have...	I want...	Formula
normalized direction and length n1,d1	a vector that goes that far in that direction new direction vector v0	$v0 = n1.mult(d1)$
point and direction vector p1,v1	to move the point new point p0	$p0 = p1.add(v1)$
direction, position and distance v1,p1,dist	Position at distance p2	$v1.normalizeLocal()$ $scaledDir = v1.mult(dist)$ $p2 = p1.add(scaledDir)$
two direction vectors or normals v1,v2	to combine both directions new direction vector v0	$v0 = v1.add(v2)$
two points p1, p2	distance between two points new scalar d0	$d0 = p1.subtract(p2).length()$ $d0 = p1.distance(p2)$
two points p1, p2	the direction from p2 to p1 new direction vector v0	$v0 = p1.subtract(p2)$
two points, a fraction p1, p2, h=0.5f	the point "halfway" ( $h=0.5f$ ) between the two points new interpolated point p0	$p0 =$ <code>FastMath.interpolateLinear(h,p1,p2)</code>
a direction vector, an up vector v, up	A rotation around this up axis towards this direction new Quaternion q	<code>Quaternion q = new Quaternion();</code> <code>q.lookAt(v,up)</code>

I have...	I want...	Formula
angle in degrees a	to convert angle a from degrees to radians new float angle phi	phi = a / 180 * FastMath.PI; OR phi=a.mult(FastMath.DEG_TO_RAD);
angle in radians phi	to convert angle phi from radians to degrees new float angle a	a = phi * 180 / FastMath.PI
radian angle and x axis phi, x	to rotate around x axis new quaternion q0	q0.fromAngleAxis( phi, Vector3f.UNIT_X )
radian angle and y axis phi, y	to rotate around y axis new quaternion q0	q0.fromAngleAxis( phi, Vector3f.UNIT_Y )
radian angle and z axis phi, z	to rotate around z axis new quaternion q0	q0.fromAngleAxis( phi, Vector3f.UNIT_Z )
several quaternions q1, q2, q3	to combine rotations, in that order new quaternion q0	q0 = q1.mult(q2).mult(q3)
point and quaternion p1, q1	to rotate the point around origin new point p0	p0 = q1.mult(p1)
angle in radians and radius phi,r	to arrange or move objects horizontally in a circle (with y=0) x and z coordinates	float x = FastMath.cos(phi)*r; float z = FastMath.sin(phi)*r;

&lt;/div&gt;

## Local vs Non-local methods?

- Non-local method creates new object as return value, v remains unchanged.

```
v2 = v.mult(); v2 = v.add(); v2 = v.subtract(); etc
```

- Local method changes v directly!

```
v.multLocal(); v.addLocal(); v.subtractLocal(); etc
```

&lt;/div&gt;

## **title:**

MonkeyBlaster Demo

## **title: Optimization reference**

# **Optimization reference**

This page is intended as a reference collection of optimization tricks that can be used to speed up JME3 applications.

## **Maintain low Geometry count**

The more Geometry objects are added to the scene, the harder it gets to handle them in a speedy fashion. The reason for this is that a render command must be done for every object, potentially creating a bottleneck between the CPU and the graphics card.

### **Possible optimization techniques**

- Use `GeometryBatchFactory.optimize(node)` to merge the meshes of the geometries contained in the given node into fewer batches, each based on common Materials used. You can optimize nodes using the SceneComposer in the SDK as well: Right-click a node and select “Optimize Geometry”.

### **Side-effects**

- Using `GeometryBatchFactory` merges individual Geometries into a single mesh. Thereby it becomes hard to apply specific Materials or to remove a single Geometry. Therefore it should be used for static Geometry only that does not require frequent changes or individual materials/texturing.
- Using a [Texture Atlas](#) provides limited individual texturing of batched geometries.
- Using the still experimental `BatchNode` allows batching Geometry while keeping the single Geometry objects movable separately (similar to animation, the buffer gets updated per Geometry position).

## **Avoid creating new objects**

Different Java implementations use different garbage collection algorithms, so depending on the platforms you target, different advice applies.

The major variants are Oracle's JRE, old (pre-Gingerbread) Androids, and newer (Gingerbread or later) Androids.

Oracle's JRE is a copying collector. This means that it does not need to do any work for objects that have become unreachable, it just keeps copying live objects to new memory areas and recycles the now-unused area as a whole. Older objects are copied less often, so the garbage collection overhead is roughly proportional to the rate at which your code creates new objects that survive for, say, more than a minute.

Gingerbread and newer Androids use a garbage collector that does some optimization tricks with local variables, but you should avoid creating and forgetting lots of objects in the scene graph.

Older Androids use a very naive garbage collector that needs to do real work for every object, both during creation and during collection. Creating local variables can build up a heap of work, particularly if the function is called often.

To avoid creating a temporary object, use *local methods* to overwrite the contents of an existing object instead of creating a new temporary object for the result.

E.g. when you use math operations like `vectorA.mult(vectorB);`, they create new objects for the result.

Check your math operations for opportunities to use the *local* version of the math operations, e.g. `vectorA.multLocal(vectorB)`. Local methods store the result in `vectorA` and do not create a new object.

## Avoid large objects in physics

To offload much computation to the less CPU intense physics broadphase collision check, avoid having large meshes that cover e.g. the whole map of your level. Instead, separate the collision shapes into multiple smaller chunks. Obviously, don't exaggerate the chunking, because having excessive amounts of physics objects similarly cause performance problems.

## Check the Statistics

SimpleApplication displays a HUD with statistics. Use `app.setDisplayStatView(true);` to activate it, and false to deactivate it. The StatsView counts Objects,Uniforms,Triangles,Vertices are in the scene, and it counts how many FrameBuffers, Textures, or Shaders:

- ... were switched in the last frame (S)
- ... were used during the last frame (F)
- ... exist in OpenGL memory (M)

For example, `Textures (M)` tells you how many textures are currently in OpenGL memory.

Generally jME3 is well optimized and optimizes these things correctly. Read [statsview](#) to learn the details about how to interpret the statistics, how to tell whether your values are off, or whether they point out a problem.

[performance](#)

</div>

## title: PGI's Rolling Tracks

# PGI's Rolling Tracks

This article uses outdated jME3 [API](#), check the main [jME3 tutorials page](#) for up to date examples!



In this intermediate article, PGI explains the workflow of how he created a game with jME3. He covers model creation and loading, physics (collision detection), game logic, game stages, input management, audio, effects, and a custom UI solution. The article contains annotated code samples and many screenshots.

1. [Download the sample game](#) and try it
2. How was it done? Download and read the tutorial:  
[rollmadness.pdf](#) or  
[rollmadness.odt](#)

In the game, you are a pilot speeding down a crazy winding rollercoaster track! Your goal is to stay on the track and shoot as many targets as you can.

Back to [jME3 Tutorials](#)

[game](#), [intermediate](#)

</div>

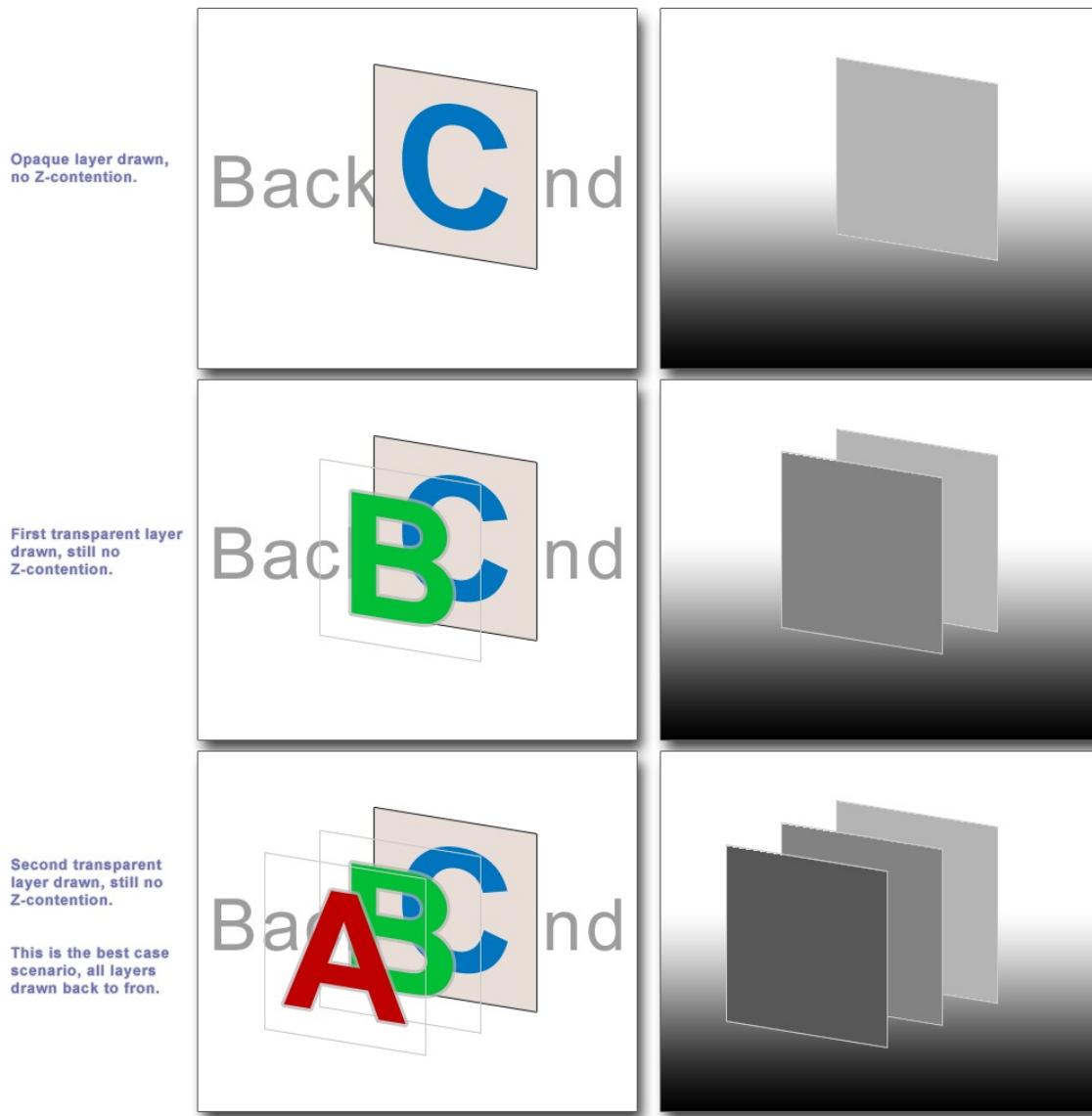
# title: Transparency Sorting

## Transparency Sorting

There is often a lot of confusion with how pixels get processed in relation to the z-buffer and why sorting is important. Most importantly it can be mind-warping trying to wrap one's head around how to 'fix' the cases where proper sorting isn't possible.

Hopefully these diagrams help show the futility of it all... I mean the trade offs one has to make to get transparency looking its best in the general case.

The first image I'll show is the 'best case' where all objects are drawn back to front. I'll then discuss briefly the steps JME goes through to try to make that happen.

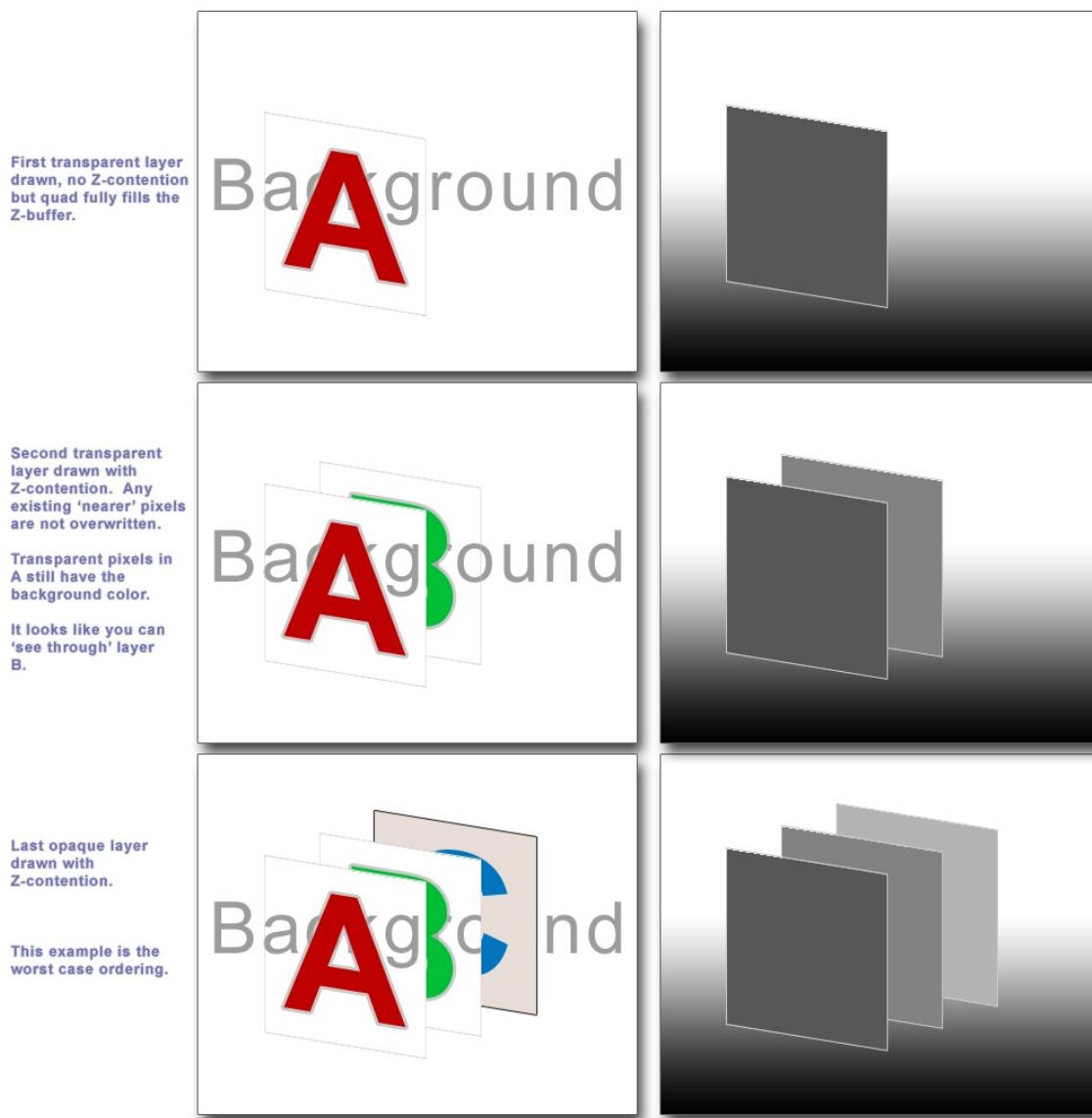


In JME, the opaque layer would generally be placed in the opaque bucket. All opaque layers are drawn first and within the ability to sort them, they are sorted front to back to prevent overdraw.

After the opaque bucket is drawn, the transparent bucket is drawn and it is sorted back to front. So in the ideal case it would appear exactly as in this picture.

Sorting is done at the object level and it can never be perfect. It is impossible to properly sort objects by distance in the general case even if they were just triangles. Simply imagine intersecting triangles and it's clear there is no proper order. JME tries its best.

Next I'll show an image of the worst case scenario to show why sorting is important and how your 'best friend' the z-buffer is really transparency's worst enemy.

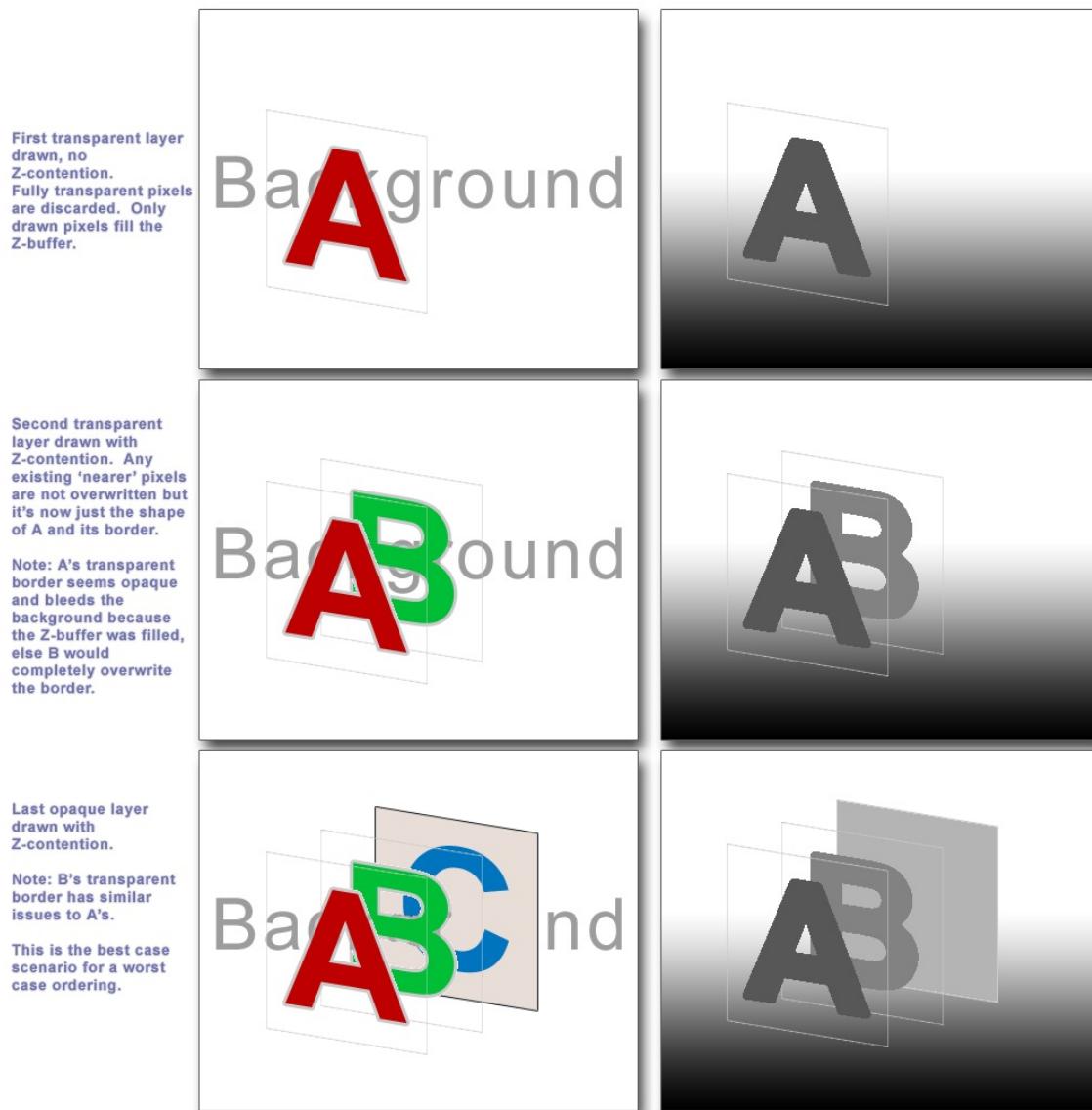


This is what will happen if you put all of your objects in the opaque buffer. JME will sort them front to back and you will get these strange 'windows' into your background.

Because sorting is done at the object level, you may have triangle to triangle overlap even within the same mesh if it is non-convex. Think of a glass donut where the near surface

triangles are drawn before the hole's triangles. There will be this same issue where they occlude the farther triangles. A different angle on the donut might produce the correct results depending on the order of the triangles in the mesh.

Finally, I'll augment the worst case sorting with something like `alphaDiscardThreshold` (or the old `alphaTest`/`alphaFalloff` values). In this example, let's pretend we only discard pixels with `alpha = 0`.



It's better but not perfect. Any partially transparent pixels will still show the issue. Partial transparency will drive you crazy if you let it.

Bottom line:

- back to front sorting would fix all issues
- accurate back to front sorting in the general case is impossible
- for purely on/off  $a = 1$  or  $a = 0$  transparency then a discard threshold is the best bet to mitigate sorting problems.

- Where  $0 < \text{alpha} < 1$ , improper sorting of triangles/pixels will always cause artifacts.
- </ul>

[transparent](#), [sorting](#), [bucket](#), [z-buffer](#), [alpha](#)

</div>

## **title: Recast Navigation**

# **Recast Navigation**

Building a Nav-Mesh for your JME game.

</div>

## **About**

</div>

## **Setup**

</div>

## **Example Code**

</div>

## **title: Changing the Name of Your APK/Application:**

### **Changing the Name of Your APK/Application:**

1. Open your project's properties and navigate to Application
2. Update the title

This has no real effect, however it keeps continuity throughout your app. Actually, this likely renamed the window created to display your app. So, now go change the actual name of your APK:

1. Select File View in the left pane of the SDK
2. Navigate to the mobile/res/values directory and open the strings.xml file
3. There should be a string tag with the following key pair: name="app\_name"
4. Replace MyGame with your app's name and save the file.
5. In File view, navigate to nbproject and open the project.properties file
6. Edit the value of application.title to reflect your game's name (unless step 1/2 above altered this for you)

</div>

### **Changing the APK Icon:**

1. Under the File view of your project navigate to mobile/res and add a “drawable” folder if one does not exist.
2. Add your icon file (png)
3. Open the Android Manifest file and add the following to your application tag:  
`android:icon="@drawable/<ICON FILE NAME WITHOUT EXTENSION>"`
4. If you would like multiple size icons, add the following folders:

```
drawable-hdpi (should contain the icon named the same at 72x72 pixels)
drawable-ldpi (should contain the icon named the same at 36x36 pixels)
drawable-mdpi (should contain the icon named the same at 48x48 pixels)
drawable-xhdpi (should contain the icon named the same at 96x96 pixels)
```

### **Adding a Splash Screen to your app:**

1. Open Android Main Activity, either through the Important Files list in Project view or in the File view under mobile/src/<package name>/ directory
2. Add the following line to the MainActivity method:

```
splashPicID = R.drawable.<IMAGE NAME WITHOUT EXTENSION>;
```

3. Add the image the the mobile/res/drawable directory

Compiling Google Play Services and Adding it to Your Project:

First get the api:

1. Download the google play services add-on through the SDK Manager under Extras (named Google Play services)
2. Copy the directory from where you downloaded it to another location (like JME Projects Folder)

```
</div>
```

## Compile the jar file for use with your project:

1. In the directory you copied, there is an android project file.
2. In JME's IDE, open this project
3. In the General section of the project properties, there is a list of potential Android target platforms. Select the one you are using for your project by clicking on the list (this is not intuitive at all, as the list looks like nothing more than info... not selectable items)
4. Under the Library section, click the checkbox that is labeled: Is Library
5. Click Ok and then Clean & Build this project.

This will compile the play services all proper like so you can add it to your project. Now, for that step:

1. Open your project's properties.
2. In the Libraries section, click the "Add JAR/folder" button.
3. Find and add the jar you compiled above (This can be found in: <COPIED DIR>\libproject\google-play-services\_lib\libs\google-play-services.jar)
4. Modify your Android Manifest by adding the following tags under application:  
`<meta-data android:name="com.google.android.gms.games.APP_ID"  
 android:value="@string/app_id" /> <meta-data  
 android:name="com.google.android.gms.version"  
 android:value="@integer/google_play_services_version"/>`
5. Add the following tag to your mobile/res/values/integers.xml file (create it if it doesn't exist):  
`<integer name="google_play_services_version">4323000</integer>`
6. Clean & Build your project

```
</div>
```

## Adding Play Games Services to Your Project:

1. Download the project from: <https://github.com/playgameservices/android-samples>
2. In the following directory, you find java files you will need to add to your project:

```
<DOWNLOAD DIR>\android-samples-master\BasicSamples\libraries\BaseGa
Grab GameHelper.java and GameHelperUtil.java and add them to the di
```

3. In the following directory, you find a resource file you will need to add to your project:

```
<DOWNLOAD DIR>\android-samples-master\BasicSamples\libraries\BaseGa
Grab the gamehelper_strings.xml into your mobile/res/values folder\
```

4. Add the following jar from the Adroid SDK folder to your project as a library:

```
<ANDROID SDK INSTALL DIR>\adt-bundle-windows-x86_64-20131030\sdk\ex
```

And this is the basics for setting this up.

```
</div>
```

## Adding AdMob Support to Your Project:

1. Open your Android Manifest and add the following tag update the application tag:  

```
<activity android:name="com.google.android.gms.ads.AdActivity"
 android:configChanges="keyboard|keyboardHidden|orientation|screenLayout|uiMode|screen
 Size|smallestScreenSize"/>
```
2. After the application tag, add the following tags:  

```
<uses-permission
 android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
```
3. In the onCreate method of your Main Activity, add the following snippet (configure however you like):

```

adView = new AdView(this);
adView.setAdSize(AdSize.FULL_BANNER);
adView.setAdUnitId("<WHATEVER AD UNIT ID YOU ARE ASSIGNED THE");
adView.buildLayer();
LinearLayout ll = new LinearLayout(this);
ll.setGravity(Gravity.BOTTOM);
ll.addView(adView);
addContentView(ll, new ViewGroup.LayoutParams(ViewGroup.LayoutParams.MATCH_PARENT, ViewGroup.LayoutParams.WRAP_CONTENT));

```

## Communication Between your Application & Main Activity:

1. Create an interface named something along the lines of JmeToHarness.java
2. Open your Android Main Activity and implement this interface.
3. In Main.java of your Application, add the following:

```

JmeToHarness harness;
public JmeToHarness getHarness() {
 return this.harness;
}
public void setHarnessListener(JmeToHarness harness) {
 this.harness = harness;
}

```

4. Add the following snippet to the onCreate method of your Android Main Activity:

```

if (app != null)
 ((Main)app).setHarnessListener(this);

```

5. Add error handling if you want it.

This bit is ultra useful for calling AdMob changes and Play Games methods (like updating achievements, leader boards, etc, etc)

**EDIT:** Keep this as generic as you possibly can as it should plug & play with iOS & Applets if you keep that in mind. Google Play Services/Play Games Services works for all of the above... soooo... anyways.

</div>

## Changing the Package Name After Project Creation:

1. Open the project properties of your Application 2. Navigate to Application > Android and edit the package name.

This does absolutely nothing, but help with consistency.

So, to actually change the package name, you will want to:

1. Open the Android Manifest
2. Edit the manifest tag key pair: package="<THE NEW PACKAGE NAME>"
3. In File view, navigate to nbproject and open the project.properties file
4. Edit the value of mobile.android.package

Take a moment or 4 to navigate through the directory structure in file view and remove any artifacts left from the previous package name build. Alternately, you can run Clean on the project prior to updating the package name.

</div>

## title: Animation in jME3

# Animation in jME3

In 3D games, you do not only load static 3D models, you also want to be able to trigger animations in the model from the Java code.

</div>

## Requirements

JME3 only loads and plays animated models, it does not create them.

What is required for an animated model? ([See also: Animation terminology](#))

1. For each model, you have to segment the model into a skeleton (**bone rigging**).
2. For each motion, you have to specify how the animation distorts parts of the model (**skinning**).
3. For each animation, you have to specify a series of snapshots of how the bones are positioned (**keyframes**).
4. One model can contain several animations. You give every animation a name when you save it in the mesh editor.

Unless you download free models, or buy them from a 3D artist, you must create your animated models in an **external mesh editor** (for example, Blender) yourself.

- [Converting Blender Models to JME3 \(.J3o files\)](#)
- [Video Series: Creating models in Blender, OgreMax, 3dsMax](#)
- [Video: Creating and Exporting OgreXML Animations from Blender 2.61 to JME3](#)
- [Scene Workflow: Exporting OgreXML scenes from Blender to JME3](#)
- [Animation Workflow: Create Animated UV-Mapped OgreXML Models in Blender, and use them in JME3](#)
- [Video: Creating Worlds with Instances in Blender](#)

What is required in your JME3-based Java class?

- One Animation Control per animated model
- As many Animation Channels per Control as you need to play your animations. In simple cases one channel is enough, sometimes you need two or more Channels per

model to play gestures and motions in parallel.

## Code Samples

- [TestSpatialAnim.java](#)
- [TestBlenderAnim.java](#)
- [TestBlenderObjectAnim.java](#)
- [TestOgreAnim.java](#)
- [TestOgreComplexAnim.java](#)
- [TestCustomAnim.java](#)

## Controlling Animations

### The Animation Control

Create one `com.jme3.animation.AnimControl` object in your JME3 application for each animated model that you want to control. You have to register each animated model to one of these Animation Controls. The control object gives you access to the available animation sequences in the model.

```
AnimControl playerControl; // you need one Control per model
Node player = (Node) assetManager.loadModel("Models/Oto/Oto.mesh");
playerControl = player.getControl(AnimControl.class); // get control
playerControl.addListener(this); // add listener
```

### Animation Channels

An Animation Control has several Animation Channels (`com.jme3.animation.AnimChannel`). Each channel can play one animation sequence at a time.

There often are situations where you want to run several animation sequences at the same time, e.g. “shooting while walking” or “boxing while jumping”. In this case, you create several channels, assign an animation to each, and play them in parallel.

```
AnimChannel channel_walk = playerControl.createChannel();
AnimChannel channel_jump = playerControl.createChannel();
...
```

To reset a Control, call `control.clearChannels();`

</div>

## Animation Control Properties

The following information is available for an AnimControl.

<b>AnimControl Property</b>	<b>Usage</b>
<code>createChannel()</code>	Returns a new channel, controlling all bones by default.
<code>getNumChannels()</code>	The number of channels registered to this Control.
<code>getChannel(0)</code>	Gets individual channels by index number. At most <code>getNumChannels()</code> .
<code>clearChannels()</code>	Clear all channels in this control.
<code>addListener(animEventListener)</code> <code>removeListener(animEventListener)</code> <code>clearListeners()</code>	Adds or removes listeners to receive animation related events.

<b>AnimControl Property</b>	<b>Usage</b>
<code>setAnimations(aniHashMap)</code>	Sets the animations that this AnimControl is capable of playing. The animations must be compatible with the skeleton given in the constructor.
<code>addAnim(boneAnim)</code> <code>removeAnim(boneAnim)</code>	Adds or removes an animation from this Control.
<code>getAnimationNames()</code>	A String Collection of names of all animations that this Control can play for this model.
<code>getAnim("anim")</code>	Retrieve an animation from the list of animations.
<code>getAnimationLength("anim")</code>	Returns the length of the given named animation in seconds

<b>AnimControl Property</b>	<b>Usage</b>
<code>getSkeleton()</code>	The Skeleton object controlled by this Control.
<code>getTargets()</code>	The Skin objects controlled by this Control, as Mesh array.
<code>getAttachmentsNode("bone")</code>	Returns the attachment node of a bone. Attach models and effects to this node to make them follow this bone's motions.

# Animation Channel Properties

The following properties are set per AnimChannel.

AnimChannel Property	Usage
<code>setLoopMode(LoopMode.Loop);</code>	From now on, the animation on this channel will repeat from the beginning when it ends.
<code>setLoopMode(LoopMode.DontLoop);</code>	From now on, the animation on this channel will play once, and the freeze at the last keyframe.
<code>setLoopMode(LoopMode.Cycle);</code>	From now on, the animation on this channel will play forward, then backward, then again forward, and so on.
<code>setSpeed(1f);</code>	From now on, play this animation slower (<1f) or faster (>1f), or with default speed (1f).
<code>setTime(1.3f);</code>	Fast-forward or rewind to a certain moment in time of this animation.

The following information is available for a channel.

AnimChannel Property	Usage
<code>getAnimationName()</code>	The name of the animation playing on this channel. Returns <code>null</code> when no animation is playing.
<code>getLoopMode()</code>	The current loop mode on this channel. The returned com.jme3.animation enum can be LoopMode.Loop, LoopMode.DontLoop, or LoopMode.Cycle.
<code>getAnimMaxTime()</code>	The total length of the animation on this channel. Or <code>0f</code> if nothing is playing.
<code>getTime()</code>	How long the animation on this channel has been playing. It returns <code>0f</code> if the channel has not started playing yet, or a value up to <code>getAnimMaxTime()</code> .
<code>getControl()</code>	The AnimControl that belongs to this AnimChannel.

Use the following methods to add or remove individual bones to an AnimChannel. This is useful when you play two animations in parallel on two channels, and each controls a subset of the bones (e.g. one the arms, and the other the legs).

<b>AnimChannel Methods</b>	<b>Usage</b>
addAllBones()	Add all the bones of the model's skeleton to be influenced by this animation channel. (default)
addBone("bone1") addBone(bone1)	Add a single bone to be influenced by this animation channel.
addToRootBone("bone1") addToRootBone(bone1)	Add a series of bones to be influenced by this animation channel: Add all bones, starting from the given bone, to the root bone.
addFromRootBone("bone1") addFromRootBone(bone1)	Add a series of bones to be influenced by this animation channel: Add all bones, starting from the given root bone, going towards the children bones.

## Playing Animations

Animations are played by channel. **Note:** Whether the animation channel plays continuously or only once, depends on the Loop properties you have set.

<b>Channel Method</b>	<b>Usage</b>
channel_walk.setAnim("Walk",0.50f);	Start the animation named "Walk" on channel channel_walk. The float value specifies the time how long the animation should overlap with the previous one on this channel. If set to 0f, then no blending will occur and the new animation will be applied instantly.

**Tip:** Use the AnimEventLister below to react at the end or start of an animation cycle.

</div>

## Usage Example

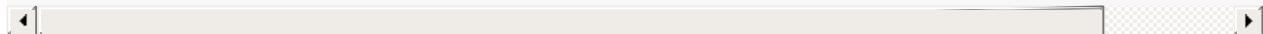
In this short example, we define the space key to trigger playing the "Walk" animation on channel2.

```

public void simpleInitApp() {
 ...
 inputManager.addMapping("Walk", new KeyTrigger(KeyInput.KEY_SPACE));
 inputManager.addListener(actionListener, "Walk");
 ...
}

private ActionListener actionListener = new ActionListener() {
 public void onAction(String name, boolean keyPressed, float tpf) {
 if (name.equals("Walk") && !keyPressed) {
 if (!channel2.getAnimationName().equals("Walk")) {
 channel2.setLoopMode(LoopMode.Loop);
 channel2.setAnim("Walk", 0.50f);
 }
 }
 }
};

```



## Animation Event Listener

A jME3 application that contains animations can implement the `com.jme3.animation.AnimEventListener` interface.

```

public class HelloAnimation extends SimpleApplication
 implements AnimEventListener { ... }

```

This optional Listener enables you to respond to animation start and end events, `onAnimChange()` and `onAnimCycleDone()`.

</div>

## Responding to Animation End

The `onAnimCycleDone()` event is invoked when an animation cycle has ended. For non-looping animations, this event is invoked when the animation is finished playing. For looping animations, this event is invoked each time the animation loop is restarted.

You have access to the following objects:

- The Control to which the listener is assigned.
- The animation channel being played.
- The name of the animation that has just finished playing.

```
public void onAnimCycleDone(AnimControl control, AnimChannel char
 // test for a condition you are interested in, e.g. ...
 if (animName.equals("Walk")) {
 // respond to the event here, e.g. ...
 channel.setAnim("Stand", 0.50f);
 }
}
```

## Responding to Animation Start

The `onAnimChange()` event is invoked every time before an animation is set by the user to be played on a given channel (`channel.setAnim()`).

You have access to the following objects

- The Control to which the listener is assigned.
- The animation channel being played.
- The name of the animation that will start playing.

</ul>

```
public void onAnimChange(AnimControl control, AnimChannel channel
 // test for a condition you are interested in, e.g. ...
 if (animName.equals("Walk")) {
 // respond to the event here, e.g. ...
 channel.setAnim("Reset", 0.50f);
 }
}
```

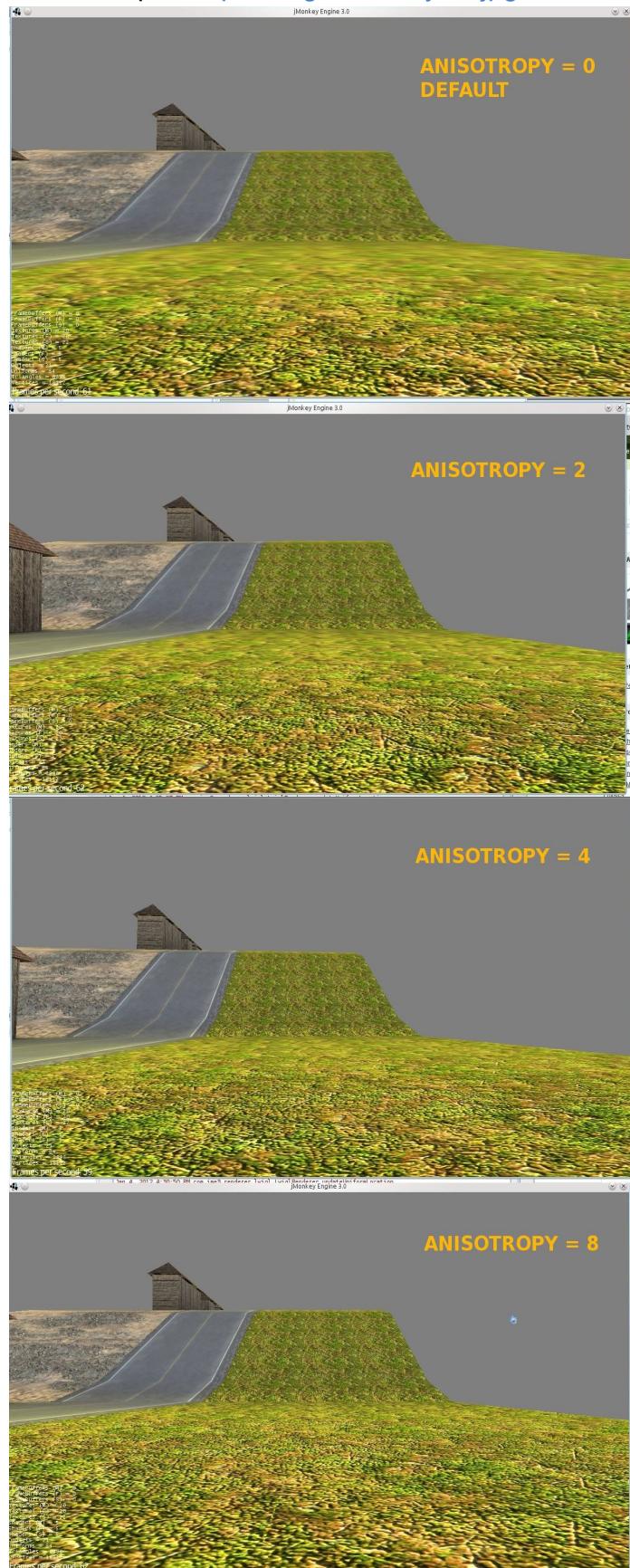
</div>

## **title: Anisotropic Filtering for Textures**

### **Anisotropic Filtering for Textures**

Anisotropic Filtering is very important for Desktop Games and their textures. Most games use AnisotropicFiltering = 4/8/16. It sharpens your textures under different Angle View. Anisotropy makes a performance draw back about 10-40 fps, but the result looks much better.

See Example: <http://i.imgur.com/0yiv9.jpg>



JME has DEFAULT AnisotropicFiltering = 0. So, if you make a game for Windows/Linux/Mac.. you need to set the Anisotropic Filtering more than 0.

Example how to set AnisotropicFiltering = 4 for all textures:

```
AssetEventListener asl = new AssetEventListener() {
 public void assetLoaded(AssetKey key) {
 // throw new UnsupportedOperationException("Not supported");

 public void assetRequested(AssetKey key) {
 if (key.getExtension().equals("png") || key.getExtension().equals("jpg"))
 System.out.println(key.getExtension());
 TextureKey tkey = (TextureKey) key;
 tkey.setAnisotropy(8);
 }
 }

 public void assetDependencyNotFound(AssetKey parentKey,
 // throw new UnsupportedOperationException("Not supported");
 };

 assetManager.addAssetEventListener(asl);
```

## title: Application States

# Application States

The `com.jme3.app.state.AppState` class is a customizable jME3 interface that allows you to control the global game logic, the overall game mechanics. (To control the behaviour of a Spatial, see [Custom Controls](#) instead. Controls and AppStates can be used together.)

## Overview

### Use Case Examples

There are situations during your game development where you think:

- Mouse and key inputs are handled differently in-game versus in the main menu. Can I group a set of input handler settings, and activate and deactivate them all in one step?
- I have the in-game scene, and a character editor, and a Captain's Quarters screen. Can I group a set of nodes and behaviours, and swap them in and out in one step?
- When I pause the game, I want the character's "idle" animation to continue, but all other loops and game events should stop. How do I define what happens when the game is paused/unpaused?
- I have a conditional block that takes up a lot of space in my `simpleUpdate()` loop. Can I wrap up this block of code, and switch it on and off in one step?
- Can I package everything that belongs in-game, and everything that belongs to the menu screen, and switch between these two "big states" in one step?

You can! This is what AppStates are there for. An AppState class is subset of (or an extension to) your application. Every AppState class has access to all fields in your main application (AssetManager, ViewPort, StateManager, InputManager, rootNode, GuiNode, etc) and hooks into the main update loop. An AppState can contain:

- a subset of class fields, functions, methods (game state data and accessors),
- a subset of GUI elements and their listeners,
- a subset of input handlers and mappings,
- a subset of nodes that you load and attach to the `rootNode`,
- a subset of conditional actions that you branch to in the `simpleUpdate()` loop,
- a subset of other AppStates and Controls

- ... or combinations thereof.

## Supported Features

Each AppState lets you define what happens to it in the following situations:

- **The AppState is initialized:** You load and initialize game data, InputHandlers, AppStates and Controls and attach nodes.  
The AppState executes its own `simpleInitApp()` method when it is attached, so to speak.
- **The AppState has been enabled (unpaused):** This toggles a boolean `isEnabled()` to true. Here you attach nodes and listeners that should become active while it's running.
- **While the AppState is running/paused:** You can poll `isEnabled()` to define paused and unpause game behaviour in the `update()` loop. In `update()`, you poll and modify the game state, modify the scene graph, and trigger events. Test if `!isEnabled()`, and write code that skips the running sections of this AppState's `update()` loop.  
Each AppState has its own update loop, which hooks into the main `simpleUpdate()` loop (callback).
- **The AppState has been disabled (paused):** This toggles a boolean `isEnabled()` to false. Here you switch all objects to their specific "paused" behaviour.
- **The AppState is cleaned up:** Here you decide what happens when the AppState is detached. Save this AppState's game state, unregister Controls and InputHandlers, detach related AppStates, detach nodes from the `rootNode`, etc.

Tip: AppStates are extremely handy to swap out, or pause/unpause whole sets of other AppStates. For example, an InGameState (loads in-game GUI, activates click-to-shoot input mappings, inits game content, starts game loop) versus MainScreenState (stops game loop, saves and detaches game content, switches to menu screen GUI, switches to click-to-select input mappings).

</div>

## Usage

To implement game logic:

1. Create one `AbstractAppState` instance for each set of game mechanics.
2. Implement game behaviour in the AppState's `update()` method.
  - You can pass custom data as arguments in the constructor.
  - The AppState has access to everything inside the app's scope via the Application `app` object.
3. Create and attach the AppState to the `AppStateManager`  
`(stateManager.attach(myAppState); )` and initialize it.
4. Enable and disable (unpause and pause) the AppStates that you need during the game.

5. Detach the AppState from the AppStateManager (`stateManager.detach(myAppState);`) and clean it up.

When you add several AppStates to one Application and activate them, their initialize() methods and update() loops are executed in the order in which the AppStates were added to the AppStateManager.

</div>

## Code Samples

JME3 comes with a BulletAppState that implements Physical behaviour (using the jBullet library). You, for example, could write an Artificial Intelligence AppState to control all your enemy units. Existing examples in the code base include:

- [BulletAppState](#) controls physical behaviour in PhysicsControl'ed Spatial's.
- [TestAppStates.java](#) an example of a custom AppState
  - [RootNodeState.java](#)

## AppState

The AppState interface lets you initialize sets of objects, and hook a set of continuously executing code into the main loop.

AppState Method	Usage
initialize(asm,app)	<p>When this AppState is added to the game, the RenderThread initializes the AppState and then calls this method. You can modify the scene graph from here (e.g. attach nodes). To get access to the main app, call:</p> <pre data-bbox="589 399 1256 473"><code>super.initialize(stateManager, app); this.app = (SimpleApplication) app;</code></pre>
cleanup()	<p>This method is executed after you remove the AppState from the game. Here you implement clean-up code for when this state is detached. You can modify the scene graph from here (e.g. detach nodes).</p>
update(float tpf)	<p>Here you implement the behaviour that you want to hook into the simpleUpdate() loop while this state is attached to the game. You can modify the scene graph from here.</p>
isInitialized()	<p>Your implementations of this interface should return the correct respective boolean value. (See AbstractAppState)</p>
setActive(true) setActive(false)	<p>Temporarily enables or disables an AppState. (See AbstractAppState)</p>
isActive()	<p>Test whether AppState is enabled or disabled. Your implementation should consider the boolean. (See AbstractAppState)</p>
stateAttached(asm) stateDetached(asm)	<p>The AppState knows when it is attached to, or detached from, the AppStateManager, and triggers these two methods. Don't modify the scene graph from here! (Typically not used.)</p>
render(RenderManager rm)	<p>Renders the state, plus your optional customizations. (Typically not used.)</p>
postRender()	<p>Called after all rendering commands are flushed, including your optional customizations. (Typically not used.)</p>

## AbstractAppState

The AbstractAppState class already implements some common methods (`isInitialized()`, `setEnabled()`, `cleanup()`) and makes creation of custom AppStates a bit easier. We recommend you extend AbstractAppState and override the remaining AppState methods:  
`initialize()`, `setEnabled()`, `cleanup()`.

Definition:

```
public class MyAppState extends AbstractAppState {
```

```
private SimpleApplication app;

private Node x = new Node("x"); // some custom class fields...
public Node getX(){ return x; } // some custom methods...

@Override
public void initialize(AppStateManager stateManager, Application app) {
 super.initialize(stateManager, app);
 this.app = (SimpleApplication)app; // cast to a more specific type

 // init stuff that is independent of whether state is PAUSED
 this.app.getRootNode().attachChild(getX()); // modify scene graph
 this.app.doSomething(); // call custom method
}

@Override
public void cleanup() {
 super.cleanup();
 // unregister all my listeners, detach all my nodes, etc...
 this.app.getRootNode().detachChild(getX()); // modify scene graph
 this.app.doSomethingElse(); // call custom method
}

@Override
public void setEnabled(boolean enabled) {
 // Pause and unpause
 super.setEnabled(enabled);
 if(enabled){
 // init stuff that is in use while this state is RUNNING
 this.app.getRootNode().attachChild(getX()); // modify scene graph
 this.app.doSomethingElse(); // call custom method
 } else {
 // take away everything not needed while this state is PAUSED
 ...
 }
}

// Note that update is only called while the state is both attached and active
@Override
```

```

 public void update(float tpf) {
 // do the following while game is RUNNING
 this.app.getRootNode().getChild("blah").scale(tpf); // modify
 x.setUserData(...); // call s
 }

 }

```

## Pausing and Unpausing

You define what an AppState does when Paused or Unpaused, in the `setEnabled()` and `update()` methods. Call `myState.setEnabled(false)` on all states that you want to pause. Call `myState.setEnabled(true)` on all states that you want to unpause.

</div>

## AppStateManager

The `com.jme3.app.state.AppStateManager` holds the list of AppStates for an application. `AppStateManager` ensures that active AppStates can modify the scene graph, and that the `update()` loops of active AppStates is executed. There is one `AppStateManager` per application. You typically attach several AppStates to one `AppStateManager`, but the same state can only be attached once.

<b>AppStateManager Method</b>	<b>Usage</b>
<code>hasState(myState)</code>	Is AppState object 'myState' attached?
<code>getState(MyAppState.class)</code>	Returns the first attached state that is an instance of a subclass of <code>MyAppState.class</code> .

The `AppStateManager`'s `render()`, `postRender()`, `cleanup()` methods are internal, ignore them, users never call them directly.

- If a detached AppState is attached then `initialize()` will be called on the following render pass.
- If an attached AppState is detached then `cleanup()` will be called on the following render pass.
- If you attach an already-attached AppState then the second attach is a no-op and will return false.
- If you both attach and detach an AppState within one frame then neither `initialize()` or

`cleanup()` will be called, although if either is called both will be.

- If you both detach and then re-attach an AppState within one frame then on the next update pass its `cleanup()` and `initialize()` methods will be called in that order.

# Best Practices

## Communication Among AppStates

You can only access other AppStates (read from and write to them) from certain places: From a Control's `update()` method, from an AppState's `update()` method, and from the SimpleApplication's `simpleUpdate()` loop. Don't mess with the AppState from other places, because from other methods you have no control over the order of modifications; the game can go out of sync because you can't know when (during which half-finished step of another state change) your modification will be performed.

You can use custom accessors to get data from AppStates, to set data in AppStates, or to trigger methods in AppStates.

```
this.app.getStateManager().getState(MyAppState.class).doSomeCustomS
◀ ▶
```

</div>

## Initialize Familiar Class Fields

To access class fields of the SimpleApplication the way you are used to, initialize them to local variables, as shown in the following AppState template:

```
private SimpleApplication app;
private Node rootNode;
private AssetManager assetManager;
private AppStateManager stateManager;
private InputManager inputManager;
private ViewPort viewPort;
private BulletAppState physics;

public class MyAppState extends AbstractAppState {
 @Override
 public void initialize(AppStateManager stateManager, Application application) {
 super.initialize(stateManager, application);
 this.app = (SimpleApplication) application; // can cast Application to SimpleApplication
 this.rootNode = this.app.getRootNode();
 this.assetManager = this.app.getAssetManager();
 this.stateManager = this.app.getStateManager();
 this.inputManager = this.app.getInputManager();
 this.viewPort = this.app.getViewPort();
 this.physics = this.stateManager.getState(BulletAppState.class);
 }
}
```

&lt;/div&gt;

## **title: Simple AppStates Demo**

# **Simple AppStates Demo**

THIS DEMO IS OUT OF DATE AND NEEDS CORRECTING

</div>

**THIS DEMO IS OUT OF DATE AND NEEDS CORRECTING FOR NOW PLEASE SEE**

[http://jmonkeyengine.org/wiki/doku.php/jme3:advanced:application\\_states](http://jmonkeyengine.org/wiki/doku.php/jme3:advanced:application_states)

Note: this tutorial needs to be fixed and is currently not correct. One should almost never override stateDetached and stateAttached... and should certainly never do anything scene related in them.

This demo is a simple example of how you use AppStates to toggle between a StartScreen and a SettingsScreen (press RETURN) while the game is paused, and start the game by switching to a GameRunning state (press BACKSPACE).

There are four files, Main.java, GameRunningState.java, StartScreenState.java, SettingsScreenState.java.

</div>

## **Main.java**

```
package chapter04.appstatedemo;

import com.jme3.app.SimpleApplication;
import com.jme3.input.KeyInput;
import com.jme3.input.controls.ActionListener;
import com.jme3.input.controls.KeyTrigger;
import com.jme3.input.controls.Trigger;
```

```
/*
 * This demo shows a simple "game" with three AppStates. Instead of
 * it just displays three cubes on different backgrounds.
 *
 * StartScreenState: This state is enabled
 * when the user starts the application, or the the game is paus
 * Press BACKSPACE to return to the game, press RETURN to go to
 * GameRunningState: This state shows the game content and is en
 * Press BACKSPACE to pause and return to the start screen.
 * SettingsScreenState: This Settings screen state can be reached
 * Press RETURN to toggle it on and off.
 *
 */
public class Main extends SimpleApplication {

 private Trigger pause_trigger = new KeyTrigger(KeyInput.KEY_BACK);
 private Trigger save_trigger = new KeyTrigger(KeyInput.KEY_RETURN);
 private boolean isRunning = false; // starts at startscreen
 private GameRunningState gameRunningState;
 private StartScreenState startScreenState;
 private SettingsScreenState settingsScreenState;

 /**
 * Start the jMonkeyEngine application */
 public static void main(String[] args) {
 Main app = new Main();
 app.start();
 }

 /**
 * initialize the scene here
 */
 @Override
 public void simpleInitApp() {
 setDisplayFps(false);
 setDisplayStatView(false);

 gameRunningState = new GameRunningState(this);
 startScreenState = new StartScreenState(this);
 settingsScreenState = new SettingsScreenState(this);
 }
}
```

```
stateManager.attach(startScreenState);

inputManager.addMapping("Game Pause Unpause", pause_trigger);
inputManager.addListener(actionListener, new String[]{"Game Pa
inputManager.addMapping("Toggle Settings", save_trigger);
inputManager.addListener(actionListener, new String[]{"Toggle S
}

private ActionListener actionListener = new ActionListener() {
 public void onAction(String name, boolean isPressed, float tpf)
 System.out.println("key" + name);
 if (name.equals("Game Pause Unpause") && !isPressed) {
 if (isRunning) {
 stateManager.detach(gameRunningState);
 stateManager.attach(startScreenState);
 System.out.println("switching to startscreen...");
 } else {
 stateManager.detach(startScreenState);
 stateManager.attach(gameRunningState);
 System.out.println("switching to game...");
 }
 isRunning = !isRunning;
 } else if (name.equals("Toggle Settings") && !isPressed && !i
 if (!isRunning && stateManager.hasState(startScreenState))
 stateManager.detach(startScreenState);
 stateManager.attach(settingsScreenState);
 System.out.println("switching to settings...");
 } else if (!isRunning && stateManager.hasState(settingsScre
 stateManager.detach(settingsScreenState);
 stateManager.attach(startScreenState);
 System.out.println("switching to startscreen...");
 }
 }
 }
};

@Override
public void simpleUpdate(float tpf) {}
```

```
}
```



```
</div>
```

## GameRunningState.java

```
package chapter04.appstatedemo;

import com.jme3.app.Application;
import com.jme3.app.SimpleApplication;
import com.jme3.app.state.AbstractAppState;
import com.jme3.app.state.AppStateManager;
import com.jme3.asset.AssetManager;
import com.jme3.font.BitmapFont;
import com.jme3.font.BitmapText;
import com.jme3.material.Material;
import com.jme3.math.ColorRGBA;
import com.jme3.math.Vector3f;
import com.jme3.renderer.ViewPort;
import com.jme3.scene.Geometry;
import com.jme3.scene.Node;
import com.jme3.scene.shape.Box;

/**
 * A template how to create an Application State. This example state
 * changes the background color depending on the camera position.
 */
public class GameRunningState extends AbstractAppState {

 private ViewPort viewPort;
 private Node rootNode;
 private Node guiNode;
 private AssetManager assetManager;
 private Node localRootNode = new Node("Game Screen RootNode");
 private Node localGuiNode = new Node("Game Screen GuiNode");
 private final ColorRGBA backgroundColor = ColorRGBA.Blue;
```

```

public GameRunningState(SimpleApplication app){
 this.rootNode = app.getRootNode();
 this.viewPort = app.getViewPort();
 this.guiNode = app.getGuiNode();
 this.assetManager = app.getAssetManager();
}

@Override
public void initialize(AppStateManager stateManager, Application app) {
 super.initialize(stateManager, app);

 /** Load this scene */
 viewPort.setBackgroundColor(backgroundColor);

 Box mesh = new Box(Vector3f.ZERO, 1, 1, 1);
 Geometry geom = new Geometry("Box", mesh);
 Material mat = new Material(assetManager,
 "Common/MatDefs/Misc/Unshaded.j3md");
 mat.setColor("Color", ColorRGBA.Green);
 geom.setMaterial(mat);
 geom.setLocalTranslation(1, 0, 0);
 localRootNode.attachChild(geom);

 /** Load the HUD*/
 BitmapFont guiFont = assetManager.loadFont(
 "Interface/Fonts/Default.fnt");
 BitmapText displaytext = new BitmapText(guiFont);
 displaytext.setSize(guiFont.getCharSet().getRenderedSize());
 displaytext.move(10, displaytext.getLineHeight() + 20, 0);
 displaytext.setText("Game running. Press BACKSPACE to pause and");
 localGuiNode.attachChild(displaytext);
}

@Override
public void update(float tpf) {
 /** the action happens here */
 Vector3f v = viewPort.getCamera().getLocation();
 viewPort.setBackgroundColor(new ColorRGBA(v.getX() / 10, v.getY(),
 rootNode.getChild("Box").rotate(tpf, tpf, tpf));
}

```

```
}

@Override
public void stateAttached(AppStateManager stateManager) {
 rootNode.attachChild(localRootNode);
 guiNode.attachChild(localGuiNode);
 viewPort.setBackgroundColor(backgroundColor);
}

@Override
public void stateDetached(AppStateManager stateManager) {
 rootNode.detachChild(localRootNode);
 guiNode.detachChild(localGuiNode);

}

}

</div>
```

## SettingsScreenState.java

```
package chapter04.appstatedemo;

import com.jme3.app.Application;
import com.jme3.app.SimpleApplication;
import com.jme3.app.state.AbstractAppState;
import com.jme3.app.state.AppStateManager;
import com.jme3.asset.AssetManager;
import com.jme3.font.BitmapFont;
import com.jme3.font.BitmapText;
import com.jme3.material.Material;
import com.jme3.math.ColorRGBA;
import com.jme3.math.Vector3f;
import com.jme3.renderer.ViewPort;
import com.jme3.scene.Geometry;
import com.jme3.scene.Node;
import com.jme3.scene.shape.Box;
```

```
/*
 * A template how to create an Application State. This example stat
 * changes the background color depending on the camera position.
 */
public class SettingsScreenState extends AbstractAppState {

 private ViewPort viewPort;
 private Node rootNode;
 private Node guiNode;
 private AssetManager assetManager;
 private Node localRootNode = new Node("Settings Screen RootNode");
 private Node localGuiNode = new Node("Settings Screen GuiNode");
 private final ColorRGBA backgroundColor = ColorRGBA.DarkGray;

 public SettingsScreenState(SimpleApplication app) {
 this.rootNode = app.getRootNode();
 this.viewPort = app.getViewPort();
 this.guiNode = app.getGuiNode();
 this.assetManager = app.getAssetManager();
 }

 @Override
 public void initialize(AppStateManager stateManager, Application app) {
 super.initialize(stateManager, app);

 /** Load this scene */
 viewPort.setBackgroundColor(backgroundColor);

 Box mesh = new Box(new Vector3f(-1, -1, 0), .5f, .5f, .5f);
 Geometry geom = new Geometry("Box", mesh);
 Material mat = new Material(assetManager,
 "Common/MatDefs/Misc/Unshaded.j3md");
 mat.setColor("Color", ColorRGBA.Red);
 geom.setMaterial(mat);
 geom.setLocalTranslation(1, 0, 0);
 localRootNode.attachChild(geom);

 /** Load the HUD */
 BitmapFont guiFont = assetManager.loadFont(
```

```

 "Interface/Fonts/Default.fnt");
BitmapText displaytext = new BitmapText(guiFont);
displaytext.setSize(guiFont.getCharSet().getRenderedSize());
displaytext.move(10, displaytext.getLineHeight() + 20, 0);
displaytext.setText("Settings screen. Press RETURN to save "
 + "and return to start screen.");
localGuiNode.attachChild(displaytext);
}

@Override
public void update(float tpf) {
 /** the action happens here */
}

@Override
public void stateAttached(AppStateManager stateManager) {
 rootNode.attachChild(localRootNode);
 guiNode.attachChild(localGuiNode);
 viewPort.setBackgroundColor(backgroundColor);
}

@Override
public void stateDetached(AppStateManager stateManager) {
 rootNode.detachChild(localRootNode);
 guiNode.detachChild(localGuiNode);
}

}

```

&lt;/div&gt;

## StartScreenState.java

```

package chapter04.appstatedemo;

import com.jme3.app.Application;
import com.jme3.app.SimpleApplication;
import com.jme3.app.state.AbstractAppState;

```

```
import com.jme3.app.state.AppStateManager;
import com.jme3.asset.AssetManager;
import com.jme3.font.BitmapFont;
import com.jme3.font.BitmapText;
import com.jme3.material.Material;
import com.jme3.math.ColorRGBA;
import com.jme3.math.Vector3f;
import com.jme3.renderer.ViewPort;
import com.jme3.scene.Geometry;
import com.jme3.scene.Node;
import com.jme3.scene.shape.Box;

/**
 * A template how to create an Application State. This example stat
 * changes the background color depending on the camera position.
 */
public class StartScreenState extends AbstractAppState {

 private ViewPort viewPort;
 private Node rootNode;
 private Node guiNode;
 private AssetManager assetManager;
 private Node localRootNode = new Node("Start Screen RootNode");
 private Node localGuiNode = new Node("Start Screen GuiNode");
 private final ColorRGBA backgroundColor = ColorRGBA.Gray;

 public StartScreenState(SimpleApplication app){
 this.rootNode = app.getRootNode();
 this.viewPort = app.getViewPort();
 this.guiNode = app.getGuiNode();
 this.assetManager = app.getAssetManager();
 }

 @Override
 public void initialize(AppStateManager stateManager, Application app) {
 super.initialize(stateManager, app);

 /** Init this scene */
 viewPort.setBackgroundColor(backgroundColor);
 }
}
```

```

 Box mesh = new Box(new Vector3f(-1,1,0), .5f,.5f,.5f);
 Geometry geom = new Geometry("Box", mesh);
 Material mat = new Material(assetManager,
 "Common/MatDefs/Misc/Unshaded.j3md");
 mat.setColor("Color", ColorRGBA.Yellow);
 geom.setMaterial(mat);
 geom.setLocalTranslation(1, 0, 0);
 localRootNode.attachChild(geom);

 /** Load a HUD */
 BitmapFont guiFont = assetManager.loadFont(
 "Interface/Fonts/Default.fnt");
 BitmapText displaytext = new BitmapText(guiFont);
 displaytext.setSize(guiFont.getCharSet().getRenderedSize());
 displaytext.move(10, displaytext.getLineHeight() + 20, 0);
 displaytext.setText("Start screen. Press BACKSPACE to resume th
 + "press RETURN to edit Settings.");
 localGuiNode.attachChild(displaytext);
}

@Override
public void update(float tpf) {
 /** the action happens here */
}

@Override
public void stateAttached(AppStateManager stateManager) {
 rootNode.attachChild(localRootNode);
 guiNode.attachChild(localGuiNode);
 viewPort.setBackgroundColor(backgroundColor);
}

@Override
public void stateDetached(AppStateManager stateManager) {
 rootNode.detachChild(localRootNode);
 guiNode.detachChild(localGuiNode);
}

}

```

</div>

## title: AssetManager

# AssetManager

By assets we mean multi-media files, such as 3D models, materials, textures, scenes, custom shaders, music and sound files, and custom fonts. JME3 has an integrated asset manager that helps you keep your project assets organized. Think of the asset manager as the filesystem of your game, independent of the actual deployment platform. By default, store your assets in the `MyGame/assets/` directory of your project.

Advantages of the AssetManager:

- The paths stay the same, no matter whether the game runs on Windows, Mac, Linux, etc!
- The AssetManager automatically caches and optimizes the handling of OpenGL objects.  
For example, the same textures are not uploaded to the graphics card multiple times when multiple models use them.
- The [default build script](#) automatically bundles the contents of the `assets` directory into the executable.

Advanced users can write a custom build and packaging script, and can register custom paths to the AssetManager, but this is up to you then.

## Context

```
jMonkeyProjects/MyGame/assets/ # You store assets in subfolders
jMonkeyProjects/MyGame/build/ # SDK generates built classes here
jMonkeyProjects/MyGame/build.xml # You customize Ant build script
jMonkeyProjects/MyGame/nbproject/ # SDK stores default build.xml ar
jMonkeyProjects/MyGame/dist/ # SDK generates executable distri
jMonkeyProjects/MyGame/src/ # You store Java sources here
jMonkeyProjects/MyGame/test/ # You store test classes here (op
(*) Managed by jMonkeyEngine SDK -- don't edit!
```

See also [Best Practices](#).

# Usage

The `assetManager` object is an `com.jme3.asset.AssetManager` instance that every `com.jme3.app.Application` can access. It maintains a root that also includes your project's classpath by default, so you can load any asset that's on the classpath, that is, the top level of your project directory.

You can use the inherited `assetManager` object directly, or use the accessor `app.getAssetManager()`.

Here is an example how you load assets using the AssetManager. This lines loads a default Material from the built-in `Common/` directory:

```
Material mat = (Material) assetManager.loadAsset(
 new AssetKey("Common/Materials/RedColor.j3m"));
```

This Material is “somewhere” in the jME3 JAR; the default Asset Manager is configured to handle a `Common/...` path correctly, so you don't have to specify the whole path when referring to built-in assets (such as default Materials).

Additionally, you can configure the Asset Manager and add any path to its root. This means, you can load assets from any project directory you specify. The next example shows how you load assets from your project's assets directory.

# Asset Directory

By default, jME3 searches for models in a directory named `assets`.

In Java projects created with the jMonkeyEngine SDK, an `assets` folder is created by default in your project directory. If you are using any other IDE, or the command line, you simply create an `assets` directory manually (see the Codeless Project tip below).

This is our recommended directory structure for storing assets:

```
jMonkeyProjects/MyGame/src/... # Packages, .java source code
jMonkeyProjects/MyGame/assets/... # The assets directory:
jMonkeyProjects/MyGame/assets/Interface/ # .font, .jpg, .png, .xml
jMonkeyProjects/MyGame/assets/MatDefs/ # .j3md
jMonkeyProjects/MyGame/assets/Materials/ # .j3m
jMonkeyProjects/MyGame/assets/Models/ # .j3o
jMonkeyProjects/MyGame/assets/Scenes/ # .j3o
jMonkeyProjects/MyGame/assets/Shaders/ # .j3f, .vert, .frag
jMonkeyProjects/MyGame/assets/Sounds/ # .ogg, .wav
jMonkeyProjects/MyGame/assets/Textures/ # .jpg, .png; also .mesh
```



These subdirectories are just the most common examples.

You can rename/delete/add (sub)directories inside the `assets` directory in any way you like. Note however that there is no automatic refactoring for asset paths in the SDK, so if you modify them late in the development process, you have to refactor all paths manually.

**Examples:** You can rename `assets/Sounds` to `assets/Audio`, you can delete `assets/MatDefs` if you don't use it, you can create `assets/AIscripts`, etc. You can rename/move the `assets/Textures` directory or its subdirectories, but then you have to re-export all models, and re-convert them all to `.j3o`, so plan ahead!

Store textures in `assets/Textures/` before you work with them in a mesh editor! Export and save 3D model files (`.mesh.xml+.material`, `.mtl+.obj`, `.blend`) into the `assets/Textures/` (!) before you convert the model to binary format (`.j3o`)! This ensures that texture paths correctly point to the `assets/Textures` directory.

After the conversion, you move the `.j3o` file into the `assets/Models/` or `assets/Scenes/` directories. This way, you can reuse textures, your binaries consistently link the correct textures, and the `assets/Models` and `assets/Scenes` directories don't become cluttered.

## Example Code: Loading Assets

Creating a material instance with the definition “Unshaded.j3md”:

```
Material mat_brick = new Material(
 assetManager, "Common/MatDefs/Misc/Unshaded.j3md");
```

Applying a texture to the material:

```
mat_brick.setTexture("ColorMap",
 assetManager.loadTexture("Textures/Terrain/BrickWall/BrickWall.
```

Loading a font:

```
guiFont = assetManager.loadFont("Interface/Fonts/Default.fnt");
```

Loading a model:

```
Spatial ninja = assetManager.loadModel("Models/Ninja/Ninja.mesh.xml")
```

Loading a scene from an Ogre3D dotScene file stored inside a zip:

```
assetManager.registerLocator("town.zip", ZipLocator.class);
Spatial scene = assetManager.loadModel("main.scene");
rootNode.attachChild(scene);
```

Alternatively to ZipLocator, there is also a HttpZipLocator that can stream models from a zip file online:

```
assetManager.registerLocator("http://jmonkeyengine.googlecode.com/f
HttpZipLocator.class);
Spatial scene = assetManager.loadModel("main.scene");
rootNode.attachChild(scene);
```

jME3 also offers a ClasspathLocator, ZipLocator, FileLocator, HttpZipLocator, and UrlLocator (see `com.jme3.asset.plugins` ).

The custom build script does not automatically include all ZIP files in the executable build.  
See “Cannot Locate Resource” solution below.

## Common AssetManager Tasks

Task?	Solution!
Load a model with materials	<p>Use the asset manager's <code>loadModel()</code> method and attach the Spatial to the rootNode:</p> <pre>Spatial elephant = assetManager.loadModel("Models/Elephant"); rootNode.attachChild(elephant);</pre> <p>Spatial elephant = assetManager.loadModel("Models/Elephant"); rootNode.attachChild(elephant);</p>
Load a model without materials	<p>If you have a model without materials, you have to add a default material to it:</p> <pre>Spatial teapot = assetManager.loadModel("Models/Teapot/1"); Material mat = new Material(assetManager, "Common/MatDefs/Misc/Unshaded.j3m"); teapot.setMaterial(mat); rootNode.attachChild(teapot);</pre>
Load a scene	<p>You load scenes just like you load models:</p> <pre>Spatial scene = assetManager.loadModel("Scenes/house/main"); rootNode.attachChild(scene);</pre>

## NullPointerException: Cannot locate resource?

### Problem:

My game runs fine when I run it right from the jMonkeyEngine SDK. But when I run the stand-alone executables (.jar, .jnlp, .exe, .app), a DesktopAssetManager error message occurs in the console, and it quits?

```
com.jme3.asset.DesktopAssetManager loadAsset
WARNING: Cannot locate resource: Scenes/town/main.scene
com.jme3.app.Application handleError
SEVERE: Uncaught exception thrown in Thread[LWJGL Renderer Thread,5]
java.lang.NullPointerException
```

### Reason:

If you use the default build script, **original models and scenes (.mesh.xml, .obj, .blend, .zip), are excluded** from the distribution automatically. A stand-alone executable includes converted **.j3o files** (models and scenes) only. The default build script makes sure to bundle existing .j3o files in the distribution, but you need to remember to convert the models (from mesh.xml→.j3o, or .obj→.j3o, etc) yourself.

## Solution

Before building the executable, you must use the jMonkeyEngine SDK's context menu action to [convert 3D models to .j3o binary format](#).

1. Save your original models (.mesh.xml, .scene, .blend, or .obj files, plus textures) into `assets/Textures/ . (!)`
2. Open the jME3 project in the jMonkeyEngine SDK.
3. Browse to the `assets` directory in the Projects window.
4. Right-click an original model in `assets/Textures/`, and choose “Convert to JME3 binary”.
5. The converted file appears in the same directory as the original file. It has the same name and a `.j3o` suffix.
6. Move the .j3o file into the `assets/Models/` or `assets/Scenes/` directory.
7. Use the assetManager's `load()` method to load the `.j3o` file.

This ensures that the model's Texture paths keep working between your 3D mesh editor and JME3.

If you must load custom assets from a non-.j3o ZIP file, you must manually amend the [default build script](#) to copy ZIP files into your distribution. ZIPs are skipped by default.

# Asset Handling For Other IDEs: Codeless Projects

## Problem:

I use another IDE than jMonkeyEngine SDK for coding (Eclipse, IntelliJ, text editor). Where is my `asset` folder and .j3o converter?

## Solution:

You can code in any IDE, but you must create a so-called codeless project in the jMonkeyEngine SDK to maintain assets. **A code-less jMonkeyEngine project does not meddle with your sources or custom build scripts.** You merely use it to convert models to .j3o binaries.

1. Create your (Eclipse or whatever) project as you like.
2. Create a directory in your project folder and name it, for example, `assets`.  
Store your assets there as described above.
3. Download and install the jMonkeyEngine SDK.
4. In the SDK, go to File → Import Projects → External Project Assets.
5. Select your (Eclipse or whatever) project and your assets folder in the Import Wizard.
6. You can now open this (Eclipse or whatever) project in the jMonkeyEngine SDK.  
Convert assets as described above.

If you don't use the SDK for some reason, you can still convert models to j3o format: Load any model in Ogre3D or Wavefront format with the `AssetManager.loadModel()` as a spatial. Then save the spatial as j3o file using [BinaryExporter](#).

Use file version control and let team members check out the project. Your developers open the project in Eclipse (etc) as they are used to. Additionally to their graphic tools, ask your graphic designers to install the jMonkeyEngine SDK, and to check out the codeless project that you just prepared. This makes it easy for non-coding team member to browse and preview game assets, to arrange scenes, and to convert files. At the same time, non-coders don't accidentally mess with code, and developers don't accidentally mess with assets. :)

</div>

## **title: Atom framework Introduction**

### **Atom framework Introduction**

Hi Monkeys,

**Atom framework for game developing in Java. Powered by JME3.**

**Atom** framework which on top of JME3 and have some features like AI, Scripting, Database, VirtualReallity, Trigger, Multiplayer...(more below) to make developing game process in JME3 much more easier!

Go to the Atomix Game making tutorials. [atomixtuts](#)

</div>

### **Open Source**

Atom Core : The AtomCore source code is hosted in Googlecode

Googlecode: <https://code.google.com/p/atom-game-framework/>

Github: <https://github.com/atomixnmc/atom-game-framework>

Wiki: <https://code.google.com/p/atom-game-framework/wiki/>

### **Documentations:**

[Detailed architecture designs](#)

[Detailed documentation](#)

[Detailed comparison](#)

### **Idea & Buzz**

**Better, more freedom, more fun!**

From ~~UDK and Unity~~ 's battle field, I've dreamt about making even better game engine with the help of my 2 fav technologies - **Java** and **opensource**.

JME3 and Netbean combination are the most brighten idea I 've seen in years. I think it deserve a better reputation worldwide.

## Initial Ideas:

- Ease the learning curve
- Fun play/create game like kids in a sandbox
- Framework for games and apps(3D)
- Nextgen techs

## Why call it Atom?

<https://en.wikipedia.org/wiki/Atom>

Yes, this is for game developing. But in its heart is future's technologies.

Atom Etymology: Devive the thing as small as you can, than compose



The same in programming, after all, modulize is to be integrated again! That really is the core idea of Atom framework, use the most simple primitives to compose the bigger, bigger matter! Every where, small tiny, fastest, embed inside others, stick together well...Take a look in Atom Ex [atomex](#), you will see the picture more clearly.

The most conceptual inspiration for Atom framework is project

<http://ptolemy.eecs.berkeley.edu/index.htm> . Unfortunately Ptolemy is in “for research only” area and its direction toward the much large scale than game developing. That's why Atom framework was born with the learnt architecture from Ptolemy.

The Ptolemy project studies modeling, simulation, and design of concurrent, real-time, embedded systems. The focus is on assembly of concurrent components. The key underlying principle in the project is the use of well-defined models of computation that govern the interaction between components. A major problem area being addressed is the use of heterogeneous mixtures of models of computation

Atom is designed to MAKE concurrent, real-time, embedded systems and GAMES. So it focus more in code generation, profile, monitoring; focus more in graphics, physics, player experience...etc. Underlying, it borrow quite a bunch of concept that built in Ptolemy.

## Features

### Atom framework Highlights

- Flexible: Game | simulations centric but not forced!
- Modular: Dependency injection along with Component injection

- Parallel: Embrace parallel computing
- Next gen: Come with Bleeding edge technologies and powers of Java languages
- Cloud ready: Scale to web and distributed computing
- With ease: GUI Tools everywhere, almost zero config need

## Full Features list

- Cross game genre framework for JME3
  - General stage - world ; game play and cycle
  - General Entity framework
  - General Event framework
  - Intuitive config framework - toward zero-config approach
  - Actor and interactive agents, workers
  - Execution and monitor
- Network | Web ready
  - Web scale support
  - Network & Messaging framework
  - Social foundation framework
- Additional to JME3 features (as libraries or toolkits)
  - UltimateEffects
    - UltimateParticles
    - Cross dimension model
    - Filters
  - Cinematic
  - General extensible GUI with CSS
  - Ultimate asset pipeline
  - Ultimate animation framework
  - Extra rendering methods

## Architecture and components

Here are its architecture and components.



## Atom Core Libraries

- **AtomCore** : addition to JME3 core packages.
    - Cross game-genre elements: stage, cycle, entity, logic, trigger, event, config;
    - Managers: Advanced assets manager, dependency injection, factory, scripting, basic DB..;
    - Common case: Common state, common scenario, common UIs...
    - More Buzz? [AtomCore documentation](#)
  - **Atom Scripting** Base technology for use Groovy (potentional Scala, Jython..) as JME game scripting language...
    - Provide Test bed environment, thread-safe and intelligent Groovy swing component to extend the SDK in seconds
    - More Buzz? [Atom Scripting](#)
  - **Atom AI** : a “framework” to bring AI to jME3 game (also means real-time application)!
- But it's awesome in its own way.
- Focus in AI coding, creating, testing, simulating, profiling in 3d environments.
  - Come with tools as SDK plugins!
  - Check [Atom AI wiki](#) for more buzz

## Ingame editor facilities and kits

- **Atom Editor**: sophisticated in-game editor application and API for 3D games modelled toward netbean platform architecture. [In contrast with SimpleGameEditor project].

- **Atom 2D Editor**: for 2D games.
- **Code Gen**: a “framework” that intend to become the base technologies for all generation related techs in the Atom framework. [codegen](#)
  - Focus in provide general and abstract way to modeling|design game|real-time app concept and object, source codes.
  - Its first attempt to become a GLSL, Groovy generator, then become a Logic, source code generator...
  - Come with tools as SDK plugins!
- **City Gen**: a “framework” at first try to be a city generator, then grow up to be fullfill every geometric generating operations in 3D.
  - Focus in “Level” generator with 3d models, blueprint and geometric shapes, such as dugeon, city, rivers, mountain, trees...
  - Can corporate with Code gen and other geometric libs to become a generative 3D editor...
  - Come with tools as SDK plugins!

## Atom SDK

- **Atom SDK** : Expansion for current functions and features of the jME SDK on top of Netbean platform for desktop Swing based editing, more intuitive more user friendly and suchs.
  - Full List? [atomsdk](#)
- **TeeheeComposer** : Act as the base editor for video, cinematic, audio, effects, facial composer... anything require time-base keyframed or unlinear editing like sequences.
  - An almighty composer, think about 3DSMax or Adobe After Effect in 3D
  - Come with a lot of tools for the SDK : [teehee](#)
    - Cinematic composer
    - Dialogue composer
    - Effect composer
    - Particle composer
    - Animation composer
- **RPGCreator** : Despite of its name, its not just for Role playing game!
  - Provide functions to create| test| config basic game with these key elements : characters| stories| skills| items| modes| regions... almost every game genre has them embedded partly ( cross game genre)
  - Come with tools as SDK plugins! [rpgcreator](#)
- **Nextgen Tools**
  - Facial tools : Think FaceFX for JME :p [facial](#)
  - Character customization management tools : Smart way to organize and corporate your assets, config, database and code for CC [cc](#)

- Vitural reality tools : Toolset for corporate vitural reality artifact in your app [vr](#)
- MMORPG tools : Toolset for creating of a MMORPG game's component and all its management structure. Epic! [mmorgtools](#)
- Human Simulation tools: Think advanced locomotion and AI (like Mechanim of Unity) multiply 10. In fact, it's quite similar with tool from Autodesk that simulations social behaviours of human characters. Epic! [humansim](#)

## AtomEx Libraries and platform

- **Atom Ex** : addition to Atom framework which make its much more modulizable, extensible and enterprise ready. Distributed computing, web based, database... much more.
  - More Buzz? [AtomEx documentation](#)

## Vision

### Java,... again??!

Yeah, it was long time ago, you quit learning java because java gaming is a dead end.

But Android come to play, and the the market are open so freaking big that even companies live with their C++ code base want to take advantage of the new wave...

Recently Java has so much improvements and then JME3 enchant the talents all around the world to develop the master peices of software!

### But did we chasing after them?

No, we are not. We are going ahead of them with all the techniques from the almighty open-source.

Java communities are much more open and helpful than any of those Microsoft, Apple, UDK, Unity,... evils... Let's make a fairplay at last!

### Can we win?

The time will tell... but at least, we once gain give the power to the hands of the people, not just some rich and intelligent people, that's the most critical point!

### Project status

If you interest in contribute to Atom framework open-sourced project, here is the status of the project in 2014 and some mile stones it want to reach in the future.

[Atom framework open-sourced project Status - 2014](#)

## Other open-source dependencies

Actually it use directly/indirectly various projects of JME3 great contributors and open source projects:

- AI from @Sploreg, @shirkit and mine
- VirtualReallity integrated with OpenCV, JavaCV : @noncom + mine
- ShaderBlow from @mifth
- SpriteEngine @dansion
- Forestor from @androlo
- Multiplayer on top of MirrorMonkey, Kryonet, Arianne, ThreeRings, ...
- MonkeyZone code which I believe written by @normen @nehon and core guys :p
- Database using Cayenne, Depot
- ... other contributors

( I will add them later :p please forgive if I can't remember your name immediately )

Hundred of opensource projects...Nail it

*I want to thank all of you for your great great great contributions, help me and my friends here to start learning game programming and doing our own game. Salute! My job is to glue those great gems together, (pretty time consuming job) :*

As the splitting above, then I will make two different topic to keep them separate, the Atom framework and the Series of game making.

[Atomix Series of game making](#)

[GOTO Detailed Atom framework Documentation](#)

</div>

## **title: Audio Environment Presets**

# **Audio Environment Presets**

Use these presets together with [Audio Nodes](#) to create different “moods” for sounds. Environment effects make your audio sound as if the listener were in various places that have different types of echoes.

Usage:

```
Environment Generic = new Environment(
 new float[]{ 0, 7.5f, 1f, -1000, -100, 0, 1.49f, 0.83f, 1f, -26
 0.007f, 0f, 0f, 0f, 200, 0.011f, 0f, 0f, 0f, 0.25f
 0f, 0.250f, 0f, -5f, 5000f, 250f, 0f, 0x3f});
audioRenderer.setEnvironment(myEnvironment);
```

</div>

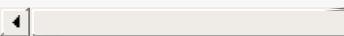
## **Castle**

```
CastleSmallRoom = new Environment (new float[]{ 26, 8.3f, 0.89f
CastleShortPassage = new Environment (new float[]{ 26, 8.3f, 0.89f
CastleMediumroom = new Environment (new float[]{ 26, 8.3f, 0.93f
CastleLongpassage = new Environment (new float[]{ 26, 8.3f, 0.89f
CastleLargeroom = new Environment (new float[]{ 26, 8.3f, 0.82f
CastleHall = new Environment (new float[]{ 26, 8.3f, 0.81f
CastleCupboard = new Environment (new float[]{ 26, 8.3f, 0.89f
CastleCourtyard = new Environment (new float[]{ 26, 8.3f, 0.42f
CastleAlcove = new Environment (new float[]{ 26, 8.3f, 0.89f
```

</div>

## **Warehouse, Factory**

```
FactoryAlcove = new Environment (new float[]{ 26, 1.8f, 0.59
FactoryShortpassage = new Environment (new float[]{ 26, 1.8f, 0.64
FactoryMediumroom = new Environment (new float[]{ 26, 1.9f, 0.82
FactoryLongpassage = new Environment (new float[]{ 26, 1.8f, 0.64
FactoryLargeroom = new Environment (new float[]{ 26, 1.9f, 0.75
FactoryHall = new Environment (new float[]{ 26, 1.9f, 0.75
FactoryCupboard = new Environment (new float[]{ 26, 1.7f, 0.63
FactoryCourtyard = new Environment (new float[]{ 26, 1.7f, 0.57
FactorySmallroom = new Environment (new float[]{ 26, 1.8f, 0.82
```



&lt;/div&gt;

## Ice Palace

```
IcepalaceAlcove = new Environment (new float[]{ 26, 2.7f, 0.
IcepalaceShortpassage = new Environment (new float[]{ 26, 2.7f, 0.
IcepalaceMediumroom = new Environment (new float[]{ 26, 2.7f, 0.
IcepalaceLongpassage = new Environment (new float[]{ 26, 2.7f, 0.
IcepalaceLargerom = new Environment (new float[]{ 26, 2.9f, 0.
IcepalaceHall = new Environment (new float[]{ 26, 2.9f, 0.
IcepalaceCupboard = new Environment (new float[]{ 26, 2.7f, 0.
IcepalaceCourtyard = new Environment (new float[]{ 26, 2.9f, 0.
IcepalaceSmallroom = new Environment (new float[]{ 26, 2.7f, 0.
```



&lt;/div&gt;

## Space Station

```

SpacestationAlcove = new Environment (new float[]{ 26, 1.5f,
SpacestationMediumroom = new Environment (new float[]{ 26, 1.5f,
SpacestationShortpassage = new Environment (new float[]{ 26, 1.5f,
SpacestationLongpassage = new Environment (new float[]{ 26, 1.9f,
SpacestationLargeroom = new Environment (new float[]{ 26, 1.8f,
SpacestationHall = new Environment (new float[]{ 26, 1.9f,
SpacestationCupboard = new Environment (new float[]{ 26, 1.4f,
SpacestationSmallroom = new Environment (new float[]{ 26, 1.5f,

```



&lt;/div&gt;

## Wooden Hut or Ship

```

WoodenAlcove = new Environment (new float[]{ 26, 7.5f, 1
WoodenShortpassage = new Environment (new float[]{ 26, 7.5f, 1
WoodenMediumroom = new Environment (new float[]{ 26, 7.5f, 1
WoodenLongpassage = new Environment (new float[]{ 26, 7.5f, 1
WoodenLargeroom = new Environment (new float[]{ 26, 7.5f, 1
WoodenHall = new Environment (new float[]{ 26, 7.5f, 1
WoodenCupboard = new Environment (new float[]{ 26, 7.5f, 1
WoodenSmallroom = new Environment (new float[]{ 26, 7.5f, 1
WoodenCourtyard = new Environment (new float[]{ 26, 7.5f, 6

```



&lt;/div&gt;

## Sport

```

SportEmptystadium = new Environment (new float[]{ 26, 7.2f, 1
SportSquashcourt = new Environment (new float[]{ 26, 7.5f, 6
SportSmallswimmingpool = new Environment (new float[]{ 26, 36.2f,
SportLargeswimmingpool = new Environment (new float[]{ 26, 36.2f,
SportGymnasium = new Environment (new float[]{ 26, 7.5f, 6
SportFullstadium = new Environment (new float[]{ 26, 7.2f, 1

```



</div>

## Pipes

```
Sewerpipe = new Environment (new float[]{ 21, 1.7f, 0.800f, -1000, 0.0f, 0.0f, 0.0f });
PipeSmall = new Environment (new float[]{ 26, 50.3f, 1f, -1000, 0.0f, 0.0f, 0.0f });
PipeLongthin = new Environment (new float[]{ 26, 1.6f, 0.910f, -1000, 0.0f, 0.0f, 0.0f });
PipeLarge = new Environment (new float[]{ 26, 50.3f, 1f, -1000, 0.0f, 0.0f, 0.0f });
PipeResonant = new Environment (new float[]{ 26, 1.3f, 0.910f, -1000, 0.0f, 0.0f, 0.0f });
```



</div>

## Moods

```
Heaven = new Environment (new float[]{ 26, 19.6f, 0.940f, -1000, 0.0f, 0.0f, 0.0f });
Hell = new Environment (new float[]{ 26, 100f, 0.570f, -1000, 0.0f, 0.0f, 0.0f });
Memory = new Environment (new float[]{ 26, 8f, 0.850f, -1000, 0.0f, 0.0f, 0.0f });
Drugged = new Environment (new float[]{ 23, 1.9f, 0.500f, -1000, 0.0f, 0.0f, 0.0f });
Dizzy = new Environment (new float[]{ 24, 1.8f, 0.600f, -1000, 0.0f, 0.0f, 0.0f });
Psychotic = new Environment (new float[]{ 25, 1f, 0.500f, -1000, 0.0f, 0.0f, 0.0f });
```



</div>

## Car Racing

```
DrivingCommentator = new Environment (new float[]{ 26, 3f, 0f, 0.0f, 0.0f, 0.0f, 0.0f });
DrivingPitgarage = new Environment (new float[]{ 26, 1.9f, 0.6f, 0.0f, 0.0f, 0.0f, 0.0f });
DrivingIncarRacer = new Environment (new float[]{ 26, 1.1f, 0.6f, 0.0f, 0.0f, 0.0f, 0.0f });
DrivingIncarSports = new Environment (new float[]{ 26, 1.1f, 0.6f, 0.0f, 0.0f, 0.0f, 0.0f });
DrivingIncarLuxury = new Environment (new float[]{ 26, 1.6f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f });
DrivingFullgrandstand = new Environment (new float[]{ 26, 8.3f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f });
DrivingEmptygrandstand = new Environment (new float[]{ 26, 8.3f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f });
DrivingTunnel = new Environment (new float[]{ 26, 3.1f, 0.6f, 0.0f, 0.0f, 0.0f, 0.0f });
```



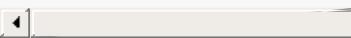
&lt;/div&gt;

## City

```

CityIndoors = new Environment (new float[]{ 16, 7.5f, 0.500f, -1000,
CityStreets = new Environment (new float[]{ 26, 3f, 0.780f, -1000,
CitySubway = new Environment (new float[]{ 26, 3f, 0.740f, -1000,
CityMuseum = new Environment (new float[]{ 26, 80.3f, 0.820f, -1000,
CityLibrary = new Environment (new float[]{ 26, 80.3f, 0.820f, -1000,
CityUnderpass = new Environment (new float[]{ 26, 3f, 0.820f, -1000,
CityAbandoned = new Environment (new float[]{ 26, 3f, 0.690f, -1000

```



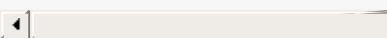
&lt;/div&gt;

## Small Indoor Rooms

```

Room = new Environment (new float[]{ 2, 1.9f, 1f, -1000, -1000,
Bathroom = new Environment (new float[]{ 3, 1.4f, 1f, -1000, -1000,
Livingroom = new Environment (new float[]{ 4, 2.5f, 1f, -1000, -1000,
Paddedcell = new Environment (new float[]{ 1, 1.4f, 1f, -1000, -1000,
Stoneroom = new Environment (new float[]{ 5, 11.6f, 1f, -1000, -1000

```



&lt;/div&gt;

## Medium-Sized Indoor Rooms

```

Workshop = new Environment (new float[]{ 26, 1.9f, 1f, -1000, -1000,
Schoolroom = new Environment (new float[]{ 26, 1.86f, 0.690f, -1000,
Practiseroom = new Environment (new float[]{ 26, 1.86f, 0.870f, -1000,
Outhouse = new Environment (new float[]{ 26, 80.3f, 0.820f, -1000,
Caravan = new Environment (new float[]{ 26, 8.3f, 1f, -1000, -1000,
Dustyroom = new Environment (new float[]{ 26, 1.8f, 0.560f, -1000,
Chapel = new Environment (new float[]{ 26, 19.6f, 0.840f, -1000

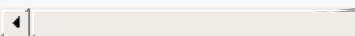
```



```
</div>
```

## Large Indoor Rooms

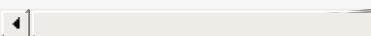
```
Auditorium = new Environment (new float[]{ 6, 21.6f, 1f, -1000f, 0.0f, 0.0f, 0.0f });
Concerthall = new Environment (new float[]{ 7, 19.6f, 1f, -1000f, 0.0f, 0.0f, 0.0f });
Cave = new Environment (new float[]{ 8, 14.6f, 1f, -1000f, 0.0f, 0.0f, 0.0f });
Arena = new Environment (new float[]{ 9, 36.2f, 1f, -1000f, 0.0f, 0.0f, 0.0f });
Hangar = new Environment (new float[]{ 10, 50.3f, 1f, -1000f, 0.0f, 0.0f, 0.0f });
DomeTomb = new Environment (new float[]{ 26, 51.8f, 0.790f, 0.0f, 0.0f, 0.0f, 0.0f });
DomeSaintPauls = new Environment (new float[]{ 26, 50.3f, 0.870f, 0.0f, 0.0f, 0.0f, 0.0f });
```



```
</div>
```

## Hallways, Alleys

```
Carpettedhallway = new Environment (new float[]{ 11, 1.9f, 1f, -1000f, 0.0f, 0.0f, 0.0f });
Hallway = new Environment (new float[]{ 12, 1.8f, 1f, -1000f, 0.0f, 0.0f, 0.0f });
Stonecorridor = new Environment (new float[]{ 13, 13.5f, 1f, -1000f, 0.0f, 0.0f, 0.0f });
Alley = new Environment (new float[]{ 14, 7.5f, 0.300f, 0.0f, 0.0f, 0.0f, 0.0f });
```



```
</div>
```

## Outdoors

```
Backyard = new Environment (new float[]{ 26, 80.3f, 0.450f, -
Plain = new Environment (new float[]{ 19, 42.5f, 0.210f, -
Rollingplains = new Environment (new float[]{ 26, 80.3f, 0f, -1000f,
Deepriver = new Environment (new float[]{ 26, 80.3f, 0.740f, -
Creek = new Environment (new float[]{ 26, 80.3f, 0.350f, -
Valley = new Environment (new float[]{ 26, 80.3f, 0.280f, -
Forest = new Environment (new float[]{ 15, 38f, 0.300f, -1000f,
Mountains = new Environment (new float[]{ 17, 100f, 0.270f, -1000f,
Quarry = new Environment (new float[]{ 18, 17.5f, 1f, -1000f,
Parkinglot = new Environment (new float[]{ 20, 8.3f, 1f, -1000f,
```

&lt;/div&gt;

## Water

```
Underwater = new Environment (new float[]{ 22, 1.8f, 1f, -1000f,
Smallwaterroom = new Environment (new float[]{ 26, 36.2f, 0.700f,
```

&lt;/div&gt;

## title: Audio in jME3

# Audio in jME3

Place audio files in the `assets/Sound/` directory of your project. jME3 supports Ogg Vorbis audio compression (.ogg) and uncompressed PCM Wave (.wav) formats. You can use for example [Audacity](#) to convert from other formats.

## Audio Terminology

- **Streaming:** There are two ways to load audio data: Short audio files are to be stored entirely in memory (prebuffered), while long audio files, such as music, are streamed from the hard drive as it is played.
- **Looping:** You can play a sound either once and then stop, or repeatedly (continuously) in a loop.  
You cannot loop streamed sounds.
- **Instance:** If you play the same audio twice, the playing is queued up and jME plays one after the other. If you play instances of sounds, several instances of the same sound can play at the same time.

## Creating Audio Nodes: Streamed or Buffered

The main jME audio class to look at is `com.jme3.audio.AudioNode`. When creating a new audio node you need to declare whether how you want to load this sound:

- **Buffered:** By default, a new audio node is buffered. This means jME3 loads the whole file into memory before playing. Use this for short sounds. You create a buffered sound by setting the boolean to false, or using no boolean at all:

```
AudioNode boom = new AudioNode(assetManager, "Sound/boom.wav");
AudioNode boom = new AudioNode(assetManager, "Sound/boom.wav",
```



- **Streamed:** If it is a long file such as music or a dialog, you stream the audio. Streaming means, you load and play in parallel until the sound is done. You cannot loop streams. You create a streamed sound by setting the boolean to true:

```
AudioNode music = new AudioNode(assetManager, "Sound/music.wav")
```



## Getting AudioNode Properties

AudioNode Method	Usage
getStatus()	Returns either AudioSource.Status.Playing, AudioSource.Status.Stopped, or AudioSource.Status.Paused.
getVolume()	Returns the volume.
getPitch()	Returns the pitch.

Note: There are other obvious getters to poll the status of all corresponding setters listed here.

## Setting AudioNode Properties

AudioNode Method	Usage
setTimeOffset(0.5f)	Play the sound starting at a 0.5 second offset from the beginning. Default is 0.
setPitch(1)	Makes the sound play in a higher or lower pitch. Default is 1. 2 is twice as high, .5f is half as high.
setVolume(1)	Sets the volume gain. 1 is the default volume, 2 is twice as loud, etc. 0 is silent/mute.
setRefDistance(50f)	The reference distance controls how far a sound can still be heard at 50% of its original volume ( <i>this is assuming an exponential fall-off!</i> ). A sound with a high RefDist can be heard loud over wide distances; a sound with a low refDist can only be heard when the listener is close by. Default is 10 world units.
setMaxDistance(100f)	The 'maximum attenuation distance' specifies how far from the source the sound stops growing more quiet (sounds in nature don't do that). Set this to a smaller value to keep the sound loud even at a distance; set this to higher value to let the sound fade out quickly. Default is 20 world units.
setLooping(false)	Configures the sound so that, if it is played, it plays once and stops. No looping is the default.

## Looping & Ambient Sounds

AudioNode Method	Usage
setPositional(false) setDirectional(false)	All 3D effects switched off. This sound is global and plays in headspace (it appears to come from everywhere). Good for environmental ambient sounds and background music.
setLooping(true)	Configures the sound to be a loop: After the sound plays, it repeats from the beginning, until you call stop() or pause(). Good for music and ambient background noises. <b>Looping does not work on streamed sounds.</b>

## Positional 3D Sounds

AudioNode Method	Usage
setPositional(true) setLocalTranslation(...)	Activates 3D audio: The sound appears to come from a certain position, where it is loudest. Position the AudioNode in the 3D scene, or move it with mobile players or NPCs.
setReverbEnabled(true)	Reverb is a 3D echo effect that only makes sense with positional AudioNodes. Use Audio Environments to make scenes sound as if they were “outdoors”, or “indoors” in a large or small room, etc. The reverb effect is defined by the com.jme3.audio.Environment that the audioRenderer is in. See “Setting Audio Environment Properties” below.

Positional 3D sounds require an `AudioListener` object in the scene (representing the player's ears).

</div>

## Directional 3D Sounds

AudioNode Method	Usage
setDirectional(true) setDirection(...)	Activates 3D audio: This sound can only be heard from a certain direction. Specify the direction and angle in the 3D scene if you have <code>setDirectional()</code> true. Use this to restrict noises that should not be heard, for example, through a wall.
setInnerAngle() setOuterAngle()	Set the angle in degrees for the directional audio. The angle is relative to the direction. Note: By default, both angles are 360° and the sound can be heard from all directions!

Directional 3D sounds require an `AudioListener` object in the scene (representing the player's ears).

</div>

# Play, Pause, Stop

You play, pause, and stop a node called `myAudioNode` by using the respective of the following three methods:

```
myAudioNode.play();
```

```
myAudioNode.pause();
```

```
myAudioNode.stop();
```

**Note:** Whether an Audio Node plays continuously or only once, depends on the Loop properties you have set above!

You can also start playing instances of an `AudioNode`. Use the `playInstance()` method if you need to play the same `AudioNode` multiple times, possibly simultaneously. Note that changes to the parameters of the original `AudioNode` do not affect the instances that are already playing!

```
myAudioNode.playInstance();
```

# The Audio Listener

The default `AudioListener` object `listener` in `SimpleApplication` is the user's ear in the scene. If you use 3D audio (positional or directional sounds), you must move the `AudioListener` with the player: For example, for a first-person player, you move the listener with the camera. For a third-person player, you move the listener with the player avatar Geometry.

```
@Override
public void simpleUpdate(float tpf) {
 // first-person: keep the audio listener moving with the camera
 listener.setLocation(cam.getLocation());
 listener.setRotation(cam.getRotation());
}
```

# Setting Audio Environment Properties

Optionally, You can choose from the following environmental presets from `com.jme3.audio.Environment`. This presets influence subtle echo effects (reverb) that evoke associations of different environments in your users. That is, it makes you scene sound “indoors” or “outdoors” etc. You use Audio Environments together with `setReverbEnabled(true)` on positional AudioNodes (see above).

Environment	density	diffusion	gain	gainHf	decayTime	decayHf
Garage	1.00f	1.0f	1.0f	1.00f	0.90f	0.5f
Dungeon	0.75f	1.0f	1.0f	0.75f	1.60f	1.0f
Cavern	0.50f	1.0f	1.0f	0.50f	2.25f	1.0f
AcousticLab	0.50f	1.0f	1.0f	1.00f	0.28f	1.0f
Closet	1.00f	1.0f	1.0f	1.00f	0.15f	1.0f

## 1. Activate a Environment preset

- Either use a default, e.g. make your scene sounds like a dungeon environment:

```
audioRenderer.setEnvironment(new Environment(Environment.Dun
```

- Or activate [custom environment settings](#) in the Environment constructor:

```
audioRenderer.setEnvironment(
 new Environment(density, diffusion, gain, gainHf, d
 reflGain, reflDelay, lateGain, lateDelay))
```

## 2. Activate 3D audio for certain sounds:

```
footstepsAudio.setPositional(true);
footstepsAudio.setReverbEnabled(true);
```

A sound engineer can create a custom `com.jme3.audio.Environment` object and specify custom environment values such as density, diffusion, gain, decay, delay... You can find many [examples of custom audio environment presets](#) here.

Advanced users find more info about OpenAL and its features here: [OpenAL 1.1 Specification](#).

It depends on the hardware whether audio effects are supported (if not, you get the message `OpenAL EFX not available! Audio effects won't work.` )

[sound](#), [documentation](#), [environment](#)

</div>

## title: Bloom and Glow

# Bloom and Glow

Bloom is a popular shader effect in 3D games industry. It usually consist in displaying a glowing halo around light sources or bright areas of a scene. In practice, the bright areas are extracted from the rendered scene, blurred and finally added up to the render.

Those images gives an idea of what bloom does. The left image has no bloom effect, the right image does.



# Bloom Usage

1. Create a FilterPostProcessor
2. Create a BloomFilter
3. Add the filter to the processor
4. Add the processor to the viewPort

```
FilterPostProcessor fpp=new FilterPostProcessor(assetManager);
BloomFilter bloom=new BloomFilter();
fpp.addFilter(bloom);
viewPort.addProcessor(fpp);
```

Here are the parameters that you can tweak :

Parameter	Method	Default	Description
blur scale	setBlurScale(float)	1.5f	the scale of the bloom effect, but be careful, high values does artifacts
exposure Power	setExposurePower(float)	5.0f	the glowing channel color is raised to the value power
exposure cut-off	setExposureCutoff(float)	0.0f	the threshold of color to bloom during extraction
bloom intensity	setBloomIntensity(float)	2.0f	the resulting bloom value is multiplied by this intensity

You'll probably need to adjust those parameters depending on your scene.

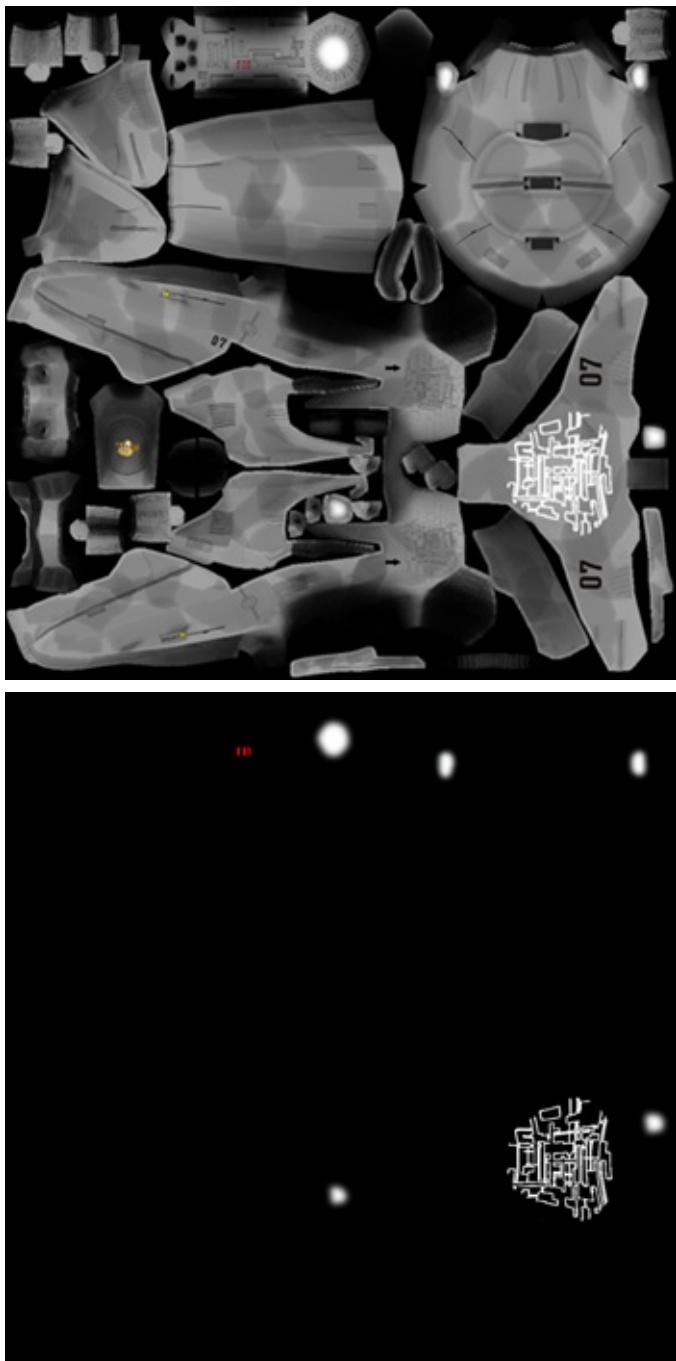
# Bloom with a glow map

Sometimes, you want to have more control over what glows and does not glow. The bloom filter supports a glow map or a glow color.

## Creating a glow-map

Let's take the hover tank example bundled with JME3 test data.

Here you can see the diffuse map of the tank, and the associated glow map that only contains the parts of the texture that will glow and their glowing color:



Glow maps work with `Lighting.j3md`, `Particles.j3md` and `SolidColor.j3md` material definitions.  
The tank material looks like this :

```

Material My Material : Common/MatDefs/Light/Lighting.j3md {
 MaterialParameters {
 SpecularMap : Models/HoverTank/tank_specular.png
 Shininess : 8
 NormalMap : Models/HoverTank/tank_normals.png
 DiffuseMap : Models/HoverTank/tank_diffuse.png
 GlowMap : Models/HoverTank/tank_glow_map_highres.png
 UseMaterialColors : true
 Ambient : 0.0 0.0 0.0 1.0
 Diffuse : 1.0 1.0 1.0 1.0
 Specular : 1.0 1.0 1.0 1.0
 }
}

```

The glow map is defined here : **GlowMap :**

**Models/HoverTank/tank\_glow\_map\_highres.png**

## Usage

1. Create a FilterPostProcessor
2. Create a BloomFilter with the GlowMode.Objects parameter
3. Add the filter to the processor
4. Add the processor to the viewPort

```

FilterPostProcessor fpp=new FilterPostProcessor(assetManager);
BloomFilter bf=new BloomFilter(BloomFilter.GlowMode.Objects);
fpp.addFilter(bf);
viewPort.addProcessor(fpp);

```

Here is the result :



## Bloom with a glow color

Sometimes you need an entire object to glow, not just parts of it. In this case you'll need to use the glow color parameter.

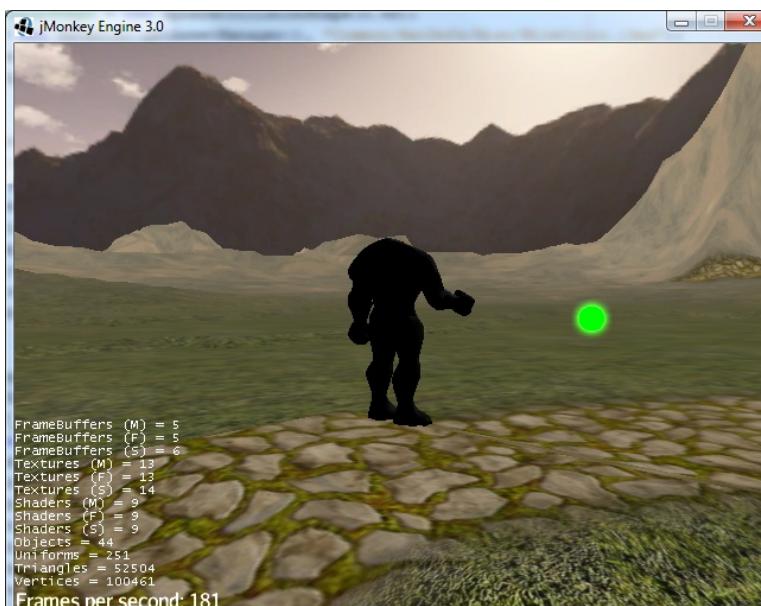
## Usage

1. Create a material for your object and set the GlowColor parameter
2. Create a FilterPostProcessor
3. Create a BloomFilter with the GlowMode.Objects parameter
4. Add the filter to the processor
5. Add the processor to the viewPort

```
Material mat = new Material(assetManager, "Common/MatDefs/
mat.setColor("Color", ColorRGBA.Green);
mat.setColor("GlowColor", ColorRGBA.Green);
fpp=new FilterPostProcessor(assetManager);
bloom= new BloomFilter(BloomFilter.GlowMode.Objects);
fpp.addFilter(bloom);
viewPort.addProcessor(fpp);
```



Here is the result on Oto's plasma ball (before and after) :



## Hints and tricks

### Increasing the blur range and reducing fps cost

The glow render is sampled on a texture that has the same dimensions as the viewport. You can reduce the size of the bloom sampling just by using the `setDownSamplingFactor` method like this :

```
BloomFilter bloom=new BloomFilter();
bloom.setDownSamplingFactor(2.0f);
```

In this example the sampling size is divided by 4 (width/2,height/2), resulting in less work to blur the scene. The resulting texture is then up sampled to the screen size using hardware bilinear filtering. this results in a wider blur range.

### Using classic bloom combined with a glow map

let's say you want a global bloom on your scene, but you have also a glowing object on it. You can use only one bloom filter for both effects like that

```
BloomFilter bloom=new BloomFilter(BloomFilter.GlowMode.SceneAndObj
```

However, note that both effects will share the same values of attribute, and sometimes, it won't be what you need.

### Making your home brewed material definition support Glow

Let's say you have made a custom material on your own, and that you want it to support glow maps and glow color. In your material definition you need to add those lines in the MaterialParameters section :

```
MaterialParameters {

 // Texture of the glowing parts of the material
 Texture2D GlowMap
 // The glow color of the object
 Color GlowColor
}
```

Then add the following technique :

```
Technique Glow {

 LightMode SinglePass

 VertexShader GLSL100: Common/MatDefs/Misc/SimpleTextured.
 FragmentShader GLSL100: Common/MatDefs/Light/Glow.frag

 WorldParameters {
 WorldViewProjectionMatrix
 }

 Defines {
 HAS_GLOWMAP : GlowMap
 HAS_GLOWCOLOR : GlowColor
 }
}
```

Then you can use this material with the BloomFilter

### Make a glowing object stop to glow

If you are using a glow map, remove the texture from the material.

```
material.clearTextureParam("GlowMap");
```

If you are using a glow color, set it to black

```
material.setColor("GlowColor", ColorRGBA.Black);
```

[documentation](#), [effect](#), [light](#)

</div>

## **title: How to Build the jNavigation Recast Bindings**

# **How to Build the jNavigation Recast Bindings**

jNavigation is Java jME port for Recast Navigation written in C++. The project has two parts:

1. [jNavigationNative](#) contains Recast Navigation library and C++ wrapper for java
2. [jNavigation](#) is Java project that uses jNavigationNative and is the project that the end user will use

If there is need for updating Recast Navigation from native side, there are two kinds of updating bindings:

1. only updating methods as the Recast made more efficient or more precise
2. adding new methods for Recast use

## **Updating methods**

Only updating methods are easy. The requirements needed:

- C++ compiler

The jNavigationNative that has following folders and files (it has more, but these are the most important for now):

- Recast
  - Include
  - Source
- README.md
- Recast\_wrap.cxx - Java - C++ wrapper

Updating the methods is only the matter of putting all headers from Recast Navigation to Include folder, source to Source folders, and then building the project.

As author of this project used the NetBeans 7.4 for building the project, the following instruction is included, if the building from terminal doesn't work.

1. Setting [parameters for NetBeans compiler](#)
2. Remove all headers from Header Files
3. Remove all source files **EXCEPT Recast\_wrap.cxx** from Source Files
4. Right click on Header files, select `Add Existing Item...` or `Add Existing Items from Folders...` and select needed headers
5. Right click on Source files, select `Add Existing Item...` or `Add Existing Items from Folders...` and select needed source files
6. Build
7. Add built project to jNavigation project
8. Build jNavigation project

## Adding new methods from native side

This is more complicated process and it includes the all work done in NetBeans mentioned in previous section. After that, there are two ways to add new function:

- manually adding code to wrapper
- creating new wrapper with [SWIG](#)

## Manually adding code to wrapper

Example of method in wrapper:

```
SWIGEXPORT jint JNICALL Java_com_jme3_ai_navigation_utils_RecastJNI
 ...
}
```

The `Recast_wrap.cxx` uses SWIG wrapper so for declaring method in wrapper you must first use the keyword `SWIGEXPORT` then returning type (for more information on returning types see [link](#)), then again keyword `JNICALL` and then as the name of method

`Java_com_jme3_ai_navigation_utils_RecastJNI_ + name of class + name of method`, after that, there goes list of parameters needed for the function (for more information see [link](#)). In body of method write how the function should be used.

After adding method to wrapper, compile project and add it to jNavigation. In jNavigation project in class `com.jme3.ai.navigation.utils.RecastJNI.java` add native method, and after that add in class from which you would like to use this method to call this native method. It seems a little bit more complicated than it should be, but this also for support for updating native side with SWIG.

## Creating new wrapper with SWIG

In some temporary folder add all headers. It shouldn't contain any subfolders.

The following script was used for generating wrapper:

```
%module Recast
%include "carrays.i"
%array_class(double, DoubleArray);
%array_class(float, FloatArray);
%array_class(int, IntArray);
%array_class(unsigned char, UCharArray);
%array_class(unsigned short, USHORTArray);
%array_class(unsigned int, UIntArray);
%array_class(long, LongArray);
%array_class(bool, BooleanArray)

%{
#include "DetourAlloc.h"
#include "DetourAssert.h"
#include "DetourCommon.h"
#include "DetourCrowd.h"
#include "DetourLocalBoundary.h"
#include "DetourMath.h"
#include "DetourNavMesh.h"
#include "DetourNavMeshBuilder.h"
#include "DetourNavMeshQuery.h"
#include "DetourNode.h"
#include "DetourObstacleAvoidance.h"
#include "DetourPathCorridor.h"
#include "DetourPathQueue.h"
#include "DetourProximityGrid.h"
#include "DetourStatus.h"
#include "DetourTileCache.h"
#include "DetourTileCacheBuilder.h"
#include "Recast.h"
#include "RecastAlloc.h"
#include "RecastAssert.h"
%}

/* Let's just grab the original header file here */
```

```
%include "DetourAlloc.h"
%include "DetourAssert.h"
%include "DetourCommon.h"
%include "DetourCrowd.h"
%include "DetourLocalBoundary.h"
%include "DetourMath.h"
%include "DetourNavMesh.h"
%include "DetourNavMeshBuilder.h"
%include "DetourNavMeshQuery.h"
%include "DetourNode.h"
%include "DetourObstacleAvoidance.h"
%include "DetourPathCorridor.h"
%include "DetourPathQueue.h"
%include "DetourProximityGrid.h"
%include "DetourStatus.h"
%include "DetourTileCache.h"
%include "DetourTileCacheBuilder.h"
%include "Recast.h"
%include "RecastAlloc.h"
%include "RecastAssert.h"

%pragma(java) jniclasscode=%{
 static {
 System.load("Recast");
 }
%}
```

If there are more headers at some moment, include them in both places.

1. Save script as Recast.i into temp folder with rest of the headers
2. Install SWIG if not already
3. Open terminal and go to folder where the script is
4. Execute command `swig -c++ -java Recast.i`
5. Now SWIG will generate Java classes and new Recast\_wrap.cxx
6. Recast\_wrap.cxx put in jNavigationNative with new headers and source files, as previously mentioned
7. Build that project
8. For jNavigation side, put only new methods in RecastJNI, and use where they are being used. For that you can see in Java class that are build with SWIG.
9. If method uses some explicit SWIG type, try to use some method for converting it into

jME type, or similar. You can probably find something in package  
com.jme3.ai.navigation.utils

</div>

# title: Multithreading Bullet Physics in jme3

## Multithreading Bullet Physics in jme3

### Introduction

Since bullet is not (yet) multithreaded or GPU accelerated, the jME3 implementation allows to run each physics space on a separate thread that is executed in parallel to rendering.

### How is it handled in jme3 and bullet?

A SimpleApplication with a BulletAppState allows setting the threading type via

```
setThreadingType(ThreadingType type);
```

where ThreadingType can be either SEQUENTIAL or PARALLEL. By default, it's SEQUENTIAL.

You can activate PARALLEL threading in the simpleInitApp() method:

```
bulletAppState = new BulletAppState();
bulletAppState.setThreadingType(BulletAppState.ThreadingType.PARALLEL);
stateManager.attach(bulletAppState);
```

Now the physics update happens in parallel to render(), that is, after the user's changes in the update() call have been applied. During update() the physics update loop pauses. This way the loop logic is still maintained: the user can set and change values in physics and scenegraph objects before render() and physicsUpdate() are called in parallel. This allows you to use physics methods in update() as if it was single-threaded.

PARALLEL	SEQUENTIAL
1. update(), 2. render() and physics update().	1. update(), 2. render(), 3. physics update().
Physics Debug View is rendered inaccurately (out of sync)	Physics Debug View is rendered accurately.

You can add more physics spaces by using multiple PARALLEL bulletAppStates. You would do that if you have sets physical objects that never collide (for example, underground boulders and flying cannon balls above ground), so you put those into separate physics spaces, which improves performances (less collisions to check!).

[documentation](#), [physics](#), [threading](#)

</div>

## **title: The jME3 Camera**

# **The jME3 Camera**

Note that by default, the mouse pointer is invisible, and the mouse is set up to control the camera rotation.

## **Default Camera**

The default `com.jme3.renderer.Camera` object is `cam` in `com.jme3.app.Application`.

The camera object is created with the following defaults:

- Width and height are set to the current Application's `settings.getWidth()` and `settings.getHeight()` values.
- Frustum Perspective:
  - Frame of view angle of 45° along the Y axis
  - Aspect ratio of width divided by height
  - Near view plane of 1 wu
  - Far view plane of 1000 wu
- Start location at (0f, 0f, 10f).
- Start direction is looking at the origin.

Method	Usage
cam.getLocation(), setLocation()	The camera position
cam.getRotation(), setRotation()	The camera rotation
cam.getLeft(), setLeft()	The left axis of the camera
cam.getUp(), setUp()	The up axis of the camera, usually Vector3f(0,1,0)
cam.getDirection()	The vector the camera is facing
cam.getAxes(), setAxes(left,up,dir)	One accessor for the three properties left/up/direction.
cam.getFrame(), setFrame(loc,left,up,dir)	One accessor for the four properties location/left/up/direction.
cam.resize(width, height, fixAspect)	Resize an existing camera object while keeping all other settings. Set fixAspect to true to adjust the aspect ratio (?)
cam.setFrustum( near, far, left, right, top, bottom )	The frustum is defined by the near/far plane, left/right plane, top/bottom plane (all distances as float values)
cam.setFrustumPerspective( fovY, aspect ratio, near, far)	The frustum is defined by view angle along the Y axis (in degrees), aspect ratio, and the near/far plane.
cam.lookAt(target,up)	Turn the camera to look at Coordinate target, and rotate it around the up axis.
cam.setParallelProjection(false)	Normal perspective
cam.setParallelProjection(true)	Parallel projection perspective
cam.getScreenCoordinates()	?

**Tip:** After you change view port, frustum, or frame, call `cam.update();`

## FlyBy Camera

The `flyCam` class field gives you access to an AppState that extends the default camera in `com.jme3.app.SimpleApplication` with more features. The input manager of the `com.jme3.input.FlyByCamera` AppState is preconfigured to respond to the WASD keys for walking forwards and backwards, and strafing to the sides; move the mouse to rotate the camera (“Mouse Look”), scroll the mouse wheel for zooming in or out. The QZ keys raise or lower the camera vertically.

```

Q W up forw
A S D --> left back right
Z down

```

Method	Usage
<code>flyCam.setEnabled(true);</code>	Activate the flyby cam
<code>flyCam.setMoveSpeed(10);</code>	Control the move speed
<code>flyCam.setRotationSpeed(10);</code>	Control the rotation speed
<code>flyCam.setDragToRotate(true)</code>	Forces the player to keep mouse button pressed to rotate camera, typically used for Applets. If false (default), all mouse movement will be captured and interpreted as rotations.

The FlyByCamera is active by default, but you can change all these defaults for your game.

## Chase Camera

jME3 also supports an optional Chase Cam that can follow a moving target Spatial (`com.jme3.input.ChaseCamera`). When you use the chase cam, the player clicks and hold the mouse button to rotate the camera around the camera target. You can use a chase cam if you need the mouse pointer visible in your game.

```

flyCam.setEnabled(false);
ChaseCamera chaseCam = new ChaseCamera(cam, target, inputManager);

```

Method	Usage
chaseCam.setSmoothMotion(true);	Interpolates a smoother acceleration/deceleration when the camera moves.
chaseCam.setChasingSensitivity(5f)	The lower the chasing sensitivity, the slower the camera will follow the target when it moves.
chaseCam.setTrailingSensitivity(0.5f)	The lower the trailing sensitivity, the slower the camera will begin to go after the target when it moves. Default is 0.5;
chaseCam.setRotationSensitivity(5f)	The lower the sensitivity, the slower the camera will rotate around the target when the mouse is dragged. Default is 5.
chaseCam.setTrailingRotationInertia(0.1f)	This prevents the camera to stop too abruptly when the target stops rotating before the camera has reached the target's trailing position. Default is 0.1f.
chaseCam.setDefaultDistance(40);	The default distance to the target at the start of the application.
chaseCam.setMaxDistance(40);	The maximum zoom distance. Default is 40f.
chaseCam.setMinDistance(1);	The minimum zoom distance. Default is 1f.
chaseCam.setMinVerticalRotation(-FastMath.PI/2);	The minimal vertical rotation angle of the camera around the target. Default is 0.
chaseCam.setDefaultVerticalRotation(-FastMath.PI/2);	The default vertical rotation angle of the camera around the target at the start of the application.
chaseCam.setDefaultHorizontalRotation(-FastMath.PI/2);	The default horizontal rotation angle of the camera around the target at the start of the application.

[camera](#), [documentation](#)

</div>

## **title: Capture Audio/Video to a File**

# **Capture Audio/Video to a File**

So you've made your cool new JMonkeyEngine3 game and you want to create a demo video to show off your hard work. Or maybe you want to make a cutscene for your game using the physics and characters in the game itself. Screen capturing is the most straightforward way to do this, but it can slow down your game and produce low-quality video and audio as a result. A better way is to record video and audio directly from the game while it is running using `VideoRecorderAppState`.

Combine this method with jMonkeyEngine's [Cinematics](#) feature to record high-quality game trailers!

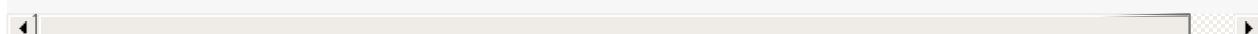
</div>

## **Simple Way**

First off, if all you need is to record video at 30fps with no sound, then look no further than jMonkeyEngine 3's built in `VideoRecorderAppState` class.

Add the following code to your `simpleInitApp()` method.

```
stateManager.attach(new VideoRecorderAppState()); //start recording
```



The game will run slow, but the recording will be in high-quality and normal speed. Recording starts when the state is attached, and ends when the application quits or the state is detached.

The video files will be stored in your **user home directory**. If you want to save to another path, specify a `File` object in the `VideoRecorderAppState` constructor.

That's all!

## **Advanced Way**

This way of A/V recording is still in development. It works for all of jMonkeyEngine's test

cases. If you experience any problems or if something isn't clear, please [let me know](#). – bortreb

If you want to record audio as well, record at different framerates, or record from multiple viewpoints at once, then there's a full solution for doing this already made for you here:

<http://www.aurellem.com/releases/jmeCapture-latest.zip>

<http://www.aurellem.com/releases/jmeCapture-latest.tar.bz2>

Download the archive in your preferred format, extract, add the jars to your project, and you are ready to go.

The javadoc is here: <http://www.aurellem.com/jmeCapture/docs/>

To capture video and audio you use the `com.aurellem.capture.Capture` class, which has two methods, `captureAudio()` and `captureVideo()`, and the `com.aurellem.capture.IsoTimer` class, which sets the audio and video framerate.

The steps are:

```
yourApp.setTimer(new IsoTimer(desiredFramesPerSecond));
```

This causes jMonkeyEngine to take as much time as it needs to fully calculate every frame of the video and audio. You will see your game speed up and slow down depending on how computationally demanding your game is, but the final recorded audio and video will be perfectly synchronized and will run at exactly the fps which you specified.

```
captureVideo(yourApp, targetVideoFile);
captureAudio(yourApp, targetAudioFile);
```

These will cause the app to record audio and video when it is run. Audio and video will stop being recorded when the app stops. Your audio will be recorded as a 44,100 Hz linear PCM wav file, while the video will be recorded according to the following rules:

1.) (Preferred) If you supply an empty directory as the file, then the video will be saved as a sequence of .png files, one file per frame. The files start at 0000000.png and increment from there. You can then combine the frames into your preferred container/codec. If the directory is not empty, then writing video frames to it will fail, and nothing will be written.

2.) If the filename ends in ".avi" then the frames will be encoded as a RAW stream inside an AVI 1.0 container. The resulting file will be quite large and you will probably want to re-encode it to your preferred container/codec format. Be advised that some video players cannot process AVI with a RAW stream, and that AVI 1.0 files generated by this method that

exceed 2.0GB are invalid according to the AVI 1.0 [spec](#) (but many programs can still deal with them.) Thanks to [Werner Randelshofer](#) for his excellent work which made the AVI file writer option possible.

3.) Any non-directory file ending in anything other than “.avi” will be processed through Xuggle. Xuggle provides the option to use many codecs/containers, but you will have to install it on your system yourself in order to use this option. Please visit <http://www.xuggle.com/> to learn how to do this.

Note that you will not hear any sound if you choose to record sound to a file.

</div>

## Basic Example

Here is a complete example showing how to capture both audio and video from one of jMonkeyEngine3's advanced demo applications.

```
import java.io.File;
import java.io.IOException;

import jme3test.water.TestPostWater;

import com.aurellem.capture.Capture;
import com.aurellem.capture.IsoTimer;
import com.jme3.app.SimpleApplication;

/**
 * Demonstrates how to use basic Audio/Video capture with a
 * jMonkeyEngine application. You can use these techniques to make
 * high quality cutscenes or demo videos, even on very slow laptops
 *
 * @author Robert McIntyre
 */

public class Basic {

 public static void main(String[] ignore) throws IOException{
 File video = File.createTempFile("JME-water-video", ".avi");
 File audio = File.createTempFile("JME-water-audio", ".wav");

 SimpleApplication app = new TestPostWater();
 app.setTimer(new IsoTimer(60));
 app.setShowSettings(false);

 Capture.captureVideo(app, video);
 Capture.captureAudio(app, audio);

 app.start();

 System.out.println(video.getCanonicalPath());
 System.out.println(audio.getCanonicalPath());
 }
}
```

## How it works

A standard JME3 application that extends `SimpleApplication` or `Application` tries as hard as it can to keep in sync with *user-time*. If a ball is rolling at 1 game-mile per game-hour in the game, and you wait for one user-hour as measured by the clock on your wall, then the ball should have traveled exactly one game-mile. In order to keep sync with the real world, the game throttles its physics engine and graphics display. If the computations involved in running the game are too intense, then the game will first skip frames, then sacrifice physics accuracy. If there are particularly demanding computations, then you may only get 1 fps, and the ball may tunnel through the floor or obstacles due to inaccurate physics simulation, but after the end of one user-hour, that ball will have traveled one game-mile.

When we're recording video, we don't care if the game-time syncs with user-time, but instead whether the time in the recorded video (video-time) syncs with user-time. To continue the analogy, if we recorded the ball rolling at 1 game-mile per game-hour and watched the video later, we would want to see 30 fps video of the ball rolling at 1 video-mile per *user-hour*. It doesn't matter how much user-time it took to simulate that hour of game-time to make the high-quality recording.

The IsoTimer ignores real-time and always reports that the same amount of time has passed every time it is called. That way, one can put code to write each video/audio frame to a file without worrying about that code itself slowing down the game to the point where the recording would be useless.

</div>

## Advanced Example

The package from aurellem.com was made for AI research and can do more than just record a single stream of audio and video. You can use it to:

- 1.) Create multiple independent listeners that each hear the world from their own perspective.
- 2.) Process the sound data in any way you wish.
- 3.) Do the same for visual data.

Here is a more advanced example, which can also be found along with other examples in the `jmeCapture.jar` file included in the distribution.

```
package com.aurellem.capture.examples;

import java.io.File;
```

```
import java.io.IOException;
import java.lang.reflect.Field;
import java.nio.ByteBuffer;

import javax.sound.sampled.AudioFormat;

import org.tritonus.share.sampled.FloatSampleTools;

import com.aurellem.capture.AurellemSystemDelegate;
import com.aurellem.capture.Capture;
import com.aurellem.capture.IsoTimer;
import com.aurellem.capture.audio.CompositeSoundProcessor;
import com.aurellem.capture.audio.MultiListener;
import com.aurellem.capture.audio.SoundProcessor;
import com.aurellem.capture.audio.WaveFileWriter;
import com.jme3.app.SimpleApplication;
import com.jme3.audio.AudioNode;
import com.jme3.audio.Listener;
import com.jme3.cinematic.MotionPath;
import com.jme3.cinematic.events.AbstractCinematicEvent;
import com.jme3.cinematic.events.MotionTrack;
import com.jme3.material.Material;
import com.jme3.math.ColorRGBA;
import com.jme3.math.FastMath;
import com.jme3.math.Quaternion;
import com.jme3.math.Vector3f;
import com.jme3.scene.Geometry;
import com.jme3.scene.Node;
import com.jme3.scene.shape.Box;
import com.jme3.scene.shape.Sphere;
import com.jme3.system.AppSettings;
import com.jme3.system.JmeSystem;

/**
 *
 * Demonstrates advanced use of the audio capture and recording
 * features. Multiple perspectives of the same scene are
 * simultaneously rendered to different sound files.
 *
 * A key limitation of the way multiple listeners are implemented i
```

```
* that only 3D positioning effects are realized for listeners other
* than the main LWJGL listener. This means that audio effects such
* as environment settings will *not* be heard on any auxiliary
* listeners, though sound attenuation will work correctly.
*
* Multiple listeners as realized here might be used to make AI
* entities that can each hear the world from their own perspective
*
* @author Robert McIntyre
*/
public class Advanced extends SimpleApplication {

 /**
 * You will see three grey cubes, a blue sphere, and a path which
 * circles each cube. The blue sphere is generating a constant
 * monotone sound as it moves along the track. Each cube is
 * listening for sound; when a cube hears sound whose intensity
 * greater than a certain threshold, it changes its color from
 * grey to green.
 *
 * Each cube is also saving whatever it hears to a file. The
 * scene from the perspective of the viewer is also saved to a
 * video file. When you listen to each of the sound files
 * alongside the video, the sound will get louder when the sphere
 * approaches the cube that generated that sound file. This
 * shows that each listener is hearing the world from its own
 * perspective.
 *
 */
 public static void main(String[] args) {
 Advanced app = new Advanced();
 AppSettings settings = new AppSettings(true);
 settings.setAudioRenderer(AurellemSystemDelegate.SEND);
 JmeSystem.setSystemDelegate(new AurellemSystemDelegate());
 app.setSettings(settings);
 app.setShowSettings(false);
 app.setPauseOnLostFocus(false);
 }
}
```

```
try {
 Capture.captureVideo(app, File.createTempFile("advanced",
 Capture.captureAudio(app, File.createTempFile("advanced
})
catch (IOException e) {e.printStackTrace();}

app.start();
}

private Geometry bell;
private Geometry ear1;
private Geometry ear2;
private Geometry ear3;
private AudioNode music;
private MotionTrack motionControl;
private IsoTimer motionTimer = new IsoTimer(60);

private Geometry makeEar(Node root, Vector3f position){
 Material mat = new Material(assetManager, "Common/MatDefs/M
 Geometry ear = new Geometry("ear", new Box(1.0f, 1.0f, 1.0f
 ear.setLocalTranslation(position);
 mat.setColor("Color", ColorRGBA.Green);
 ear.setMaterial(mat);
 root.attachChild(ear);
 return ear;
}

private Vector3f[] path = new Vector3f[]{
 // loop 1
 new Vector3f(0, 0, 0),
 new Vector3f(0, 0, -10),
 new Vector3f(-2, 0, -14),
 new Vector3f(-6, 0, -20),
 new Vector3f(0, 0, -26),
 new Vector3f(6, 0, -20),
 new Vector3f(0, 0, -14),
 new Vector3f(-6, 0, -20),
 new Vector3f(0, 0, -26),
 new Vector3f(6, 0, -20),
```

```
// loop 2
new Vector3f(5, 0, -5),
new Vector3f(7, 0, 1.5f),
new Vector3f(14, 0, 2),
new Vector3f(20, 0, 6),
new Vector3f(26, 0, 0),
new Vector3f(20, 0, -6),
new Vector3f(14, 0, 0),
new Vector3f(20, 0, 6),
new Vector3f(26, 0, 0),
new Vector3f(20, 0, -6),
new Vector3f(14, 0, 0),
// loop 3
new Vector3f(8, 0, 7.5f),
new Vector3f(7, 0, 10.5f),
new Vector3f(6, 0, 20),
new Vector3f(0, 0, 26),
new Vector3f(-6, 0, 20),
new Vector3f(0, 0, 14),
new Vector3f(6, 0, 20),
new Vector3f(0, 0, 26),
new Vector3f(-6, 0, 20),
new Vector3f(0, 0, 14),
// begin ellipse
new Vector3f(16, 5, 20),
new Vector3f(0, 0, 26),
new Vector3f(-16, -10, 20),
new Vector3f(0, 0, 14),
new Vector3f(16, 20, 20),
new Vector3f(0, 0, 26),
new Vector3f(-10, -25, 10),
new Vector3f(-10, 0, 0),
// come at me!
new Vector3f(-28.00242f, 48.005623f, -34.648228f),
new Vector3f(0, 0 , -20),
};

private void createScene() {
 Material mat = new Material(assetManager, "Common/MatDefs/M
bell = new Geometry("sound-emitter" , new Sphere(15,15,1))
```

```
mat.setColor("Color", ColorRGBA.Blue);
bell.setMaterial(mat);
rootNode.attachChild(bell);

ear1 = makeEar(rootNode, new Vector3f(0, 0 , -20));
ear2 = makeEar(rootNode, new Vector3f(0, 0 , 20));
ear3 = makeEar(rootNode, new Vector3f(20, 0 , 0));

MotionPath track = new MotionPath();

for (Vector3f v : path){
 track.addWayPoint(v);
}
track.setCurveTension(0.80f);

motionControl = new MotionTrack(bell,track);
// for now, use reflection to change the timer...
// motionControl.setTimer(new IsoTimer(60));

try {
 Field timerField;
 timerField = AbstractCinematicEvent.class.getDeclaredField("timer");
 timerField.setAccessible(true);
 try {timerField.set(motionControl, motionTimer);}
 catch (IllegalArgumentException e) {e.printStackTrace()}
 catch (IllegalAccessException e) {e.printStackTrace();}
}

catch (SecurityException e) {e.printStackTrace();}
catch (NoSuchFieldException e) {e.printStackTrace();}

motionControl.setDirectionType(MotionTrack.Direction.PathArc);
motionControl.setRotation(new Quaternion().fromAngleNormalAxis(0, 1, 0, 0));
motionControl.setInitialDuration(20f);
motionControl.setSpeed(1f);

track.enableDebugShape(assetManager, rootNode);
positionCamera();
}
```

```
private void positionCamera(){
 this.cam.setLocation(new Vector3f(-28.00242f, 48.005623f, -10.0f));
 this.cam.setRotation(new Quaternion(0.3359635f, 0.34280345f, 0.0f, 1.0f));
}

private void initAudio() {
 org.lwjgl.input.Mouse.setGrabbed(false);
 music = new AudioNode(assetManager, "Sound/Effects/Beep.ogg");

 rootNode.attachChild(music);
 audioRenderer.playSource(music);
 music.setPositional(true);
 music.setVolume(1f);
 music.setReverbEnabled(false);
 music.setDirectional(false);
 music.setMaxDistance(200.0f);
 music.setRefDistance(1f);
 //music.setRolloffFactor(1f);
 music.setLooping(false);
 audioRenderer.pauseSource(music);
}

public class Dancer implements SoundProcessor {
 Geometry entity;
 float scale = 2;
 public Dancer(Geometry entity){
 this.entity = entity;
 }

 /**
 * this method is irrelevant since there is no state to clear
 */
 public void cleanup() {}

 /**
 * Respond to sound! This is the brain of an AI entity that
 * hears its surroundings and reacts to them.
 */
}
```

```

 public void process(ByteBuffer audioSamples, int numSamples)
 audioSamples.clear();
 byte[] data = new byte[numSamples];
 float[] out = new float[numSamples];
 audioSamples.get(data);
 FloatSampleTools.byte2floatInterleaved(data, 0, out, 0,
 numSamples/format.getFrameSize(), format);

 float max = Float.NEGATIVE_INFINITY;
 for (float f : out){if (f > max) max = f;}
 audioSamples.clear();

 if (max > 0.1){entity.getMaterial().setColor("Color", ColorRGBA.Orange);
 else {entity.getMaterial().setColor("Color", ColorRGBA.Blue);
 }

 }

private void prepareEar(Geometry ear, int n){
 if (this.audioRenderer instanceof MultiListener){
 MultiListener rf = (MultiListener)this.audioRenderer;

 Listener auxListener = new Listener();
 auxListener.setLocation(ear.getLocalTranslation());

 rf.addListener(auxListener);
 WaveFileWriter aux = null;

 try {aux = new WaveFileWriter(File.createTempFile("advan",
 catch (IOException e) {e.printStackTrace();}

 rf.registerSoundProcessor(auxListener,
 new CompositeSoundProcessor(new Dancer(ear), aux));
 }

}

public void simpleInitApp() {
 this.setTimer(new IsoTimer(60));
 initAudio();
}

```

```

 createScene();

 prepareEar(ear1, 1);
 prepareEar(ear2, 1);
 prepareEar(ear3, 1);

 motionControl.play();

 }

 public void simpleUpdate(float tpf) {
 motionTimer.update();
 if (music.getStatus() != AudioSource.Status.Playing){
 music.play();
 }
 Vector3f loc = cam.getLocation();
 Quaternion rot = cam.getRotation();
 listener.setLocation(loc);
 listener.setRotation(rot);
 music.setLocalTranslation(bell.getLocalTranslation());
 }

}

```

[oCEfK0yhDrY?.swf](#)

</div>

## Using Advanced features to Record from more than one perspective at once

[WIJt9aRGusc?.swf](#)

</div>

## More Information

This is the old page showing the first version of this idea

<http://aurellem.org/cortex/html/capture-video.html>

All source code can be found here:

<http://hg.bortreb.com/audio-send>

<http://hg.bortreb.com/jmeCapture>

More information on the modifications to OpenAL to support multiple listeners can be found here.

<http://aurellem.org/audio-send/html/ear.html>

</div>

## title: JME3 Cinematics

# JME3 Cinematics

JME3 cinematics (com.jme.cinematic) allow you to remote control nodes and cameras in a 3D game: You can script and play cinematic scenes. You can use cinematics to create [cutscenes](#) and movies/trailers for your game. Another good use case is efficient “destruction physics”: Playing back prerecorded flying pieces of debris for demolitions is much faster than calculating them with live physics.

Internally, Cinematics are implemented as [AppStates](#).

Short overview of the cinematic process:

1. Plan the script of your movie.  
Write down a timeline (e.g. on paper) of which character should be at which spot at which time.
2. Attach the scene objects that you want to remote-control to one Node.  
This Node can be the rootNode, or a Node that is attached to the rootNode.
3. Create a Cinematic object for this movie scene. The Cinematic will contain and manage the movie script.
4. For each line in your script (for each keyframe in your timeline), add a CinematicEvent to the Cinematic.

## Sample Code

- [TestCinematic.java](#)

## How to Use a Cinematic

A Cinematic is like a movie script for a node.

```
Cinematic cinematic = new Cinematic(sceneNode, duration);
cinematic.addCinematicEvent(starttime1, event1);
cinematic.addCinematicEvent(starttime2, event2);
cinematic.addCinematicEvent(starttime2, event3);
...
stateManager.attach(cinematic);
```

1. Create one Cinematic per scripted scene.
  - `sceneNode` is the node containing the scene (can be the `rootNode`).
  - `duration` is the duration of the whole scene in seconds.
  - Each Cinematic is a set of CinematicEvents, that are triggered at a given moment on the timeline.
2. Create one CinematicEvent for each line of your movie script.
  - `event` is one motion of a moving object. You can add several events. More details below.
  - `starttime` is the time when this particular cinematic event starts on the timeline. Specify the start time in seconds since the beginning of the cinematic.
3. Attach the Cinematic to the SimpleApplication's stateManager.
4. Play, stop and pause the Cinematic from your code.

Method	Usage
<code>cinematic.play()</code>	Starts playing the cinematic from the start, or from where it was paused.
<code>cinematic.stop()</code>	Stops playing the cinematic and rewinds it.
<code>cinematic.pause()</code>	Pauses the cinematic.

## Events(CinematicEvents)

Just like a movie script consists of lines with instructions to the actors, each Cinematic consists of a series of events.

Here is the list of available CinematicEvents that you use as events. Each track remote-controls scene objects in a different way:

Events(CinematicEvents)	Description
MotionEvent	Use a MotionEvent to move a Spatial non-linearly over time. A MotionEvent is based on a list of waypoints in a MotionPath. The curve goes through each waypoint, and you can adjust the tension of the curve to modify the roundedness of the path. This is the motion interpolation you are going to use in most cases.
SoundEvent	Use a SoundEvent to play a <a href="#">sound</a> at a given time for the given duration.
GuiEvent	Displays a <a href="#">Nifty GUI</a> at a given time for the given duration. Use it to display subtitles or HUD elements. Bind the Nifty GUI XML to the cinematic using <code>cinematic.bindUi("path/to/nifty/file.xml");</code>
AnimationEvent	Use this to start playing a model <a href="#">animation</a> at a given time (a character walking animation for example)

You can add custom events by extending AbstractCinematicEvent.

</div>

## MotionEvent

A MotionEvent moves a Spatial along a complex path.

```
MotionEvent events= new MotionEvent (thingNode, path);
```

Details of the constructor:

- `thingNode` is the Spatial to be moved.
- `path` is a complex [MotionPath](#).

To create a MotionEvent, do the following:

1. [Create a MotionEvent](#)
2. Create a MotionEvent based on the MotionPath.
3. Configure your MotionEvent (see below).
4. Add the MotionEvent to a Cinematic.

MotionEvent configuration method	Usage
<code>event.setLoopMode(LoopMode.Loop)</code>	Sets whether the animation path should loop (LoopMode.Loop) or play only once (LoopMode.Once).
<code>event.setDirectionType(MotionEvent.Direction.None)</code>	Sets the direction behavior of the controlled node. Direct (None) deactivates this feature, while the other values choose from the following: LookAt, Path, PathAndRotation.
<code>event.setDirectionType(MotionEvent.Direction.LookAt)</code>	The spatial turns (rotates) to always face the point it's facing a certain point with its forward vector. Specify the point with the <code>setLookAt()</code> method.
<code>event.setDirectionType(MotionEvent.Direction.Path)</code>	The spatial always faces the direction of the path while moving along it.
<code>event.setDirectionType(MotionEvent.Direction.PathAndRotation)</code>	The spatial faces the direction of the path, plus an added rotation together with the <code>setRotation()</code> method.
<code>event.setDirectionType(MotionEvent.Direction.Rotation)</code>	The spatial spins (rotates) around its own axis while moving. You describe the rotation with a custom quaternion. Use the <code>setRotation()</code> method.
<code>event.setLookAt(teapot.getWorldTranslation(), Vector3f.UNIT_Y)</code>	The spatial always faces the specified location. Use together with <code>setDirectionType(MotionEvent.Direction.LookAt)</code> .
<code>event.setRotation(quaternion)</code>	Sets the rotation. Use together with <code>setDirectionType(MotionEvent.Direction.Rotation)</code> .

**Tip:** Most likely you remote-control more than one object in your scene. Give the events and paths useful names such as `dragonEvent`, `dragonPath`, `heroEvent`, `heroPath`, etc.

</div>

## SoundEvent

A SoundEvent plays a sound as part of the cinematic.

```
SoundEvent(audioPath, isStream, duration, loopMode)
```

Details of the constructor:

- `audioPath` is the path to an audio file as String, e.g. "Sounds/mySound.wav".
- `isStream` toggles between streaming and buffering. Set to true to stream long audio file, set to false to play short buffered sounds.
- `duration` is the time that it should take to play.
- `loopMode` can be `LoopMode.Loop`, `LoopMode.DontLoop`, `LoopMode.Cycle`.

## GuiEvent

A GuiEvent shows or hide a NiftyGUI as part of a cinematic.

```
GuiEvent(screen, duration, loopMode)
```

You must use this together with bindUI() to specify the Nifty GUI XML file that you want to load:

```
cinematic.bindUi("Interface/subtitle.xml");
```

Details of the constructor:

- `screen` is the name of the Nifty GUI screen to load, as String.
- `duration` is the time that it should take to play.
- `loopMode` can be `LoopMode.Loop`, `LoopMode.DontLoop`, `LoopMode.Cycle`.

## AnimationEvent

An AnimationEvent triggers an animation as part of a cinematic.

```
AnimationEvent(thingNode, animationName, duration, loopMode)
```

Details of the constructor:

- `thingNode` is the Spatial whose animation you want to play.
- `animationName` the name of the animation stored in the animated model that you want to trigger, as a String.
- `duration` is the time that it should take to play.
- `loopMode` can be `LoopMode.Loop`, `LoopMode.DontLoop`, `LoopMode.Cycle`.

## Camera Management

There is a built in system for camera switching in Cinematics. It based on CameraNode, and the cinematic just enable the given CameraNode control at a given time.

First you have to bind a camera to the cinematic with a unique name. You'll be provided with a CameraNode

```
CameraNode camNode = cinematic.bindCamera("topView", cam);
```

then you can do whatever you want with this camera node : place it so that you have a the camera angle you'd like, attach it to a motion event to have some camera scrolling, attach control of your own that give it whatever behavior you'd like. In the above example, I want it to be a top view of the scene looking at the world origin.

```
//set its position
camNode.setLocalTranslation(new Vector3f(0, 50, 0));
// set it to look at the world origin
camNode.lookAt(Vector3F.ZERO, Vector3f.UNIT_Y);
```

Then i just have to schedule its activation in the cinematic. I want it to get activated 3 seconds after the start of the cinematic so I just have to do

```
cinematic.activateCamera(3, "topView");
```

## Customizations

You can extend individual CinematicEvents. The [SubtitleTrack.java example](#) shows how to extend a GuiTrack to script subtitles. See how the subtitles are used in the [TestCinematic.java example](#).

You can also create new CinematicEvent by extending [AbstractCinematicEvent](#). An AbstractCinematicEvent implements the CinematicEvent interface and provides duration, time, speed, etc... management. Look at the [TestCinematic.java example](#) is to use this for a custom fadeIn/fadeOut effect in combination with a com.jme3.post.filters.FadeFilter.

```
</div>
```

## Interacting with Cinematics

```
</div>
```

## CinematicEventListener

```
CinematicEventListener cel = new CinematicEventListener() {
 public void onPlay(CinematicEvent cinematic) {
 chaseCam.setEnabled(false);
 System.out.println("play");
 }

 public void onPause(CinematicEvent cinematic) {
 chaseCam.setEnabled(true);
 System.out.println("pause");
 }

 public void onStop(CinematicEvent cinematic) {
 chaseCam.setEnabled(true);
 System.out.println("stop");
 }
}
cinematic.addListener(cel);
```

</div>

## Physics Interaction

Upcoming.

</div>

## title: Collision and Intersection

# Collision and Intersection

The term collision can be used to refer to [physical interactions](#) (where physical objects collide, push and bump off one another), and also to non-physical *intersections* in 3D space. This article is about the non-physical (mathematical) collisions.

Non-physical collision detection is interesting because it uses less computing resources than physical collision detection. The non-physical calculations are faster because they do not have any side effects such as pushing other objects or bumping off of them. Tasks such as [mouse picking](#) are easily implemented using mathematical techniques such as ray casting and intersections. Experienced developers optimize their games by finding ways to simulate certain (otherwise expensive physical) interactions in a non-physical way.

**Example:** One example for an optimization is a physical vehicle's wheels. You could make the wheels fully physical disks, and have jME calculate every tiny force – sounds very accurate? It's total overkill and too slow for a racing game. A more performant solution is to cast four invisible rays down from the vehicle and calculate the intersections with the floor. These non-physical wheels require (in the simplest case) only four calculations per tick to achieve an effect that players can hardly distinguish from the real thing.

## Collidable

The interface `com.jme3.collision.Collidable` declares one method that returns how many collisions were found between two Collidables: `collideWith(Collidable other, CollisionResults results)`.

- A `com.jme3.collision.CollisionResults` object is an `ArrayList` of comparable `com.jme3.collision.CollisionResult` objects.
- You can iterate over the `CollisionResults` to identify the other parties involved in the collision.

Note that jME counts *all* collisions, this means a ray intersecting a box will be counted as two hits, one on the front where the ray enters, and one on the back where the ray exits.

<b>CollisionResults Method</b>	<b>Usage</b>
size()	Returns the number of CollisionResult objects.
getClosestCollision()	Returns the CollisionResult with the lowest distance.
getFarthestCollision()	Returns the CollisionResult with the farthest distance.
getCollision(i)	Returns the CollisionResult at index i.

A CollisionResult object contains information about the second party of the collision event.

<b>CollisionResult Method</b>	<b>Usage</b>
getContactPoint()	Returns the contact point coordinate on the second party, as Vector3f.
getContactNormal()	Returns the Normal vector at the contact point, as Vector3f.
getDistance()	Returns the distance between the Collidable and the second party, as float.
getGeometry()	Returns the Geometry of the second party.
getTriangle(t)	Binds t to the triangle t on the second party's mesh that was hit.
getTriangleIndex()	Returns the index of the triangle on the second party's mesh that was hit.

## Code Sample

Assume you have two collidables a and b and want to detect collisions between them. The collision parties can be Geometries, Nodes with Geometries attached (including the rootNode), Planes, Quads, Lines, or Rays. An important restriction is that you can only collide geometry vs bounding volumes or rays. (This means for example that a must be of Type Node or Geometry and b respectively of Type BoundingBox, BoundingSphere or Ray.)

The following code snippet can be triggered by listeners (e.g. after an input action such as a click), or timed in the update loop.

```

// Calculate detection results
CollisionResults results = new CollisionResults();
a.collideWith(b, results);
System.out.println("Number of Collisions between" +
 a.getName() + " and " + b.getName() + ": " + results.size());
// Use the results
if (results.size() > 0) {
 // how to react when a collision was detected
 CollisionResult closest = results.getClosestCollision();
 System.out.println("What was hit? " + closest.getGeometry().get(
 System.out.println("Where was it hit? " + closest.getContactPoi(
 System.out.println("Distance? " + closest.getDistance());
} else {
 // how to react when no collision occurred
}
}

```

You can also loop over all results and trigger different reactions depending on what was hit and where it was hit. In this example, we simply print info about them.

```

// Calculate Results
CollisionResults results = new CollisionResults();
a.collideWith(b, results);
System.out.println("Number of Collisions between" + a.getName()+
 + b.getName() : " + results.size());
// Use the results
for (int i = 0; i < results.size(); i++) {
 // For each hit, we know distance, impact point, name of geomet
 float dist = results.getCollision(i).getDistance();
 Vector3f pt = results.getCollision(i).getContactPoint();
 String party = results.getCollision(i).getGeometry().getName(
 int tri = results.getCollision(i).getTriangleIndex();
 Vector3f norm = results.getCollision(i).getTriangle(new Triang
 System.out.println("Details of Collision #" + i + ":");

 System.out.println(" Party " + party + " was hit at " + pt + "
 System.out.println(" The hit triangle #" + tri + " has a norma
}

```

Knowing the distance of the collisions is useful for example when you intersect Lines and Rays with other objects.

## Bounding Volumes

A `com.jme3.bounding.BoundingVolume` is an interface for dealing with containment of a collection of points. All BoundingVolumes are `Collidable` and are used as optimization to calculate non-physical collisions more quickly: It's always faster to calculate an intersection between simple shapes like spheres and boxes than between complex shapes like models.

jME3 computes bounding volumes for all objects. These bounding volumes are later used for frustum culling, which is making sure only objects visible on-screen are actually sent for rendering.

All fast-paced action and shooter games use BoundingVolumes as an optimization. Wrap all complex models into simpler shapes – in the end, you get equally useful collision detection results, but faster. [More about bounding volumes...](#)



Supported types:

- Type.AABB = Axis-aligned bounding box, that means it doesn't rotate, which makes it less precise. A `com.jme3.bounding.BoundingBox` is an axis-aligned cuboid used as a container for a group of vertices of a piece of geometry. A BoundingBox has a center and extents from that center along the x, y and z axis. This is the default bounding volume, since it is fairly fast to generate and gives better accuracy than the bounding sphere.
- Type.Sphere: `com.jme3.bounding.BoundingSphere` is a sphere used as a container for a group of vertices of a piece of geometry. A BoundingSphere has a center and a radius.
- Type.OBB = Oriented bounding box. This bounding box is more precise because it can rotate with its content, but is computationally more expensive. (Currently not supported.)
- Type.Capsule = Cylinder with rounded ends, also called “swept sphere”. Typically used for mobile characters. (Currently not supported.)

Note: If you are looking for bounding volumes for physical objects, use [CollisionShapes](#).  
</div>

## Usage

For example you can use Bounding Volumes on custom meshes, or complex non-physical shapes.

```
mesh.setBound(new BoundingSphere());
mesh.updateBound();
```

## Mesh and Scene Graph Collision

One of the supported `Collidable`s are meshes and scene graph objects. To execute a collision detection query against a scene graph, use `Spatial.collideWith()`. This will traverse the scene graph and return any mesh collisions that were detected. Note that the first collision against a particular scene graph may take a long time, this is because a special data structure called [Bounding Interval Hierarchy \(BIH\)](#) needs to be generated for the meshes. At a later point, the mesh could change and the BIH tree would become out of date, in that case, call [Mesh.createCollisionData\(\)](#) on the changed mesh to update the BIH tree.

## Intersection

A `com.jme3.math.Ray` is an infinite line with a beginning, a direction, and no end; whereas a `com.jme3.math.Line` is an infinite line with only a direction (no beginning, no end).

Rays are used to perform line-of-sight calculations. This means you can detect what users were “aiming at” when they clicked or pressed a key. You can also use this to detect whether game characters can see something (or someone) or not.

- **Click to select:** You can determine what a user has clicked by casting a ray from the camera forward in the direction of the camera. Now identify the closest collision of the ray with the `rootNode`, and you have the clicked object.
- **Line of sight:** Cast a ray from a player in the direction of another player. Then you detect all collisions of this ray with other entities (walls versus foliage versus window panes) and use this to calculate how likely it is that one can see the other.

These simple but powerful ray-surface intersection tests are called Ray Casting. As opposed to the more advanced Ray Tracing technique, Ray Casting does not follow the ray's reflection after the first hit – the ray just goes straight on.

Learn the details of how to implement [Mouse Picking](#) here.

---

TODO:

- Bounding Interval Hierarchy ( `com.jme3.collision.bih.BIHNode` )
- `com.jme3.scene.CollisionData`

</div>

## title: Combo Moves

# Combo Moves

The ComboMoves class allows you to define combinations of inputs that trigger special actions. Entering an input combo correctly can bring the player incremental rewards, such as an increased chance to hit, an increased effectiveness, or decreased chance of being blocked, whatever the game designer chooses. [More background info](#)

Combos are usually a series of inputs, in a fixed order: For example a keyboard combo can look like: “press Down, then Down+Right together, then Right”.

Usage:

1. Create input triggers
2. Define combos
3. Detect combos in ActionListener
4. Execute combos in update loop

Copy the two classes ComboMoveExecution.java and ComboMove.java into your application and adjust them to your package paths.

## Example Code

- [TestComboMoves.java](#)
- [ComboMoveExecution.java](#) ← required
- [ComboMove.java](#) ← required

## Create Input Triggers

First you [define your game's inputs](#) as you usually do: Implement the com.jme3.input.controls.ActionListener interface for your class, and add triggers mappings such as com.jme3.input.controls.KeyTrigger and com.jme3.input.KeyInput.

For example:

```

inputManager.addMapping("Left", new KeyTrigger(KeyInput.KEY_LEFT)
inputManager.addMapping("Right", new KeyTrigger(KeyInput.KEY_RIGHT)
inputManager.addMapping("Up", new KeyTrigger(KeyInput.KEY_UP))
inputManager.addMapping("Down", new KeyTrigger(KeyInput.KEY_DOWN))
inputManager.addMapping("Attack1", new KeyTrigger(KeyInput.KEY_SPACE))
...
inputManager.addListener(this, "Left", "Right", "Up", "Down", "Attack1")

```

## Define Combos

For each of your combo moves, you specify the series of inputs that will trigger it. The order in which you define them is the order the player has to press them for the step to be recorded. When all steps have been recorded, the combo is triggered.

The following example shows how a fireball combo move is triggered by pressing the navigation keys for “down, down+right, right”, in this order.

```

ComboMove fireball = new ComboMove("Fireball");
fireball.press("Down").notPress("Right").done();
fireball.press("Right", "Down").done();
fireball.press("Right").notPress("Down").done();
fireball.notPress("Right", "Down").done();
fireball.setUseFinalState(false);

```

Also create a `ComboMoveExecution` object for each `ComboMove`. You need it later to execute the detected combo.

```
ComboMoveExecution fireballExec = new ComboMoveExecution(fireball);
```

## ComboMove Class Methods

Use the following `ComboMove` methods to specify the combo:

ComboMove Method	Description
press("A").done(); press("A", "B").done();	Combo step is recorded if A is entered. Combo step is recorded if A and B are entered simultaneously.
notPress("A").done(); notPress("A", "B").done();	Combo step is recorded if A is released. Combo step is recorded if A and B are both released.
press("A").notPress("B").done();	Combo step is recorded if A is entered, and not B
press("A").notPress("B").timeElapsed(0.11f).done();	Combo step is recorded a certain time after A and not B is entered. etc, etc ...
setPriority(0.5f);	If there is an ambiguity, a high-priority combo will trigger instead of a low-priority combo. This prevents that a similar looking combo step "hijacks" another Combo. Use only once per ComboMove.
setUseFinalState(false); setUseFinalState(true);	This is the final command of the series. False: Do not wait on a final state, chain combo steps. (?) True: This is the final state, do not chain combo steps. (?)

The `press()` and `notPress()` methods accept sets of Input Triggers, e.g.

```
fireball.press("A", "B", "C").done() .
```

The following getters give you more information about the game state:

ComboMove Method	Usage
getCastTime()	Returns the time since the last step has been recorded. (?)
getMoveName()	Returns the string of the current combo
getPriority()	Returns the priority of this move

## Detect Combos in ActionListener

Now that you have specified the combo steps, you want to detect them. You do that in the `onAction()` method that you get from the `ActionListener` interface.

Create a HashSet `pressMappings` to track currently pressed mappings, and a ComboMove object `currentMove` to track the current move.

We also track the cast time of a combo to determine if it has timed out (see update loop below).

```

private HashSet<String> pressedMappings = new HashSet<String>();
private ComboMove currentMove = null;
private float currentMoveCastTime = 0;
private float time = 0;
...

public void onAction(String name, boolean isPressed, float tpf) {
 // Record pressed mappings
 if (isPressed){
 pressedMappings.add(name);
 }else{
 pressedMappings.remove(name);
 }

 // The pressed mappings have changed: Update ComboExecution obj
 List<ComboMove> invokedMoves = new ArrayList<ComboMove>();
 if (fireballExec.updateState(pressedMappings, time)){
 invokedMoves.add(fireball);
 }
 // ... add more ComboExecs here...

 // If any ComboMoves have been successfully triggered:
 if (invokedMoves.size() > 0){
 // identify the move with highest priority
 float priority = 0;
 ComboMove toExec = null;
 for (ComboMove move : invokedMoves){
 if (move.getPriority() > priority){
 priority = move.getPriority();
 toExec = move;
 }
 }
 if (currentMove != null && currentMove.getPriority() > toEx
 return; // skip lower-priority moves
 }
}

```

```

 // If a ComboMove has been identified, store it in currentMove
 currentMove = toExec;
 currentMoveCastTime = currentMove.getCastTime();
 }
}

```

## Execute Combos in the Update Loop

Now that you have detected the current move, you want to execute it. You do that in the update loop.

```

@Override
public void simpleUpdate(float tpf){
 time += tpf;
 fireballExec.updateExpiration(time);
 // ... update more ComboExecs here.....

 if (currentMove != null){
 currentMoveCastTime -= tpf;
 if (currentMoveCastTime <= 0){
 System.out.println("THIS COMBO WAS TRIGGERED: " + currentMove.getMoveName());
 // TODO: for each combo, implement special actions here
 currentMoveCastTime = 0;
 currentMove = null;
 }
 }
}

```

Test `currentMove.getMoveName()` and proceed to call methods that implement any special actions and bonuses. This is up to you and depends individually on your game.

## Why Combos?

Depending on the game genre, the designer can reward the players' intrinsical or extrinsical skills:

- (intrinsical:) RPGs typically calculate the success of an attack from the character's in-game training level: The player plays the role of a character whose skill level is defined in numbers. RPGs typically do not offer any Combos.
- (extrinsical:) Sport and fighter games typically choose to reward the player's "manual" skills: The success of a special move solely depends on the player's own dexterity. These games typically offer optional Combos.

[keyinput](#), [input](#), [documentation](#)

</div>

## title: Custom Controls

# Custom Controls

A `com.jme3.scene.control.Control` is a customizable jME3 interface that allows you to cleanly steer the behaviour of game entities (Spatial), such as artificially intelligent behaviour in NPCs, traps, automatic alarms and doors, animals and pets, self-steering vehicles or platforms – anything that moves and interacts. Several instances of custom Controls together implement the behaviours of a type of Spatial.

To control global game behaviour see [Application States](#) – you often use AppStates and Control together.

- [Quick video introduction to Custom Controls](#)

To control the behaviour of spatial:

1. Create one control for each *type of behavior*. When you add several controls to one spatial, they will be executed in the order they were added.  
For example, one NPC can be controlled by a PhysicsControl instance and an AIControl instance.
2. Define the custom control and implement its behaviour in the Control's update method:
  - You can pass arguments into your custom control.
  - In the control class, the object `spatial` gives you access to the spatial and subspatial that the control is attached to.
  - Here you modify the `spatial`'s transformation (move, scale, rotate), play animations, check its environment, define how it acts and reacts.
3. Add an instance of the Control to a spatial to give it this behavior. The spatial's game state is updated automatically from now on.

```
spatial.addControl(myControl);
```

To implement game logic for a type of spatial, you will either extend AbstractControl (most common case), or implement the Control interface, as explained in this article.

## Usage

Use [Controls](#) to implement the *behaviour of types of game entities*.

- Use Controls to add a type of behaviour (that is, methods and fields) to individual Spatial entities.
- Each Control has its own `update()` loop that hooks into `simpleUpdate()`. Use Controls to move blocks of code out of the `simpleUpdate()` loop.
- One Spatial can be influenced by several Controls. (Very powerful and modular!)
- Each Spatial needs its own instance of the Control.
- A Control only has access to and control over the Spatial it is attached to.
- Controls can be saved as .j3o file together with a Spatial.

Examples: You can write

- A WalkerNavController, SwimmerNavController, FlyerNavController... that defines how a type of NPC finds their way around. All NPCs can walk, some can fly, others can swim, and some can all three, etc.
- A PlayerNavController that is steered by user-configurable keyboard and mouse input.
- A generic animation control that acts as a common interface that triggers animations (walk, stand, attack, defend) for various entities.
- A DefensiveBehaviourControl that remote-controls NPC behaviour in fight situations.
- An IdleBehaviourControl that remote-controls NPC behaviour in neutral situations.
- A DestructionControl that automatically replaces a structure with an appropriate piece of debris after collision with a projectile...

The possibilities are endless. 

## Example Code

Other examples include the built-in RigidBodyControl in JME's physics integration, the built-in TerrainLODControl that updates the terrain's level of detail depending on the viewer's perspective, etc.

Existing examples in the code base include:

- [AnimControl.java](#) allows manipulation of skeletal animation, including blending and multiple channels.
- [CameraControl.java](#) allows you to sync the camera position with the position of a given spatial.
- [BillboardControl.java](#) displays a flat picture orthogonally, e.g. a speech bubble or informational dialog.
- [PhysicsControl](#) subclasses (such as CharacterControl, RigidBodyControl, VehicleControl) allow you to add physical properties to any spatial. PhysicsControls tie

into capabilities provided by the BulletAppState.

## AbstractControl Class

The most common way to create a Control is to create a class that extends AbstractControl.

The AbstractControl can be found under `com.jme3.scene.control.AbstractControl`. This is a default abstract class that implements the Control interface.

- You have access to a boolean `isEnabled()`.
- You have access to the Spatial object `spatial`.
- You override the `controlUpdate()` method to implement the Spatial's behaviour.
- You have access to a `setEnabled(boolean)` method. This activates or deactivates this Control's behaviour in this spatial temporarily. While the AbstractControl is toggled to be disabled, the `controlUpdate()` loop is no longer executed.

For example, you disable your IdleBehaviourControl when you enable your DefensiveBehaviourControl in a spatial.

Usage: Your custom subclass implements the three methods `controlUpdate()`, `controlRender()`, `setSpatial()`, and `cloneForSpatial()` as shown here:

```
public class MyControl extends AbstractControl implements Savable,
 private int index; // can have custom fields -- example

 public MyControl(){} // empty serialization constructor

 /** Optional custom constructor with arguments that can init cust
 * Note: you cannot modify the spatial here yet! */
 public MyControl(int i){
 // index=i; // example
 }

 /** This method is called when the control is added to the spatial
 * and when the control is removed from the spatial (setting a new
 * It can be used for both initialization and cleanup. */
 @Override
 public void setSpatial(Spatial spatial) {
 super.setSpatial(spatial);
 /* Example:
 if (spatial != null){
 // initialize
 }
 }
}
```

```
 }else{
 // cleanup
 }
 */
}

/** Implement your spatial's behaviour here.
 * From here you can modify the scene graph and the spatial
 * (transform them, get and set userdata, etc).
 * This loop controls the spatial while the Control is enabled.
@Override
protected void controlUpdate(float tpf){
 if(spatial != null) {
 // spatial.rotate(tpf,tpf,tpf); // example behaviour
 }
}

@Override
public Control cloneForSpatial(Spatial spatial){
 final MyControl control = new MyControl();
 /* Optional: use setters to copy userdata into the cloned control
 * control.setIndex(i); // example
 control.setSpatial(spatial);
 return control;
}

@Override
protected void controlRender(RenderManager rm, ViewPort vp){
 /* Optional: rendering manipulation (for advanced users) */
}

@Override
public void read(JmeImporter im) throws IOException {
 super.read(im);
 // im.getCapsule(this).read(...);
}

@Override
public void write(JmeExporter ex) throws IOException {
```

```

 super.write(ex);
 // ex.getCapsule(this).write(...);
 }

}

```

See also:

- To learn more about `write()` and `read()`, see [Save and Load](#)
- To learn more about `setUserData()`, see [Spatial](#).

</div>

## The Control Interface

In the less common case that you want to create a Control that also extends another class, create a custom interface that extends jME3's Control interface. Your class can become a Control by implementing the Control interface, and at the same time extending another class.

The Control interface can be found under `com.jme3.scene.control.Control`. It has the following method signatures:

- `cloneForSpatial(Spatial)` : Clones the Control and attaches it to a clone of the given Spatial.  
Implement this method to be able to [save\(\)](#) and [load\(\)](#) Spatials carrying this Control. The AssetManager also uses this method if the same spatial is loaded twice. You can specify which fields you want your object to reuse (e.g. collisionshapes) in this case.
- `setEnabled(boolean)` : Toggles a boolean that enables or disables the Control. Goes with accessor `isEnabled()`. You test for it in the `update(float tpf)` loop before you execute anything.
- There are also some internal methods that you do not call from user code:  
`setSpatial(Spatial s)` , `update(float tpf)` ; , `render(RenderManager rm, ViewPort vp)` .

Usage example:

1. Create a custom control interface </p>

```

public interface MyControlInterface extends Control {
 public void setSomething(int x); // optionally, add custom metl
}

```

1. Create custom Controls implementing your Control interface.

```
public class MyControl extends MyCustomClass implements MyControl {
 protected Spatial spatial;
 protected boolean enabled = true;

 public MyControl() { } // empty serialization constructor

 public MyControl(int x) { // custom constructor
 super(x);
 }

 @Override
 public void update(float tpf) {
 if (enabled && spatial != null) {
 // Write custom code to control the spatial here!
 }
 }

 @Override
 public void render(RenderManager rm, ViewPort vp) {
 // optional for advanced users, e.g. to display a debug shape
 }

 @Override
 public Control cloneForSpatial(Spatial spatial) {
 MyControl control = new MyControl();
 // set custom properties
 control.setSpatial(spatial);
 control.setEnabled(isEnabled());
 // set some more properties here...
 return control;
 }

 @Override
 public void setEnabled(boolean enabled) {
 this.enabled = enabled;
 }
}
```

```

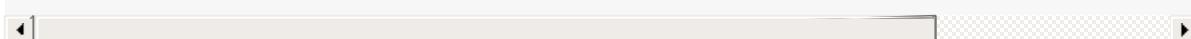
@Override
public boolean isEnabled() {
 return enabled;
}

@Override
public void setSomething(int z) {
 // You can add custom methods ...
}

@Override
public void write(JmeExporter ex) throws IOException {
 super.write(ex);
 OutputCapsule oc = ex.getCapsule(this);
 oc.write(enabled, "enabled", true);
 oc.write(spatial, "spatial", null);
 // write custom variables
}

@Override
public void read(JmeImporter im) throws IOException {
 super.read(im);
 InputCapsule ic = im.getCapsule(this);
 enabled = ic.readBoolean("enabled", true);
 spatial = (Spatial) ic.readSavable("spatial", null);
 // read custom variables
}
}

```



&lt;/div&gt;

## Best Practices

**Tip:** Use the `getControl()` accessor to get Control objects from Spatial objects. No need to pass around lots of object references. Here an example from the [MonkeyZone](#) code:

```

public class CharacterAnimControl implements Control {
 ...
 public void setSpatial(Spatial spatial) {
 ...
 animControl = spatial.getControl(AnimControl.class);
 characterControl = spatial.getControl(CharacterControl.class);
 ...
 }
}

```

**Tip:** You can create custom Control interfaces so a set of different Controls provide the same methods and can be accessed with the interface class type.

```

public interface ManualControl extends Control {
 public void steerX(float value);
 public void steerY(float value);
 public void moveX(float value);
 public void moveY(float value);
 public void moveZ(float value);
 ...
}

```

Then you create custom sub-Controls and implement the methods accordingly to the context:

```
public class ManualVehicleControl extends ManualControl {...}
```

and

```
public class ManualCharacterControl extends ManualControl {...}
```

Then add the appropriate controls to spatsials:

```

characterSpatial.addControl(new ManualCharacterControl());
...
vehicleSpatial.addControl(new ManualVehicleControl());
...

```

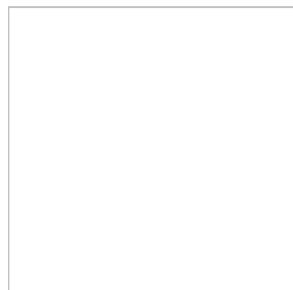
**Tip:** Use the `getControl()` method on a `Spatial` to get a specific `Control` object, and activate its behaviour!

```
ManualControl c = mySpatial.getControl(ManualControl.class);
c.steerX(steerX);
```

```
</div>
```

## title: Custom Mesh Shapes

# Custom Mesh Shapes



Use the `Mesh` class to create custom shapes that go beyond Quad, Box, Cylinder, and Sphere, even procedural shapes are possible. Thank you to KayTrance for providing the sample code!

**Note:** In this tutorial, we (re)create a very simple rectangular mesh (a quad), and we have a look at different ways of coloring it. Coding a custom quad may not be very useful because it's exactly the same as the built-in `com.jme3.scene.shape.Quad`. We chose a simple quad to teach you how to build any shape out of triangles, without the distractions of more complex shapes.

- Full code sample: [TestCustomMesh.java](#)

## Polygon Meshes

Polygon [meshes](#) are made up of triangles. The corners of the triangles are called vertices. When ever you create any new shape, you break it down into triangles.

**Example:** Let's look at a cube. A cube is made up of 6 rectangles. Each rectangle can be broken down into two triangles. This means you need 12 triangles to describe a cube mesh. Therefor you must provide the coordinates of the triangles' 8 corners (called vertices).

The important thing is that you have to specify the vertices of each triangle in the right order: Each triangle separately, counter-clockwise.

Sounds harder than it is – let's create a simple custom mesh, a quad.

## Creating a Quad Mesh

In this tutorial we want to create a  $3 \times 3$  Quad. The quad has four vertices, and is made up of two triangles. In our example, we decide that the bottom left corner is at  $0/0/0$  and the top right is at  $3/3/0$ .

```
0, 3, 0 -- 3, 3, 0
| \
| \
| \
| \
| \
0, 0, 0 -- 3, 0, 0
```

## The Mesh Object

The base class for creating meshes is `com.jme3.scene.Mesh`.

```
Mesh mesh = new Mesh();
```

Tip: If you create your own Mesh-based class (`public class MyMesh extends Mesh { }`), replace the variable `mesh` by `this` in the following examples.

## Vertex Coordinates

To define your own shape, determine the shape's **vertex coordinates** in 3D space. Store the list of corner positions in an `com.jme3.math.Vector3f` array. For a Quad, we need four vertices: Bottom left, bottom right, top left, top right. We name the array `vertices[]`.

```
Vector3f [] vertices = new Vector3f[4];
vertices[0] = new Vector3f(0,0,0);
vertices[1] = new Vector3f(3,0,0);
vertices[2] = new Vector3f(0,3,0);
vertices[3] = new Vector3f(3,3,0);
```

## Texture Coordinates

Next, we define the Quad's 2D **texture coordinates** for each vertex, in the same order as the vertices: Bottom left, bottom right, top left, top right. We name this `Vector2f` array `texCoord[]`

```
Vector2f[] texCoord = new Vector2f[4];
texCoord[0] = new Vector2f(0,0);
texCoord[1] = new Vector2f(1,0);
texCoord[2] = new Vector2f(0,1);
texCoord[3] = new Vector2f(1,1);
```

This syntax means, when you apply a texture to this mesh, the texture will fill the quad from corner to corner at 100% percent size. Especially when you stitch together a larger mesh, you use this to tell the renderer whether, and how exactly, you want to cover the whole mesh. E.g. if you use .5f or 2f as texture coordinates instead of 1f, textures will be stretched or shrunk accordingly.

## Connecting the Dots

Next we turn these unrelated coordinates into **triangles**: We define the order in which each triangle is constructed. Think of these indexes as coming in groups of three. Each group of indexes describes one triangle. If the corners are identical, you can (and should!) reuse an index for several triangles.

Remember that you must specify the vertices counter-clockwise.

```
int [] indexes = { 2,0,1, 1,3,2 };
```

This syntax means:

- The indices 0,1,2,3 stand for the four vertices that you specified for the quad in `vertices[]`.
- The 2,0,1 triangle starts at top left, continues bottom left, and ends at bottom right.
- The 1,3,2 triangle start at bottom right, continues top right, and ends at top left.

```
2\2--3
| \ | Counter-clockwise
| \ |
0--1\1
```

If the shape is more complex, it has more triangles, and therefor also more vertices/indices. Just continue expanding the list by adding groups of three indices for each triangle. (For example a three-triangle “house” shape has 5 vertices/indices and you'd specify three groups: `int [] indexes = { 2,0,1, 1,3,2, 2,3,4 };` .)

If you get the order wrong (clockwise) for some of the triangles, then these triangles face backwards. If the [Spatial](#)'s material uses the default `FaceCullMode.Back` (see “face culling”), the broken triangles appear as holes in the rendered mesh. You need to identify and fix them in your code.

</div>

## Setting the Mesh Buffer

You store the Mesh data in a buffer.

1. Using `com.jme3.util.BufferUtils`, we create three buffers for the three types of information we have:
  - vertex coordinates,
  - texture coordinates,
  - indices.
2. We assign the data to the appropriate type of buffer inside the `Mesh` object. The three buffer types (`Position`, `TexCoord`, `Index`) are taken from an enum in `com.jme3.scene.VertexBuffer.Type`.
3. The integer parameter describes the number of components of the values. Vertex positions are 3 float values, texture coordinates are 2 float values, and the indices are 3 ints representing 3 vertices in a triangle.
4. To render the mesh in the scene, we need to pre-calculate the bounding volume of our new mesh: Call the `updateBound()` method on it.

```
mesh.setBuffer(Type.Position, 3, BufferUtils.createFloatBuffer(vert
mesh.setBuffer(Type.TexCoord, 2, BufferUtils.createFloatBuffer(texC
mesh.setBuffer(Type.Index, 3, BufferUtils.createIntBuffer(indexe
mesh.updateBound();
```

Our Mesh is ready! Now we want to see it.

## Using the Mesh in a Scene

We create a `com.jme3.scene.Geometry` and `com.jme3.material.Material` from our `mesh`, apply a simple color material to it, and attach it to the `rootNode` to make it appear in the scene.

```
Geometry geo = new Geometry("OurMesh", mesh); // using our custom mesh
Material mat = new Material(assetManager,
 "Common/MatDefs/Misc/Unshaded.j3md");
mat.setColor("Color", ColorRGBA.Blue);
geo.setMaterial(mat);
rootNode.attachChild(geo);
```

Library for assetManager? Ta-daa!

## Using a Quad instead

We created a quad Mesh it can be replace by a Quad such as :

```
Quad quad = new Quad(1,1); // replace the definition of Vertex and
Geometry geo = new Geometry("OurQuad", quad); // using Quad object
Material mat = new Material(assetManager,
 "Common/MatDefs/Misc/Unshaded.j3md");
mat.setColor("Color", ColorRGBA.Blue);
geo.setMaterial(mat);
rootNode.attachChild(geo);
```

If you want to change the Textures Coordinates, in order to change the scale of the texture, use :

```
Quad quad = new Quad(1,1);
quad.scaleTextureCoordinates(new Vector2f(width , height));
```

## Dynamic Meshes

If you are modifying a mesh dynamically in a way which changes the model's bounds, you need to update it:

1. Call `updateBound()` on the mesh object, and then
2. call `updateModelBound()` on the Geometry object containing the mesh.

The `updateModelBound()` method warns you about not usually needing to use it, but that can be ignored in this special case.

*N.B.: This does not work on TerrainQuad. Please use the TerrainQuad.adjustHeight() function to edit the TerrainQuad mesh instead. Additionally, if you want to use collisions on them afterwards, you need to call TerrainPatch.getMesh().createCollisionData(); to update the collision data, else it will collide with what seems to be the old mesh.*

## Optional Mesh Features

There are more vertex buffers in a Mesh than the three shown above. For an overview, see also [mesh](#).

### Example: Vertex Colors

Vertex coloring is a simple way of coloring meshes. Instead of just assigning one solid color, each vertex (corner) has a color assigned. The faces between the vertices are then colored with a gradient. For this demo, you can use the same mesh `mesh` object that you defined above.

```
Geometry geo = new Geometry ("ColoredMesh", mesh); // using the cus
Material matVC = new Material(assetManager, "Common/MatDefs/Misc/Ur
matVC.setBoolean("VertexColor", true);
```

You create a float array color buffer:

- Assign 4 color values, RGBA, to each vertex.
  - To loop over the 4 color values, use a color index

```
int colorIndex = 0;
```

- The color buffer contains four color values for each vertex.
  - The Quad in this example has 4 vertices.

```
float[] colorArray = new float[4*4];
```

- Tip: If your mesh has a different number of vertices, you would write:

```
float[] colorArray = new float[yourVertexCount * 4]
```

Loop over the colorArray buffer to quickly set some RGBA value for each vertex. As usual, RGBA color values range from 0.0f to 1.0f. **Note that the color values in this example are arbitrarily chosen.** It's just a quick loop to give every vertex a different RGBA value (a purplish gray, purple, a greenish gray, green, see screenshot), without writing too much code. For your own mesh, you'd assign meaningful values for the color buffer depending on which color you want your mesh to have.

```
// note: the red and green values are arbitrary in this example
for(int i = 0; i < 4; i++){
 // Red value (is increased by .2 on each next vertex here)
 colorArray[colorIndex++]= 0.1f+(.2f*i);
 // Green value (is reduced by .2 on each next vertex)
 colorArray[colorIndex++]= 0.9f-(0.2f*i);
 // Blue value (remains the same in our case)
 colorArray[colorIndex++]= 0.5f;
 // Alpha value (no transparency set here)
 colorArray[colorIndex++]= 1.0f;
}
```

Next, set the color buffer. An RGBA color value contains four float components, thus the parameter `4`.

```
mesh.setBuffer(Type.Color, 4, colorArray);
geo.setMaterial(matVC);
```

When you run this code, you see a gradient color extending from each vertex.

## Example: Using Meshes With Lighting.j3md

The previous examples used the mesh together with the `Unshaded.j3md` material. If you want to use the mesh with a Phong illuminated material (such as `Lighting.j3md`), the mesh must include information about its Normals. (Normal Vectors encode in which direction a mesh polygon is facing, which is important for calculating light and shadow!)

```
float[] normals = new float[12];
normals = new float[]{0,0,1, 0,0,1, 0,0,1, 0,0,1};
mesh.setBuffer(Type.Normal, 3, BufferUtils.createFloatBuffer(normals))
```

You need to specify as many normals as the polygon has vertices. For a flat quad, the four normals point in the same direction. In this case, the direction is the Z unit vector (0,0,1), this means our quad is facing the camera.

If the mesh is more complex or rounded, calculate cross products of neighbouring vertices to identify normal vectors!

## Example: Point Mode

Additionally to coloring the faces as just described, you can hide the faces and show only the vertices as colored corner points.

```
Geometry coloredMesh = new Geometry ("ColoredMesh", cMesh);
...
mesh.setMode(Mesh.Mode.Points);
mesh.setPointSize(10f);
mesh.updateBound();
mesh.setStatic();
Geometry points = new Geometry("Points", mesh);
points.setMaterial(mat);
rootNode.attachChild(points);
rootNode.attachChild(geo);
```

This will result in a 10 px dot being rendered for each of the four vertices. The dot has the vertex color you specified above. The Quad's faces are not rendered at all in this mode. You can use this to visualize a special debugging or editing mode in your game.

## Debugging Tip: Culling

By default, jME3 optimizes a mesh by “backface culling”, this means not drawing the inside. It determines the side of a triangle by the order of the vertices: The frontface is the face where the vertices are specified counter-clockwise.

This means for you that, by default, your custom mesh is invisible when seen from “behind” or from the inside. This may not be a problem, typically this is even intended, because it's faster. The player will not look at the inside of most things anyway. For example, if your custom mesh is a closed polyhedron, or a flat wallpaper-like object, then rendering the backfaces (the inside of the pillar, the back of the painting, etc) would indeed be a waste of resources.

In case however that your usecase requires the backfaces be visible, you have two options:

- If you have a very simple scene, you can simply deactivate backface culling for this one mesh's material.

```
mat.getAdditionalRenderState().setFaceCullMode(FaceCullMode.Off)
```



- Another solution for truly double-sided meshes is to specify each triangle twice, the second time with the opposite order of vertices. The second (reversed) triangle is a second frontface that covers up the culled backface.

```
int[] indexes = { 2,0,1, 1,3,2, 2,3,1, 1,0,2 };
```

---

#### See also:

- [Spatial](#) – contains more info about how to debug custom meshes (that do not render as expected) by changing the default culling behaviour.
- [Mesh](#) – more details about advanced Mesh properties

[spatial](#), [node](#), [mesh](#), [geometry](#), [scenegraph](#)

</div>

## title: Debugging

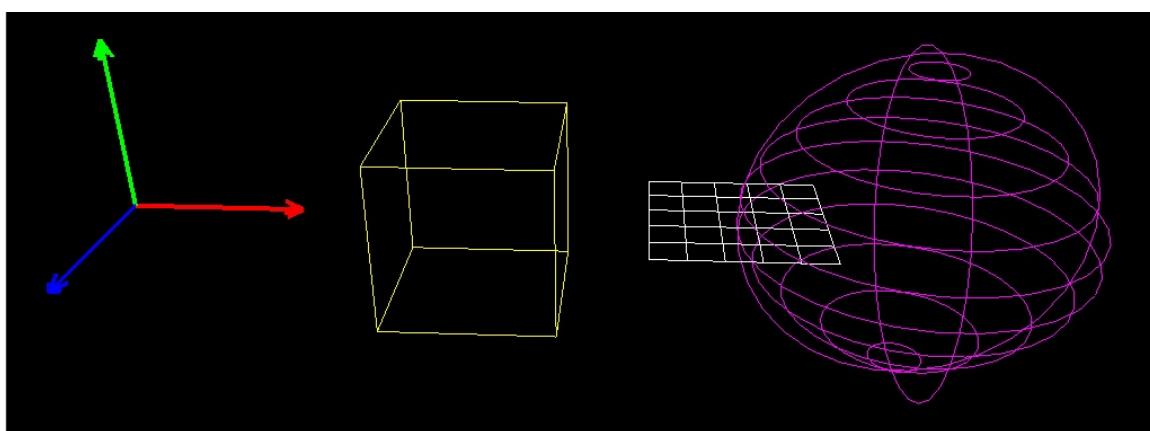
# Debugging

When you deal with complex game engine features like animations or physics it is handy to get feedback from the engine how it interpreted the current state. Is the physical object's collision shape really where you think it is? Is the skeleton of the animated character moving like you think it should? This document shows you how to activate visual debug aides.

What if you just want to quickly write code that loads models and brings them in their start position? You may not want to hunt for a sample model, convert it, add lights, and load materials. Instead you use "hasslefree" simple shapes, and a "hasslefree" unshaded material or wireframe: No model, no light source, no materials are needed to see them in your test scene.

If you ever have problems with objects appearing in the wrong spot, with the wrong scale, or wrong orientation, simply attach debug shapes to your scene to have a point of reference in 3D space – just like a giant ruler. If your code positions the debug shapes correctly, but models remain invisible when you apply the same code to them, you know that the problem must be either the model (where is its origin coordinate?), or the light (too dark? too bright? missing?), or the model's material (missing?) – and not the positioning code.

Here are some different debug shapes:



</div>

## Debug Shapes

```
</div>
```

## Coordinate Axes

The coordinate axes (com.jme3.scene.debug.Arrow) help you see the cardinal directions (X,Y,Z) from their center point. Scale the arrows to use them as a “ruler” for a certain length.

```
private void attachCoordinateAxes(Vector3f pos){
 Arrow arrow = new Arrow(Vector3f.UNIT_X);
 arrow.setLineWidth(4); // make arrow thicker
 putShape(arrow, ColorRGBA.Red).setLocalTranslation(pos);

 arrow = new Arrow(Vector3f.UNIT_Y);
 arrow.setLineWidth(4); // make arrow thicker
 putShape(arrow, ColorRGBA.Green).setLocalTranslation(pos);

 arrow = new Arrow(Vector3f.UNIT_Z);
 arrow.setLineWidth(4); // make arrow thicker
 putShape(arrow, ColorRGBA.Blue).setLocalTranslation(pos);
}

private Geometry putShape(Mesh shape, ColorRGBA color){
 Geometry g = new Geometry("coordinate axis", shape);
 Material mat = new Material(assetManager, "Common/MatDefs/Misc/Ur
 mat.getAdditionalRenderState().setWireframe(true);
 mat.setColor("Color", color);
 g.setMaterial(mat);
 rootNode.attachChild(g);
 return g;
}
```



## Wireframe Grid

Use a wireframe grid (com.jme3.scene.debug.Grid) as a ruler or simple floor.

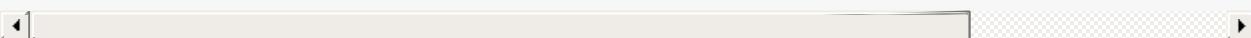
```
private Geometry attachGrid(Vector3f pos, float size, ColorRGBA col
 Geometry g = new Geometry("wireframe grid", new Grid(size, size,
 Material mat = new Material(assetManager, "Common/MatDefs/Misc/Ur
 mat.getAdditionalRenderState().setWireframe(true);
 mat.setColor("Color", color);
 g.setMaterial(mat);
 g.center().move(pos);
 rootNode.attachChild(g);
 return g;
}
```



## Wireframe Cube

Use a wireframe cube (`com.jme3.scene.debug.WireBox`) as a stand-in object to see whether your code scales, positions, or orients, loaded models right.

```
public Geometry attachWireBox(Vector3f pos, float size, ColorRGBA c
 Geometry g = new Geometry("wireframe cube", new WireBox(size, siz
 Material mat = new Material(assetManager, "Common/MatDefs/Misc/Ur
 mat.getAdditionalRenderState().setWireframe(true);
 mat.setColor("Color", color);
 g.setMaterial(mat);
 g.setLocalTranslation(pos);
 rootNode.attachChild(g);
 return g;
}
```



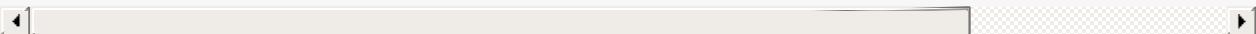
## Wireframe Sphere

Use a wireframe sphere (`com.jme3.scene.debug.WireSphere`) as a stand-in object to see whether your code scales, positions, or orients, loaded models right.

```

private Geometry attachWireSphere(Vector3f pos, float size, ColorRGBA color) {
 Geometry g = new Geometry("wireframe sphere", new WireSphere(size));
 Material mat = new Material(assetManager, "Common/MatDefs/Misc/Unshaded.jmat");
 mat.getAdditionalRenderState().setWireframe(true);
 mat.setColor("Color", color);
 g.setMaterial(mat);
 g.setLocalTranslation(pos);
 rootNode.attachChild(g);
 return g;
}

```



## Wireframe for Physics

You can display a wireframe of the (usually invisible) collision shape around all physical objects. Use this for debugging when analyzing unexpected behaviour. Does not work with DETACHED physics, please switch to PARALLEL or SEQUENTIAL for debugging.

```
physicsSpace.enableDebug(assetManager);
```

With debugging enabled, colors are used to indicate various types of physical objects:

- A magenta wire mesh indicates an active rigid body.
- A blue wire mesh indicates a rigid body which is either new or inactive.
- A yellow wire mesh indicates a ghost.
- Two green arrows indicate a joint.
- A pink wire mesh indicates a character.

## Wireframe for Animations

Making the skeleton visible inside animated models can be handy for debugging animations. The `control` object is an `AnimControl`, `player` is the loaded model.

```

 SkeletonDebugger skeletonDebug =
 new SkeletonDebugger("skeleton", control.getSkeleton());
Material mat = new Material(assetManager, "Common/MatDefs/Misc/
mat.setColor("Color", ColorRGBA.Green);
mat.getAdditionalRenderState().setDepthTest(false);
skeletonDebug.setMaterial(mat);
player.attachChild(skeletonDebug);

```



## Example: Toggle Wireframe on Model

We assume that you have loaded a model with a material `mat`.

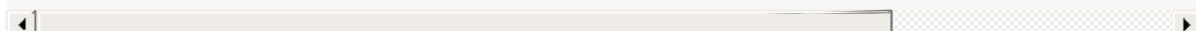
Then you can add a switch to toggle the model's wireframe on and off, like this:

1. Create a key input trigger that switches between the two materials: E.g. we toggle when the T key is pressed:

```

inputManager.addMapping("toggle wireframe", new KeyTrigger(I
inputManager.addListener(actionListener, "toggle wireframe"

```

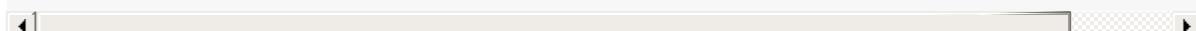


2. Now add the toggle action to the action listener

```

private ActionListener actionListener = new ActionListener() {
 @Override
 public void onAction(String name, boolean pressed, float tps) {
 // toggle wireframe
 if (name.equals("toggle wireframe") && !pressed) {
 wireframe = !wireframe; // toggle boolean
 mat.getAdditionalRenderState().setWireframe(wireframe);
 }
 // else ... other input tests.
 }
};

```



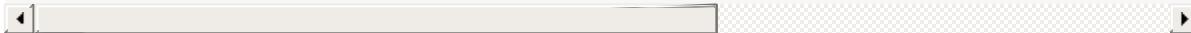
3. Alternatively you could traverse over the whole scene and toggle for all Geometry objects in there if you don't want to create a new SceneProcessor

```

private ActionListener actionListener = new ActionListener() {
 boolean wireframe = false;

 @Override
 public void onAction(String name, boolean pressed, float tps) {
 // toggle wireframe
 if (name.equals("toggle wireframe") && !pressed) {
 wireframe = !wireframe; // toggle boolean
 rootNode.depthFirstTraversal(new SceneGraphVisitor() {
 public void visit(Spatial spatial) {
 if (spatial instanceof Geometry)
 ((Geometry)spatial).getMaterial().getAdditionalRenderState()
 .setWireframe(wireframe);
 }
 });
 }
 // else ... other input tests.
 }
};

```



TIP :: To set the line width of wireframe display, use mesh.setLineWidth(lineWidth). Default line width is 1.

</div>

## Example: Toggle Wireframe on the scene

To display the wireframe of the entire scene instead on one material at a time, first create the following Scene Processor

```

public class WireProcessor implements SceneProcessor {

 RenderManager renderManager;
 Material wireMaterial;

 public WireProcessor(AssetManager assetManager) {
 wireMaterial = new Material(assetManager, "/Common/MatDefs/Wireframe/Wireframe.j3m");
 wireMaterial.setColor("Color", ColorRGBA.Blue);
 wireMaterial.getAdditionalRenderState().setWireframe(true);
 }
}

```

```
}

public void initialize(RenderManager rm, ViewPort vp) {
 renderManager = rm;
}

public void reshape(ViewPort vp, int w, int h) {
 throw new UnsupportedOperationException("Not supported yet.");
}

public boolean isInitialized() {
 return renderManager != null;
}

public void preFrame(float tpf) {

}

public void postQueue(RenderQueue rq) {
 renderManager.setForcedMaterial(wireMaterial);
}

public void postFrame(FrameBuffer out) {
 renderManager.setForcedMaterial(null);
}

public void cleanup() {
 renderManager.setForcedMaterial(null);
}

}
```

Then attach the scene processor to the [GUI](#) Viewport.

```
getViewPort().addProcessor(new WireProcessor());
```

```
</div>
```

## See also

- [Spatial](#) – if you can't see certain spatlals, you can modify the culling behaviour to identify problems (such as inside-out custom meshes)

</div>

# title: jME3 Special Effects Overview

## jME3 Special Effects Overview

jME3 supports several types of special effects: Post-Processor Filters, SceneProcessors, and Particle Emitters (also known as particle systems). This list contains screenshots and links to sample code that demonstrates how to add the effect to a scene.

## Sample Code

- There is one `com.jme3.effect.ParticleEmitter` class for all Particle Systems.
- There is one `com.jme3.post.FilterPostProcessor` class and several `com.jme3.post.filters.*` classes (all Filters have `*Filter` in their names).
- There are several `SceneProcessor` classes in various packages, including e.g. `com.jme3.shadow.*` and `com.jme3.water.*` (SceneProcessor have `*Processor` or `*Renderer` in their names).

## Particle Emitter

```
public class MyGame extends SimpleApplication {
 public void simpleInitApp() {
 ParticleEmitter pm = new ParticleEmitter("my particle effect",
 Material pmMat = new Material(assetManager, "Common/MatDefs/Misc/
 pmMat.setTexture("Texture", assetManager.loadTexture("Effects/s
 pm.setMaterial(pmMat);
 pm.setImagesX(1);
 pm.setImagesY(1);
 rootNode.attachChild(pm); // attach one or more emitters to any
 }
}
```



## Scene Processor

```
public class MyGame extends SimpleApplication {
 private BasicShadowRenderer bsr;

 public void simpleInitApp() {
 bsr = new BasicShadowRenderer(assetManager, 1024);
 bsr.setDirection(new Vector3f(.3f, -0.5f, -0.5f));
 viewPort.addProcessor(bsr); // add one or more sceneprocessors
 }
}
```

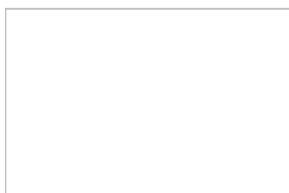
## Post-Processor Filter

```
public class MyGame extends SimpleApplication {
 private FilterPostProcessor fpp; // one FilterPostProcessor per app
 private SomeFilter sf; // one or more Filters per app

 public void simpleInitApp() {
 fpp = new FilterPostProcessor(assetManager);
 viewPort.addProcessor(fpp); // add one FilterPostProcessor

 sf = new SomeFilter();
 fpp.addFilter(sf); // add one or more Filters to FilterPostProcessor
 }
}
```

## Water



The jMonkeyEngine's "[SeaMonkey](#)" WaterFilter

simulates ocean waves, foam, including cool underwater caustics. Use the SimpleWaterProcessor (SceneProcessor) for small, limited bodies of water, such as puddles, drinking troughs, pools, fountains.

See also the [Rendering Water as Post-Process Effect](#) announcement with video.

- [jme3/src/test/jme3test/water/TestSceneWater.java](#) – SimpleWaterProcessor (SceneProcessor)

- [jme3/src/test/jme3test/water/TestSimpleWater.java](#) – SimpleWaterProcessor (SceneProcessor)



- [jme3/src/test/jme3test/water/TestPostWater.java](#) – WaterFilter
- [jme3/src/test/jme3test/water/TestPostWaterLake.java](#) – WaterFilter

## Environment Effects

### Depth of Field Blur



- [jme3/src/test/jme3test/post/TestDepthOfField.java](#) – DepthOfFieldFilter

### Fog

- [jme3/src/test/jme3test/post/TestFog.java](#) – FogFilter

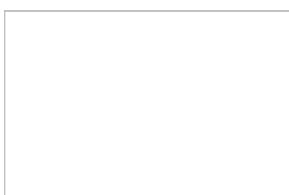
### Light Scattering

- [jme3/src/test/jme3test/post/TestLightScattering.java](#) – LightScatteringFilter

### Vegetation

- Contribution: [Grass System](#)
- Contribution: [Trees \(WIP\)](#)

### Light and Shadows



## Bloom and Glow

- [jme3/src/test/jme3test/post/TestBloom.java](#) – BloomFilter
- More details: [Bloom and Glow](#) – BloomFilter

## Light

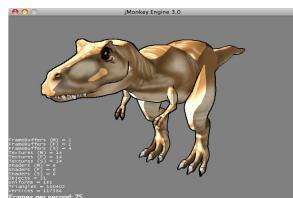
- [jme3/src/test/jme3test/light/TestSimpleLighting.java](#) – DirectionalLight, PointLight
- [jme3/src/test/jme3test/light/TestLightRadius.java](#) – DirectionalLight, PointLight
- [jme3/src/test/jme3test/light/TestManyLights.java](#) – .j3o scene
- More details: [Light and Shadow](#)



## Shadow

- [jme3/src/test/jme3test/light/TestShadow.java](#) – BasicShadowRenderer (SceneProcessor)
- [jme3/src/test/jme3test/light/TestPssmShadow.java](#) – PssmShadowRenderer (SceneProcessor), also known as Parallel-Split Shadow Mapping (PSSM).
- [jme3/src/test/jme3test/post/TestSSAO.java](#), [jme3/src/test/jme3test/post/TestSSAO2.java](#) – SSAOFilter, also known as Screen-Space Ambient Occlusion shadows (SSOA).
- [jme3/src/test/jme3test/post/TestTransparentSSAO.java](#) – SSAOFilter, also known as Screen-Space Ambient Occlusion shadows (SSOA), plus transparency
- More details: [Light and Shadow](#)

## Special: Glass, Metal, Dissolve, Toon



## Toon Effect

- [jme3/src/test/jme3test/post/TestCartoonEdge.java](#) – CartoonEdgeFilter
- [jme3/src/test/jme3test/post/TestTransparentCartoonEdge.java](#) – CartoonEdgeFilter

## Fade in / Fade out

- [Fade](#) – FadeFilter

## User Contributed



### [ShaderBlow - GLSL Shader Library](#)

- LightBlow Shader – blend material texture maps
- FakeParticleBlow Shader – jet, fire effect
- ToonBlow Shader – Toon Shading, toon edges
- Dissolve Shader – Scifi teleportation/dissolve effect
- MatCap Shader – Gold, metals, glass, toons...!
- Glass Shader – Glass
- Force Shield Shader – Scifi impact-on-force-field effect
- SimpleSprite Shader – Animated textures
- SimpleSpriteParticle Shader – Sprite library
- MovingTexture Shader – Animated cloud/mist texture
- SoftParticles Shader – Fire, clouds, smoke etc
- Displace Shader – Deformation effect: Ripple, wave, pulse, swell!

Thanks for your awesome contributions! Keep them coming!

## Particle Emitters: Explosions, Fire, Smoke



[Particle emitter effects](#) are highly configurable

and can have any texture. They can simulate smoke, dust, leaves, meteors, snowflakes, mosquitos, fire, explosions, clusters, embers, sparks...

- [jme3/src/test/jme3test/effect/TestExplosionEffect.java](#) – debris, flame, flash, shockwave, smoke, sparks

- [jme3/src/test/jme3test/effect/TestPointSprite.java](#) – cluster of points
  - [jme3/src/test/jme3test/effect/TestMovingParticle.java](#) – dust, smoke
- 

## Creating your own Filters

Here is an extract taken from @nehon in the forum thread  
(<http://hub.jmonkeyengine.org/forum/topic/how-exactly-do-filters-work/>)

The methods are called in this order (pretty much the same flow as processors): - initFilter() is called once when the FilterPostPorcessor is initialized or when the filter is added to the processor and this one as already been initialized.

for each frame the methods are called in that sequence : - preFrame() occurs before anything happens - postQueue() occcurs once the queues have been populated (there is one queue per bucket and 2 additional queues for the shadows, casters and recievers). Note that geometries in the queues are the one in the view frustum. - postFrame occurs once the main frame has been rendered (the back buffer)

Those methods are optional in a filter, they are only there if you want to hook in the rendering process.

The material variable is here for convenience. You have a getMaterial method that returns the material that's gonna be used to render the full screen quad. It just happened that in every implementation I had a material attribute in all my sub-classes, so I just put it back in the abstract class. Most of the time getMaterial returns this attribute.

Forced-technique can be any technique really, they are more related with the material system than to the filters but anyway. When you use a forced technique the renderer tries to select it on the material of each geometry, if the technique does not exists for the material the geometry is not rendered. You assume well about the SSAO filer, the normal of the scene are rendered to a texture in a pre pass.

Passes : these are filters in filters in a way. First they are a convenient way to initialize a FrameBuffer and the associated textures it needs, then you can use them for what ever you want. For example, a Pass can be (as in the SSAO filter) an extra render of the scene with a forced technique, and you have to handle the render yourself in the postQueue method. It can be a post pass to do after the main filter has been rendered to screen (for example an additional blur pass used in SSAO again). You have a list of passes called postRenderPass in the Filter abstract class. If you add a pass to this list, it'll be automatically rendered by the FilterPostProcessor during the filter chain.

The bloom Filter does an intensive use of passes.

Filters in a nutshell.

---

See also:

- [Particle Emitters](#)
- [Bloom and Glow](#)
- [Photoshop Tutorial for Sky and space effects \(article\)](#)

[documentation](#), [effect](#), [light](#), [water](#)

</div>

## title: jME3 Headless Server

# jME3 Headless Server

When adding multiplayer to your game, you may find that your server needs to know about game state (e.g. where are players, objects? Was that a direct hit? etc.) You can code all this up yourself, but there's an easier way.

It's very easy to change your current (client) game to function as a server as well.

## What Does Headless Mean?

A headless server...

- does not display any output – no window opens, no audio plays, no graphics are rendered.
- ignores all input – no input handling.
- keeps game state – you can attach to, transform, and save the rootNode, although the scene is not displayed.
- calls the `simpleUpdate()` loop – you can run tests and trigger events as usual.

## Client Code

First, let's take a look at the default way of creating a new game (in its simplest form):

```
public static void main(String[] args) {
 Application app = new Main();
 app.start();
}
```

## Headless Server Code

Now, with a simple change you can start your game in Headless mode. This means that all input and audio/visual output will be ignored. That's a good thing for a server.

```
import com.jme3.system.JmeContext;
import com.jme3.system.JmeContext.Type;

public static void main(String[] args) {
 Application app = new Main();
 app.start(JmeContext.Type.Headless);
}
```

## Next steps

Okay, so you can now start your game in a headless 'server mode', where to go from here?

- Parse `String[] args` from the `main`-method to enable server mode on demand (e.g. start your server like `java -jar mygame.jar -server`).
- Integrate [SpiderMonkey](#), to provide game updates to the server over a network.
- Only execute code that's needed. (E.g. place all rendering code inside an `if (servermode) -block`) (or `if (!servermode)` for the client).
- Add decent [logging](#) so your server actually makes sense.

[server](#), [spidermonkey](#), [headless](#), [network](#), [documentation](#)

</div>

## title: Physical Hinges and Joints

# Physical Hinges and Joints

The jMonkeyEngine3 has built-in support for [jBullet physics](#) via the `com.jme3.bullet` package.

Game Physics are not only employed to calculate collisions, but they can also simulate hinges and joints. Think of pulley chains, shaky rope bridges, swinging pendulums, or (trap)door and chest hinges. Physics are a great addition to e.g. an action or puzzle game.

In this example, we will create a pendulum. The joint is the (invisible) connection between the pendulum body and the hook. You will see that you can use what you learn from the simple pendulum and apply it to other joint/hinge objects (rope bridges, etc).

## Sample Code

- [TestPhysicsHingeJoint.java](#)

## Overview of this Physics Application

1. Create a SimpleApplication with a [BulletAppState](#)
  - This gives us a PhysicsSpace for PhysicsControls
2. For the pendulum, we use a Spatial with a PhysicsControl, and we apply physical forces to them.
  - The parts of the “pendulum” are Physics Control’ed Spatial with Collision Shapes.
  - We create a fixed `hookNode` and a dynamic `pendulumNode`.
3. We can “crank the handle” and rotate the joint like a hinge, or we can let loose and expose the joints freely to gravity.
  - For physical forces we will use the method `joint.enableMotor();`

## Creating a Fixed Node

The `hookNode` is the fixed point from which the pendulum hangs. It has no mass.

```

Node hookNode=PhysicsTestHelper.createPhysicsTestNode(
 assetManager, new BoxCollisionShape(new Vector3f(.1f, .1f, .1f
hookNode.getControl(RigidBodyControl.class).setPhysicsLocation(new

rootNode.attachChild(hookNode);
getPhysicsSpace().add(hookNode);

```

For a rope bridge, there would be two fixed nodes where the bridge is attached to the mountainside.

## Creating a Dynamic Node

The pendulumNode is the dynamic part of the construction. It has a mass.

```

Node pendulumNode=PhysicsTestHelper.createPhysicsTestNode(
 assetManager, new BoxCollisionShape(new Vector3f(.3f, .3f, .3f
pendulumNode.getControl(RigidBodyControl.class).setPhysicsLocation(
rootNode.attachChild(pendulumNode);
getPhysicsSpace().add(pendulumNode);

```

For a rope bridge, each set of planks would be one dynamic node.

## Understanding DOF, Joints, and Hinges

A PhysicsHingeJoint is an invisible connection between two nodes – here between the pendulum body and the hook. Why are hinges and joints represented by the same class? Hinges and joints have something in common: They constrain the *mechanical degree of freedom* (DOF) of another object.

Consider a free falling, “unchained” object in physical 3D space: It has 6 DOFs:

- It translates along 3 axes
- It rotates around 3 axes

Now consider some examples of objects with joints:

- An individual chain link is free to spin and move around, but joined into a chain, the link's movement is restricted to stay with the surrounding links.

- A person's arm can rotate around some axes, but not around others. The shoulder joint allows one and restricts the other.
- A door hinge is one of the most restricted types of joint: It can only rotate around one axis.

You'll understand that, when creating any type of joint, it is important to correctly specify the DOFs that the joint restricts, and the DOFs that the joint allows. For the typical DOF of a [ragDoll](#) character's limbs, jME even offers a special joint, `ConeJoint`.

## Creating the Joint

You create the HingeJoint after you have created the nodes that are to be chained together. In the code snippet you see that the HingeJoint constructor requires the two node objects. You also have to specify axes and pivots – they are the degrees of freedom that you just heard about.

```
private HingeJoint joint;
...
public void simpleInitApp() {
 ...
 // hookNode and pendulumNode are created here...
 ...

 joint=new HingeJoint(hookNode.getControl(RigidBodyControl.class)
 pendulumNode.getControl(RigidBodyControl.class)
 new Vector3f(0f, 0f, 0f), // pivot point location
 new Vector3f(0f, 1f, 0f), // pivot point location
 Vector3f.UNIT_Z, // DoF Axis of A (Z)
 Vector3f.UNIT_Z); // DoF Axis of B (Z)
```

The pivot point's position will be at `(0,0,0)` in the global 3D space. In A's local space that is at `(0,0,0)` and in B's local space (remember B's position was set to `(0, -1, 0)`) that is at `(0,1,0)`.

Specify the following parameters for each joint:

- PhysicsControl A and B – the two nodes that are to be joined
- Vector3f pivot A and pivot B – coordinates of the attachment point relative to A and B
  - The points typically lie on the surface of the PhysicsControl's Spatial, rarely in the middle.

- Vector3f axisA and axisB – around which axes each node is allowed to spin.
  - In our example, we constrain the pendulum to swing only along the Z axis.

Remember to add all joint objects to the physicsSpace, just like you would do with any physical objects.

```
bulletAppState.getPhysicsSpace().add(joint);
```

**Tip:** If you want the joint to be visible, attach a geometry to the dynamic node, and translate it to its start position.

## Apply Physical Forces

You can apply forces to dynamic nodes (the ones that have a mass), and see how other joined (“chained”) objects are dragged along.

Alternatively, you can also apply forces to the joint itself. In a game, you may want to spin an automatic revolving door, or slam a door closed in a spooky way, or dramatically open the lid of a treasure chest.

The method to call on the joint is `enableMotor()`.

```
joint.enableMotor(true, 1, .1f);
joint.enableMotor(true, -1, .1f);
```

1. Switch the motor on by supplying `true`
2. Specify the velocity with which the joint should rotate around the specified axis.
  - Use positive and negative numbers to change direction.
3. Specify the impulse for this motor. Heavier masses need a bigger impulse to be moved.

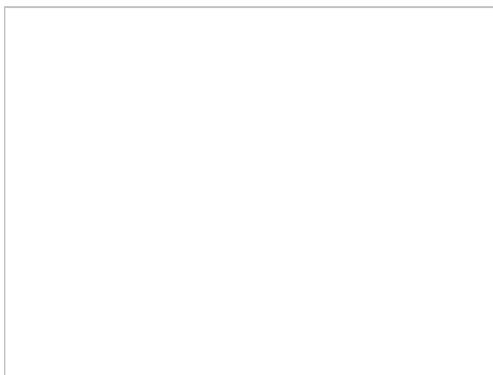
When you disable the motor, the chained nodes are exposed to gravity again:

```
joint.enableMotor(false, 0, 0);
```

[documentation](#), [physics](#), [joint](#)  
 </div>

## **title: Head-Up Display (HUD)**

# **Head-Up Display (HUD)**



A HUD (Head-Up Display) is part of a game's visual user interface. It's an overlay that displays additional information as (typically) 2-dimensional text or icons on the screen, on top of the 3D scene. Not all games have, or need a HUD. To avoid breaking the immersion and cluttering the screen, only use a HUD if it is the only way to convey certain information.

HUDs are used to supply players with essential information about the game state.

- Status: Score, minimap, points, stealth mode, ...
- Resources: Ammunition, lives/health, time, ...
- Vehicle instruments: Cockpit, speedometer, ...
- Navigational aides: Crosshairs, mouse pointer or hand, ...

You have multiple options how to create HUDs.

Option	Pros	Cons
<b>Attach elements to default guiNode:</b>	Easy to learn. jMonkeyEngine built-in <a href="#">API</a> for attaching plain images and bitmap text.	Only basic features. You will have to write custom controls / buttons / effects if you need them.
<b>Use advanced <a href="#">Nifty GUI</a> integration:</b>	Full-featured interactive user interface. Includes buttons, effects, controls. Supports XML and Java layouts.	Steeper learning curve.
<b>Use user contributed GUI libraries such as <a href="#">tonegodgui</a> or <a href="#">Lemur</a>:</b>	Both have many features that would be difficult to do with Nifty Includes buttons, effects, controls. New features are still being released	Are not necessarily guaranteed future updates, not as well documented

Using the [GUI](#) Node is the default approach in jme3 to create simple HUDs. If you just quickly want to display a line of text, or a simple icon on the screen, use the no-frills [GUI](#) Node, it's easier.

## Simple HUD: GUI Node

You already know the `rootNode` that holds the 3-dimensional scene graph. jME3 also offers a 2-dimension (orthogonal) node, the `guiNode`.

This is how you use the `guiNode` for HUDs:

- Create a [GUI](#) element: a `BitmapText` or `Picture` object.
- Attach the element to the `guiNode`.
- Place the element in the orthogonal render queue using `setQueueBucket(Bucket.Gui)`.

The `BitmapTexts` and `Pictures` appear as 2 dimensional element on the screen.

By default, the `guiNode` has some scene graph statistics attached. To clear the `guiNode` before you attach your own [GUI](#) elements, use the following methods:

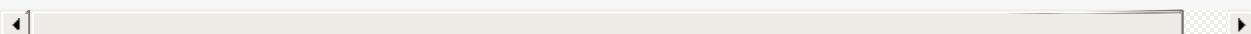
```
setDisplayStatView(false); setDisplayFps(false);
```

```
</div>
```

## Displaying Pictures in the HUD

A simple image can be displayed using `com.jme3.ui.Picture`.

```
Picture pic = new Picture("HUD Picture");
pic.setImage(assetManager, "Textures/ColoredTex/Monkey.png", true);
pic.setWidth(settings.getWidth()/2);
pic.setHeight(settings.getHeight()/2);
pic.setPosition(settings.getWidth()/4, settings.getHeight()/4);
guiNode.attachChild(pic);
```



When you set the last boolean in `setImage()` to true, the alpha channel of your image is rendered transparent/translucent.

## Displaying Text in the HUD

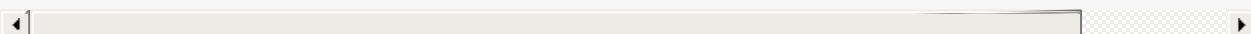
You use `com.jme3.font.BitmapText` to display text on the screen.

```
BitmapText hudText = new BitmapText(guiFont, false);
hudText.setSize(guiFont.getCharSet().getRenderedSize()); // font size
hudText.setColor(ColorRGBA.Blue); // font color
hudText.setText("You can write any string here"); // the text
hudText.setLocalTranslation(300, hudText.getLineHeight(), 0); // position
guiNode.attachChild(hudText);
```



The `BitmapFont` object `guiFont` is a default font provided by `SimpleApplication`. Copy your own fonts as `.fnt` plus `.png` files into the `assets/Interface/Fonts` directory and load them like this:

```
BitmapFont myFont = assetManager.loadFont("Interface/Fonts/Console.fnt");
hudText = new BitmapText(myFont, false);
```



## Positioning HUD Elements

- When positioning GUI text and images in 2D, the **bottom left corner** of the screen is `(0f, 0f)`, and the **top right corner** is at `(settings.getWidth(), settings.getHeight())`.
- If you have several 2D elements in the GUI bucket that overlap, define their depth order

by specifying a Z value. For example use `pic.move(x, y, -1)` to move the picture to the background, or `hudText.setLocalTranslation(x,y,1)` to move text to the foreground.

- Size and length values in the orthogonal render queue are treated like pixels. A 20\*20-wu big quad is rendered 20 pixels wide.

## Displaying Geometries in the HUD

It is technically possible to attach Quads and 3D Geometries to the HUD. They show up as flat, static GUI elements. The size unit for the guiNode is pixels, not world units. If you attach a Geometry that uses a lit Material, you must add a light to the guiNode.

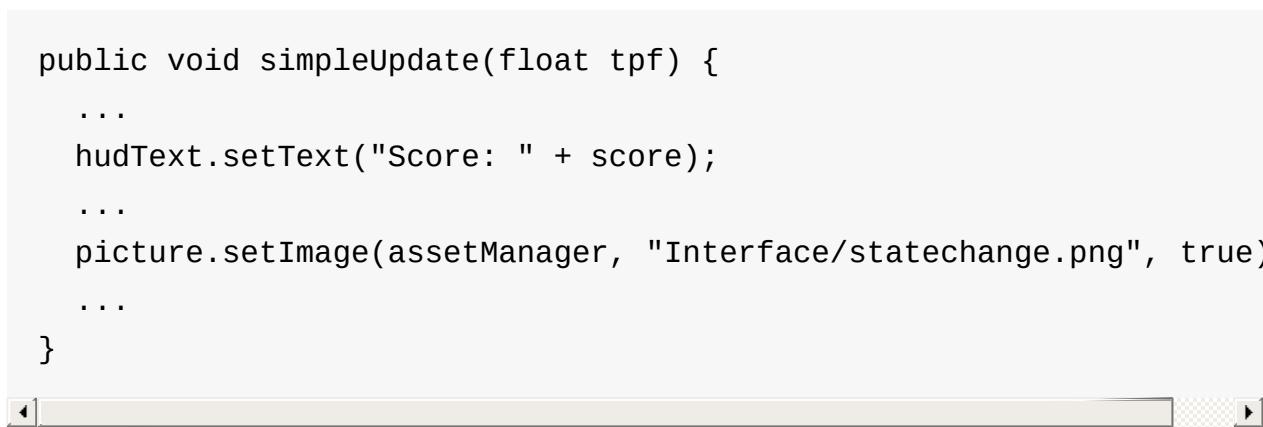
If you don't see an attached object in the GUI, check it's position and material (add a light to guiNode). Also verify whether it is not too tiny to be seen. For comparison: A 1 world-unit wide cube is only 1 pixel wide when attached to the guiNode! You may need to scale it bigger.

</div>

## Keeping the HUD Up-To-Date

Use the update loop to keep the content up-to-date.

```
public void simpleUpdate(float tpf) {
 ...
 hudText.setText("Score: " + score);
 ...
 picture.setImage(assetManager, "Interface/statechange.png", true)
 ...
}
```



## Advanced HUD: Nifty GUI

The recommended approach to create HUDs is using [Nifty GUI](#).

1. Lay out the GUI in one or several Nifty XML or Java files.
2. Write the controller classes in Java.
3. Load the XML file with the controller object in your game's `simpleInit()` method.

The advantage of Nifty GUI is that it is well integrated into jME and the jMonkeyEngine SDK, and that it offers all the features that you expect from a professional modern user interface.

For HUDs, you basically follow the same instructions as for creating a normal [Nifty GUI](#), you just don't pause the game while the HUD is up.

## See also

- [Fonts](#)

[gui](#), [display](#), [documentation](#), [hud](#)

</div>

## title: Input Handling

# Input Handling

Users interact with your jME3 application with different input devices – the mouse, the keyboard, or a joystick. To respond to inputs we use the `inputManager` object in `SimpleApplication`.

This is how you add interaction to your game:

1. For each action, choose the trigger(s) (a key or mouse click etc)
2. For each action, add a trigger mapping to the `inputManager`
3. Create at least one listener in `SimpleApplication`
4. For each action, register its mappings to a listener
5. Implement each action in the listener

## Code Samples

- [TestControls.java](#)
- [TestJoystick.java](#)

## 1. Choose Trigger

Choose one or several key/mouse events for the interaction. We use `KeyTrigger`, `MouseAxisTrigger`, `MouseButtonTrigger`, `JoyAxisTrigger` and `JoyButtonTrigger` constants from the `com.jme3.input.controls` package.

**Note:** The `MouseAxis` and `JoyAxis` triggers go along the X axis (right/left) or Y axis (up/down). These Triggers come with extra booleans for the negative half of the axis (left, down). Remember to write code that listens to the negative (true) and positive (false) axis!

Trigger	Code
Mouse button: Left Click	<code>MouseButtonTrigger(MouseInput.BUTTON_LEFT)</code>
Mouse button: Right Click	<code>MouseButtonTrigger(MouseInput.BUTTON_RIGHT)</code>
Mouse button: Middle Click	<code>MouseButtonTrigger(MouseInput.BUTTON_MIDDLE)</code>

Mouse movement: Right	MouseAxisTrigger(MouseInput.AXIS_X, true)
Mouse movement: Left	MouseAxisTrigger(MouseInput.AXIS_X, false)
Mouse movement: Up	MouseAxisTrigger(MouseInput.AXIS_Y, true)
Mouse movement: Down	MouseAxisTrigger(MouseInput.AXIS_Y, false)
Mouse wheel: Up	MouseAxisTrigger(MouseInput.AXIS_WHEEL, false)
Mouse wheel: Down	MouseAxisTrigger(MouseInput.AXIS_WHEEL, true)
NumPad: 1, 2, 3, ...	KeyTrigger(KeyInput.KEY_NUMPAD1) ...
Keyboard: 1, 2 , 3, ...	KeyTrigger(KeyInput.KEY_1) ...
Keyboard: A, B, C, ...	KeyTrigger(KeyInput.KEY_A) ...
Keyboard: Spacebar	KeyTrigger(KeyInput.KEY_SPACE)
Keyboard: Shift	KeyTrigger(KeyInput.KEY_RSHIFT), KeyTrigger(KeyInput.KEY_LSHIFT)
Keyboard: F1, F2, ...	KeyTrigger(KeyInput.KEY_F1) ...
Keyboard: Return, Enter	KeyTrigger(KeyInput.KEY_RETURN), KeyTrigger(KeyInput.KEY_NUMPADENTER)
Keyboard: PageUp, PageDown	KeyTrigger(KeyInput.KEY_PGUP), KeyTrigger(KeyInput.KEY_PGDN)
Keyboard: Delete, Backspace	KeyTrigger(KeyInput.KEY_BACK), KeyTrigger(KeyInput.KEY_DELETE)
Keyboard: Escape	KeyTrigger(KeyInput.KEY_ESCAPE)
Keyboard: Arrows	KeyTrigger(KeyInput.KEY_DOWN), KeyTrigger(KeyInput.KEY_UP) KeyTrigger(KeyInput.KEY_LEFT), KeyTrigger(KeyInput.KEY_RIGHT)
Joystick Button:	JoyButtonTrigger(0, JoyInput.AXIS_POV_X), JoyButtonTrigger(0, JoyInput.AXIS_POV_Y) ?
Joystick Movement: Right	JoyAxisTrigger(0, JoyInput.AXIS_POV_X, true)
Joystick Movement: Left	JoyAxisTrigger(0, JoyInput.AXIS_POV_X, false)
Joystick Movement: Forward	JoyAxisTrigger(0, JoyInput.AXIS_POV_Z, true)
Joystick Movement: Backward	JoyAxisTrigger(0, JoyInput.AXIS_POV_Z, false)

In your IDE, use code completion to quickly look up Trigger literals. In the jMonkeyEngine SDK for example, press **ctrl-space** or **ctrl-/** after `KeyInput.` to choose from the list of all keys.

## 2. Remove Default Trigger Mappings

```
inputManager.deleteMapping(SimpleApplication.INPUT_MAPPING_MEMORY
```

Default Mapping	Key	Description
INPUT_MAPPING_HIDE_STATS	F5	Hides the statistics in the bottom left.
INPUT_MAPPING_CAMERA_POS	KEY_C	Prints debug output about the camera.
INPUT_MAPPING_MEMORY	KEY_M	Prints debug output for memory usage.
INPUT_MAPPING_EXIT	KEY_ESCAPE	Closes the application by calling <code>stop();</code> . Typically you do not remove this, unless you replace it by another way of quitting gracefully.

## 3. Add Custom Trigger Mapping

When initializing the application, add a Mapping for each Trigger.

Give the mapping a meaningful name. The name should reflect the action, not the button/key (because buttons/keys can change). Here some examples:

```
inputManager.addMapping("Pause Game", new KeyTrigger(KeyInput.KEY_F)
inputManager.addMapping("Rotate", new KeyTrigger(KeyInput.KEY_S
...

```

There are cases where you may want to provide more than one trigger for one action. For example, some users prefer the WASD keys to navigate, while others prefer the arrow keys. Add several triggers for one mapping, by separating the Trigger objects with commas:

```
inputManager.addMapping("Left", new KeyTrigger(KeyInput.KEY_A),
 new KeyTrigger(KeyInput.KEY_LEFT))
inputManager.addMapping("Right", new KeyTrigger(KeyInput.KEY_D),
 new KeyTrigger(KeyInput.KEY_RIGHT))
...

```

## 4. Create Listeners

The jME3 input manager supports two types of event listeners for inputs: AnalogListener and ActionListener. You can use one or both listeners in the same application. Add one or both of the following code snippets to your main SimpleApplication-based class to activate the listeners.

**Note:** The two input listeners do not know, and do not care, which actual key was pressed. They only know which *named input mapping* was triggered.

### ActionListener

```
com.jme3.input.controls.ActionListener
```

- Use for absolute “button pressed or released?”, “on or off?” actions.
  - Examples: Pause/unpause, a rifle or revolver shot, jump, click to select.
- JME gives you access to:
  - The mapping name of the triggered action.
  - A boolean whether the trigger is still pressed or has just been released.
  - A float of the current time-per-frame as timing factor
- 

```
private ActionListener actionListener = new ActionListener() {
 public void onAction(String name, boolean keyPressed, float tpf)
 /** TODO: test for mapping names and implement actions */
 }
};
```

### AnalogListener

```
com.jme3.input.controls.AnalogListener
```

- Use for continuous and gradual actions.

- Examples: Walk, run, rotate, accelerate vehicle, strafe, (semi-)automatic weapon shot
- JME gives you access to:
  - The mapping name of the triggered action.
  - A gradual float value between how long the trigger has been pressed.
  - A float of the current time-per-frame as timing factor

```
private AnalogListener analogListener = new AnalogListener() {
 public void onAnalog(String name, float keyPressed, float tpf) {
 /** TODO: test for mapping names and implement actions */
 }
};
```

## 4. Register Mappings to Listeners

To activate the mappings, you must register them to a Listener. Write your registration code after the code block where you have added the mappings to the `inputManager`.

In the following example, you register the “Pause Game” mapping to the `actionListener` object, because pausing a game is in “either/or” decision.

```
inputManager.addListener(actionListener, new String[]{"Pause Game"})
```

In the following example, you register navigational mappings to the `analogListener` object, because walking is a continuous action. Players typically keep the key pressed to express continuity, for example when they want to “walk on” or “accelerate”.

```
inputManager.addListener(analogListener, new String[]{"Left", "Right"})
```

As you see, you can add several listeners in one String array. You can call the `addListener()` method more than once, each time with a subset of your list, if that helps you keep you code tidy. Again, the Listeners do not care about actual which keys are configured, you only register named trigger mappings.

Did you register an action, but it does not work? Check the string's capitalization and spelling, it's case sensitive!

</div>

## 5. Implement Actions in Listeners

You specify the action to be triggered where it says TODO in the Listener code snippets. Typically, you write a series of if/else conditions, testing for all the mapping names, and then calling the respective action.

Make use of the distinction between `if` and `else if` in this conditional.

- If several actions can be triggered simultaneously, test for all of these with a series of bare `if`s. For example, a character can be running forward *and* to the left.
- If certain actions exclude one another, test for them with `else if`, the the rest of the exclusive tests can be skipped and you save some miliseconds. For example, you either shoot or pick something up.

### ActionListener

In the most common case, you want an action to be triggered once, in the moment when the button or key trigger is released. For example, when the player presses a key to open a door, or clicks to pick up an item. For these cases, use an ActionListener and test for `&& !keyPressed`, like shown in the following example.

```
private ActionListener actionListener = new ActionListener() {
 public void onAction(String name, boolean keyPressed, float tpf

 if (name.equals("Pause Game") && !keyPressed) { // test?
 isRunning = !isRunning; // action!
 }

 if ...
 }
};
```

### AnalogListener

The following example shows how you define actions with an AnalogListener. These actions are triggered continuously, as long (intensity `value`) as the named key or mouse button is down. Use this listeners for semi-automatic weapons and navigational actions.

```
private AnalogListener analogListener = new AnalogListener() {
 public void onAnalog(String name, float value, float tpf) {

 if (name.equals("Rotate")) { // test?
 player.rotate(0, value*speed, 0); // action!
 }

 if ...

 }
};
```

## Let Users Remap Keys

It is likely that your players have different keyboard layouts, are used to “reversed” mouse navigation, or prefer different navigational keys than the ones that you defined. You should create an options screen that lets users customize their mouse/key triggers for your mappings. Replace the trigger literals in the `inputManager.addMapping()` lines with variables, and load sets of triggers when the game starts.

The abstraction of separating triggers and mappings has the advantage that you can remap triggers easily. Your code only needs to remove and add some trigger mappings. The core of the code (the listeners and actions) remains unchanged.

[keyinput](#), [input](#), [documentation](#)  
</div>

## **title: Saving and Loading Materials with .j3m Files**

# **Saving and Loading Materials with .j3m Files**

In the [Material Definitions](#) article you learned how to configure [Materials](#) programmatically in Java code. If you have certain commonly used Materials that never change, you can clean up the amount of Java code that clutters your init method, by moving material settings into .j3m files. Then later in your code, you only need to call one setter instead of several to apply the material.

If you want to colorize simple shapes (one texture all around), then .j3m are the most easily customizable solution. J3m files can contain texture mapped materials, but as usual you have to create the textures in an external editor, especially if you use UV-mapped textures.

## **Writing the .j3m File**

1. For every Material, create a file and give it a name that describes it: e.g.  
`SimpleBump.j3m`
2. Place the file in your project's `assets/Materials/` directory, e.g.  
`MyGame/src/assets/Materials/`  
`SimpleBump.j3m`
3. Edit the file and add content using the following Syntax, e.g.:

```
Material shiny bumpy rock : Common/MatDefs/Light/Lighting.j3md
{
 MaterialParameters {
 Shininess: 8.0
 NormalMap: Textures/bump_rock_normal.png
 UseMaterialColors : true
 Ambient : 0.0 0.0 0.0 1.0
 Diffuse : 1.0 1.0 1.0 1.0
 Specular : 0.0 0.0 0.0 1.0
 }
}
```

How to this file is structured:

1. Header
  - i. `Material` is a fixed keyword, keep it.
  - ii. `shiny bumpy rock` is a descriptive string that you can make up. Choose a name to help you remember for what you intend to use this material.
  - iii. After the colon, specify on which `Material` definition you base this Material.
2. Now look up the chosen Material Definition's parameters and their parameter types from the `Material` table. Add one line for each parameter.
  - For example: The series of four numbers in the example above represent RGBA color values.
3. Check the detailed syntax reference below if you are unsure.

In the jMonkeyEngine SDK, use File→New File→Material→Empty Material File to create .j3m files. You can edit .j3m files directly in the SDK. On the other hand, they are plain text files, so you can also create them in any plain text editor.

</div>

## How to Use .j3m Materials

This is how you use the prepared .j3m Material on a Spatial. Since you have saved the .j3m file to your project's Assets directory, the .j3m path is relative to `MyGame/src/assets/...`.

```
myGeometry.setMaterial(assetManager.loadMaterial("Materials/SimpleE
```

**Tip:** In the jMonkeyEngine SDK, open Windows>Palette and drag the `JME Material: Set J3M` snippet into your code.

## Syntax Reference for .j3m Files

### Paths

Make sure to get the paths to the textures (.png, .jpg) and material definitions (.j3md) right.

- The paths to the built-in .j3md files are relative to jME3's Core Data directory. Just copy the path stated in the `Material` table.
 

`Common/MatDefs/Misc/Unshaded.j3md` is resolved to `jme3/src/src/core-data/Common/MatDefs/Misc/Unshaded.j3md`.
- The paths to your textures are relative to your project's assets directory.

```
Textures/bump_rock_normal.png is resolved to
MyGame/src/assets/Textures/bump_rock_normal.png
```

## Data Types

All data types (except Color) are specified in com.jme3.shader.VarType. “Color” is specified as Vector4 in J3MLoader.java.

Name	jME Java class	.j3m file syntax
Float	(basic Java type)	a float (e.g. 0.72) , no comma or parentheses
Vector2	com.jme3.math.Vector2f	Two floats, no comma or parentheses
Vector3	com.jme3.math.Vector3f	Three floats, no comma or parentheses
Vector4	com.jme3.math.Vector4f	Four floats, no comma or parentheses
Texture2D	com.jme3.texture.Texture2D	Path to texture in assets directory, no quotation marks
Texture3D	com.jme3.texture.Texture3D	Same as texture 2D except it is interpreted as a 3D texture
TextureCubeMap	com.jme3.texture.TextureCubeMap	Same as texture 2D except it is interpreted as a cubemap texture
Boolean	(basic Java type)	true or false
Int	(basic Java type)	Integer number, no comma or parentheses
Color	com.jme3.math.ColorRGBA	Four floats, no comma or parentheses
FloatArray		(Currently not supported in J3M)
Vector2Array		(Currently not supported in J3M)
Vector3Array		(Currently not supported in J3M)
Vector4Array		(Currently not supported in J3M)
Matrix3		(Currently not supported in J3M)
Matrix4		(Currently not supported in

Matrix4		J3M)
Matrix3Array		(Currently not supported in J3M)
Matrix4Array		(Currently not supported in J3M)
TextureBuffer		(Currently not supported in J3M)
TextureArray		(Currently not supported in J3M)

## Flip and Repeat Syntax

- A texture can be flipped using the following syntax `NormalMap: Flip Textures/bump_rock_normal1.png`
- A texture can be set to repeat using the following syntax `NormalMap: Repeat Textures/bump_rock_normal1.png`
- If a texture is set to both being flipped and repeated, Flip must come before Repeat

## Syntax for Additional Render States

- A Boolean can be “On” or “Off”
- Float is “123.0” etc
- Enum - values depend on the enum

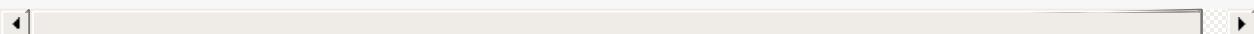
See the [RenderState](#) javadoc for a detailed explanation of render states.

Name	Type	Purpose
Wireframe	(Boolean)	Enable wireframe rendering mode
FaceCull	(Enum: FaceCullMode)	Set face culling mode (Off, Front, Back, FrontAndBack)
DepthWrite	(Boolean)	Enable writing depth to the depth buffer
DepthTest	(Boolean)	Enable depth testing
Blend	(Enum: BlendMode)	Set the blending mode
AlphaTestFalloff	(Float)	Set the alpha testing alpha falloff value (if set, it will enable alpha testing)
PolyOffset	(Float, Float)	Set the polygon offset factor and units
ColorWrite	(Boolean)	Enable color writing
PointSprite	(Boolean)	Enable point sprite rendering for point meshes

# Examples

## Example 1: Shiny

```
Spatial signpost = (Spatial) assetManager.loadAsset(
 new OgreMeshKey("Models/Sign Post/Sign Post.mesh.xml", null));
signpost.setMaterial(assetManager.loadMaterial(
 new AssetKey("Models/Sign Post/Sign Post.j3m")));
TangentBinormalGenerator.generate(signpost);
rootNode.attachChild(signpost);
```



The file `assets/Models/Sign Post/Sign Post.j3m` contains:

```
Material Signpost : Common/MatDefs/Light/Lighting.j3md {
 MaterialParameters {
 Shininess: 4.0
 DiffuseMap: Models/Sign Post/Sign Post.jpg
 NormalMap: Models/Sign Post/Sign Post_normal.jpg
 SpecularMap: Models/Sign Post/Sign Post_specular.jpg
 UseMaterialColors : true
 Ambient : 0.5 0.5 0.5 1.0
 Diffuse : 1.0 1.0 1.0 1.0
 Specular : 1.0 1.0 1.0 1.0
 }
}
```

The JPG files are in the same directory, `assets/Models/Sign Post/...`.

## Example 2: Repeating Texture

```
Material mat = assetManager.loadMaterial(
 "Textures/Terrain/Pond/Pond.j3m");
mat.setColor("Ambient", ColorRGBA.DarkGray);
mat.setColor("Diffuse", ColorRGBA.White);
mat.setBoolean("UseMaterialColors", true);
```

The file `assets/Textures/Terrain/Pond/Pond.j3m` contains:

```

Material Pong Rock : Common/MatDefs/Light/Lighting.j3md {
 MaterialParameters {
 Shininess: 8.0
 DiffuseMap: Repeat Textures/Terrain/Pond/Pond.png
 NormalMap: Repeat Textures/Terrain/Pond/Pond_normal.png
 }
}

```

The PNG files are in the same directory, `assets/Textures/Terrain/Pond/`

## Example 3: Transparent

The file `assets/Models/Tree/Leaves.j3m` contains:

```

Material Leaves : Common/MatDefs/Light/Lighting.j3md {

 Transparent On

 MaterialParameters {
 DiffuseMap : Models/Tree/Leaves.png
 UseAlpha : true
 AlphaDiscardThreshold : 0.5
 UseMaterialColors : true
 Ambient : .5 .5 .5 .5
 Diffuse : 0.7 0.7 0.7 1
 Specular : 0 0 0 1
 Shininess : 16
 }
 AdditionalRenderState {
 Blend Alpha
 AlphaTestFalloff 0.50
 FaceCull Off
 }
}

```

The PNG file is in the same directory, `assets/Models/Tree/...`

## Related Links

- Developer specification of the jME3 material system (.j3md,.j3m)

[material](#), [texture](#), [file](#), [sdk](#), [wireframe](#), [documentation](#)

</div>

## **title: Render Buckets**

# **Render Buckets**

For each viewport the rendering happens as follows:

- For each processor call preFrame
- Dispatch each geometry in a corresponding renderQueue (one for each Bucket) and build shadow queues
- For each processor call postQueues
- Rendering OpaqueBucket with object sorted front to back. (In order to minimize overdraw)
- Rendering SkyBucket with depth forced to 1.0. this means every object of this bucket will be far away and behind everything
- Rendering TransparentBucket with object sorted back to front. (So transparent objects can be seen correctly through each other)
- For each processor call postFrame
- Rendering TranslucentBucket with objects sorted back to front

The translucent bucket is rendered at the end. That's where you put transparent object that you don't want to be affected by post processing ( shadows or what ever). Self-light-emitting particle emitters (such as a fire) are a good example.

Post processors are not applied to this bucket with one exception : the FilterPostProcessor.

The filter post processor hijacks the rendering process and renders a full screen quad with the texture of the scene applied on it.

Once it's done the depth buffer is 0, so it's impossible to render a queue over it with proper depth test so if you use a FilterPostProcessor you have to add at the end of your filter stack the TranslucentBucketFilter. It will handle the translucent bucket rendering instead of the RenderManager. (Of course the correct depth information is passed to the filter).

The nice side effect of this is that if you want to apply a post filter to your translucent bucket (like bloom for example) you can just push up the translucent bucket filter in the filter stack.

# title: Shader Nodes

## Shader Nodes

### Motivations

jME3 material system is entirely based on shaders. While it's pretty powerful, this system has some issues and limitations :

- Monolithic shaders have a serious lack of flexibility, and it can be daunting to get into the code for inexperienced users.
- Maintenance ease of such shaders is poor. (for example the whole lighting shaders represent around 500 lines of code, and it could be a lot worse with more features)
- Adding new features to those shaders decrease the ease of maintenance a lot. This point made us reluctant to do so and some feature were never added (Fog to name it, but many more).
- Users can't add their own feature to the shader unless they fork it, and fall back to the same issues explained in previous points.

Shader Nodes were designed with this in mind and are the fruit of many long discussions in the core chat balancing the pros and cons of this or that pattern.

At first this system was referred to as "Shader injection". The basic idea was to allow users to inject code into shaders with a tag replacement system.

We finally came with a different concept called Shader Nodes, that is inspired from blender nodes system for textures and post process.

**The final shader is generated at run time by the system by assembling shader nodes together.**

### What is a Shader Node?

Conceptually, it's just a self sufficient piece of glsl code that accepts inputs and produce some outputs.

Inputs are glsl variables that may be fed by previous nodes output values.

Outputs are glsl variables fed with values computed in the shader node code.

In practice it's a bit more than that. A shader node is declined in several parts :

- A shader node definition, defining :

- The type of shader node (Vertex or Fragment for now)
- The minimal glsl version needed for the shader node
- The path to the shader file (containing the shader code heh)
- A **mandatory documentation block** for this Shader node. As I hope many users will do their own nodes and contribute them back this point is crucial.
- A list of inputs accepted by this shader (typed glsl variable optional or needed for the code to run properly)
- A list of outputs produced by this shader (typed glsl variable computed and fed by the node's code)
- The actual shader code file (.vert or .frag depending on the node's type like any shader file)
- A shader node declaration having :
  - A unique name(in the shader scope)
  - The shader node definition it's based on
  - An optional activation condition (based on the actual define system)
  - A list of input mapping (what will be actually fed to those inputs)
  - A list of output mapping (what will be output to the global output of the shader)

## Shader Node definition

First ShaderNodes have to be defined either in a separate file (j3sn for jme3 shader node) or directly embed in the Technique block of the j3md file.

Please refer to this documentation for global structure of a j3md file [jMonkeyEngine3 Material Specification](#)

All is included in a **ShaderNodeDefinitions** bloc. This block can have several nodes defined (it's recommended to define nodes that have strong dependencies with each other in the same j3sn file).

A ShaderNode is declared in a **ShaderNodeDefinition** block.

global structure should look like this :

```

ShaderNodeDefinitions{
 ShaderNodeDefinition <NodeDefName>{
 Type : <ShaderType>
 Shader <ShaderLangAndVersion> : <ShaderPath>
 [Shader <ShaderLangAndVersion> : <ShaderPath>]
 [...]
 Documentation {
 <DocContent>
 }
 Input {
 <GlslVarType> <VarName>
 [<GlslVarType> <VarName>]
 [...]
 }
 Output {
 <GlslVarType> <VarName>
 [<GlslVarType> <VarName>]
 [...]
 }
 }
 [ShaderNodeDefinition <NodeDef2Name> {
 [...]
 }]
 [...]
}

```

All that is not between [] is mandatory.

- **ShaderNodeDefinition** : the definition block. You can have several definition in the same ShaderNodeDefinitions block.
  - **NodeDefName** : The name of this ShaderNodeDefinition
- **Type** : define the type of this shader node
  - **ShaderType** : The type of shader for this definition. For now only “Vertex” and “Fragment” are supported.
- **Shader** : the version and path of the shader code to use. note that you can have several shader with different GLSL version. The generator will pick the relevant one according to GPU capabilities.
  - **ShaderLangAndVersion** : follows the same syntax than the shader declaration in the j3md file : GLSL<version>, version being 100 for glsl 1.0 , 130 for glsl 1.3, 150 for glsl 1.5 and so on. Note that this is the **minimum** glsl version this shader

supports

- **ShaderPath** the path to the shader code file (relative to the asset folder)
- *Documentation* : the documentation block. This is mandatory and I really recommend filling this if you want to contribute your shader nodes. This documentation will be read by the SDK and presented to users willing to add this node to their material definitions. This should contain a brief description of the node and a description for each input and output.
  - **@input** can be used to prefix an input name so the sdk recognize it and format it accordingly. the syntax id **@input <inputName> <description>**.
  - **@output** can be used to prefix an output name so the sdk recognize it and format it accordingly. the syntax id **@output <inputName> <description>**
- *Input* : The input block containing all the inputs of this node. A node can have 1 or several inputs.
  - **GlslVarType** : a valid glsl variable type that will be used in the shader for this input. see [http://www.opengl.org/wiki/GLSL\\_Type](http://www.opengl.org/wiki/GLSL_Type) and the “Declare an array” chapter
  - **VarName** : the name of the variable. Note that you can't have several inputs with the same name.
- *Output* : The output block containing all the outputs of this node. A node can have 1 or several outputs.
  - **GlslVarType** : a valid glsl variable type that will be used in the shader for this input. see [http://www.opengl.org/wiki/GLSL\\_Type](http://www.opengl.org/wiki/GLSL_Type) and the “Declare an array” chapter
  - **VarName** : the name of the variable. Note that you can't have several outputs with the same name.

**Note that if you use the same name for an input and an output, the generator will consider them as the SAME variable so they should be of the same glsl type.**

## Example

Here is a typical shader node definition

```

ShaderNodeDefinitions{
 ShaderNodeDefinition LightMapping{
 Type: Fragment
 Shader GLSL100: Common/MatDefs/ShaderNodes/LightMapping/lic
 Documentation {
 This Node is responsible for multiplying a light mapping
 @input texCoord the texture coordinates to use for light mapping
 @input lightMap the texture to use for light mapping
 @input color the color the lightmap color will be multiplied by
 @output color the resulting color
 }
 Input{
 vec2 texCoord
 sampler2D lightMap
 vec4 color
 }
 Output{
 vec4 color
 }
 }
}

```



## Declare an array

To declare an array you have to specify its size between square brackets.

### Constant size

The size can be an int constant

#### *Example*

```
float myArray[10]
```

this will declare a float array with 10 elements. Any material parameter mapped with this array should be of `FloatArray` type and it's size will be assumed as 10 when the shader is generated.

### Material parameter driven size

The size can be dynamic and driven by a material parameter. GLSL does not support non constant values for array declaration so this material parameter will be mapped to a define.

#### *Example*

```
float myArray[NumberOfElements]
```

This declares a float array with the size depending on the value of the `NumberOfElement` material parameter.

`NumberOfElement` **HAS** to be declared in the material definition as a material parameter. It will be mapped to a define and used in the shader. Note that if this value change the shader will have to be recompiled, due to the fact that it's mapped to a define.

## Shader Node code

The shader code associated with a Shader node is similar to any shader code. the code for a Vertex shader node should be in a `.vert` file and the code for a Fragment shader node should be in a `.frag` file. It has a declarative part containing variable declaration, function declaration and so on... And a main part that is embed in a “`void main(){}`” block. Input and output variables declared in the shader node definition can be used **without** being declared in the shader code. ( they shouldn't even or you'll have issues). Here is the code of the `LightMap.frag` shader.

```
void main(){
 color *= texture2D(lightMap, texCoord);
}
```

Very simple, it's just a texture fetch, but of course anything can be done.

**Do not declare uniforms, attributes or varyings in a shader node code**, the Generator will handle this, just use the inputs and outputs and optional local variables you may need.

## Shader Node declaration

To create a shader we need to plug shader nodes to each other, but also interact with built in glsl inputs and outputs. Shader nodes are declared inside the `Technique` block. The vertex nodes are declared in the `VertexShaderNodes` block and the fragment nodes are declared in the `FragmentShaderNodes` block.

Note that if the j3md has ember shader nodes definition (in a `ShaderNodesDefinitions` block) it **must** be declared before the `VertexShaderNodes` and `FragmentShaderNodes` blocks. Of course there can be several `ShaderNode` declaration in those block.

Here is how a `ShaderNode` declaration should look :

```

ShaderNode <ShaderNodeName>{
 Definition : <DefinitionName> [: <DefinitionPath>]
 [Condition : <ActivationCondition>]
 InputMapping{
 <InputVariableName>[.<Swizzle>] = <NameSpace>.<VarName>[.
 [...]
 }
 [OutputMapping{
 <NameSpace>.<VarName>[.<Swizzle>] = <OutputVariableName>[
 [...]
 }]
}

```



- *ShaderNode* the shader node block
  - **ShaderNodeName** the name of this shader node, can be anything, but has to be **unique** in the shader scope
- *Definition* : a reference to the shader node definition
  - **DefinitionName** : the name of the definition this Node use. this definition can be declared in the same j3md or in its own j3sn file.
  - **DefinitionPath** : in case the definition is declared in it's own j3sn file, you have to set the path to this file here.
- *Condition* a condition that dictates if the node is active or not.
  - **Activationcondition** : The condition for this node to be used. Today we use Defines to use different blocks of code used depending on the state of a Material Parameter. The condition here use the exact same paradigm. A valid condition must be the name of a material parameter or any combinations using logical operators “||”, “&&”, “!” or grouping characters “(” and “)”. The generator will create the corresponding define and the shader node code will be embed into and #ifdef statement.

For example, let's say we have a Color and ColorMap material parameter, this condition “Color || ColorMap” will generate this statement :

```

#ifndef COLOR
#define COLOR 0
#endif

#ifndef COLORMAP
#define COLORMAP 0
#endif

if defined(COLOR) || defined(COLORMAP)
{
 ...
}
#endif

```

- *InputMapping* the wiring of the inputs of this node, coming from previous node's outputs or from built in glsl inputs.

- **InputVariableName** : the name of the variable to map as declared in the definition.
- **Swizzle** : Swizzling for the preceding variable. More information on glsl swizzling on this page [http://www.opengl.org/wiki/GLSL\\_Type](http://www.opengl.org/wiki/GLSL_Type)
- **NameSpace** : The generator will use variable name space to avoid collision between variable names. Name space can be one of these values :
  - **MatParam** : the following variable is a Material Parameter declared in the MaterialParameters block of the materialDefinition
  - **WorldParam** : the following variable is a World Parameter declared in the WorldParameters block of the current technique block. World parameters can be one of those declared in this file :  
<https://github.com/jMonkeyEngine/jmonkeyengine/blob/master/jme3-core/src/main/java/com/jme3/shader/UniformBinding.java>
  - **Attr** : the following variable is a shader attribute. It can be one those declared in the Type enum of the VertexBuffer class  
<https://github.com/jMonkeyEngine/jmonkeyengine/blob/master/jme3-core/src/main/java/com/jme3/scene/VertexBuffer.java>
  - **Global** : the variable is a global variable to the shader. Global variables will be assigned at the end of the shader to glsl built in outputs : gl\_Position for the vertex shader, or to one of the possible outputs of the fragment shader (for example gl\_FragColor). The global variable can have whatever name pleases you, it will be assigned in the order they've been found in the declaration to the shader output. **Global variables can be inputs of a shader node. Global variables are forced to be vec4 and are defaulted to the value of the attribute inPosition in the vertex shader and vec4(1.0)(opaque white color) in the fragment shader.**
  - **The name of a previous shader node** : this must be followed by an output variable of a the named shader node. This is what allows one to plug outputs from a node to inputs of another.
- **VarName** : the name of the variable to assign to the input. This variable must be known in name space declared before.
- **MappingCondition** : Follows the same rules as the activation condition for the shaderNode, this mapping will be embed in a #ifdef statement in the resulting shader.
- **OutputMapping** : This block is optional, as mapping of output will be done in input mapping block of following shaderNodes, except if you want to output a value to the Global output of the shader.
  - **NameSpace** : the name space of the output to assign, this can only be "Global" here.
  - **VarName** : the name of a global output (can be anything, just be aware that 2 different names result in 2 different outputs)

- **OutputVariable** : Must be an output of the current node's definition.
- **MappingCondition** : Same as before.

## Complete material definition and Shader Nodes example

Here is an example of a very simple Material definition that just displays a solid color (controlled by a material parameter) on a mesh.

Shader Nodes only work if there is no shader declared in the technique. If you want to bypass the Shader Nodes, you can put a VertexShader and a FragmentShader statement in the technique and the shader nodes will be ignored.

```

MaterialDef Simple {
 MaterialParameters {
 Color Color
 }
 Technique {
 WorldParameters {
 WorldViewProjectionMatrix
 }
 VertexShaderNodes {
 ShaderNode CommonVert {
 Definition : CommonVert : Common/MatDefs/ShaderNodes
 InputMappings {
 worldViewProjectionMatrix = WorldParam.WorldViewProjectionMatrix
 modelPosition = Global.position.xyz
 }
 OutputMappings {
 Global.position = projPosition
 }
 }
 }
 FragmentShaderNodes {
 ShaderNode ColorMult {
 Definition : ColorMult : Common/MatDefs/ShaderNodes
 InputMappings {
 color1 = MatParam.Color
 color2 = Global.color
 }
 OutputMappings {
 Global.color = outColor
 }
 }
 }
 }
}

```

This Material definition has one Default technique with 2 node declarations.

### **CommonVert Definition**

CommonVert is a vertex shader node that has commonly used input and outputs of a vertex

shader. It also computes the position of the vertex in projection space here is the definition content (Common/MatDefs/ShaderNodes/Common/CommonVert.j3sn) :

```

ShaderNodesDefinitions {
 ShaderNodeDefinition CommonVert {
 Type: Vertex
 Shader GLSL100: Common/MatDefs/ShaderNodes/Common/commonVer
 Documentation {
 This Node is responsible for computing vertex position
 It also can pass texture coordinates 1 & 2, and vertexC
 @input modelPosition the vertex position in model space
 @input worldViewProjectionMatrix the World View Project
 @input texCoord1 The first texture coordinates of the v
 @input texCoord2 The second texture coordinates of the
 @input vertColor The color of the vertex (usually assig
 @output projPosition Position of the vertex in projecti
 @output vec2 texCoord1 The first texture coordinates of
 @output vec2 texCoord2 The second texture coordinates c
 @output vec4 vertColor The color of the vertex (output
 }
 Input{
 vec3 modelPosition
 mat4 worldViewProjectionMatrix
 vec2 texCoord1
 vec2 texCoord2
 vec4 vertColor
 }
 Output{
 vec4 projPosition
 vec2 texCoord1
 vec2 texCoord2
 vec4 vertColor
 }
 }
}

```

Note that texCoord1/2 and vertColor are declared both as input and output. the generator will use the same variables for them

here is the shader Node code ( Common/MatDefs/ShaderNodes/Common/commonVert.vert)

```
void main(){
 projPosition = worldViewProjectionMatrix * vec4(modelPosition,
}
```

As you can see all the inputs and outputs are not used. that's because most of them are attributes meant to be passed to the fragment shader as varyings. all the wiring will be handled by the generator only if those variables are used in an input or output mapping.

### ***CommonVert input mapping***

here we have the most basic yet mandatory thing in a vertex shader, computing vertex position in projection space. for this we have 2 mapping :

- **worldViewProjectionMatrix = WorldParam.WorldViewProjectionMatrix** : the input parameter worldViewProjectionMatrix is assigned with the WorldViewProjectionMatrix World parameter declared in the WorlParameters block of the technique.
- **modelPosition = Global.position.xyz** : the modelPosition (understand the vertex position in the model coordinate space) is assigned with the Global position variable.

As mentioned before Global position is initialized with the attribute inPosition, so this is equivalent to : modelPosition = Attr.inPosition.xyz

also note the swizzle of the Global.position variable. modelPosition is a vec3 and GlobalPosition is a vec4 so we just take the first 3 components.

### ***CommonVert output mapping***

- **Global.position = projPosition** : The result of the multiplication of the worldViewProjectionMatrix and the modelPosition is assigned to the Globale position

The Global.position variable will be assigned to the gl\_Position glsl built in output at the end of the shader.

### ***ColorMult Definition***

ColorMult is a very basic Shader Node that takes two colors as input and multiply them. here is the definition content (Common/MatDefs/ShaderNodes/Basic/ColorMult.j3sn) :

```

ShaderNodeDefinitions{
 ShaderNodeDefinition ColorMult {
 Type: Fragment
 Shader GLSL100: Common/MatDefs/ShaderNodes/Basic/colorMult.
 Documentation{
 Multiplies two colors
 @input color1 the first color
 @input color2 the second color
 @output outColor the resulting color
 }
 Input {
 vec4 color1
 vec4 color2
 }
 Output {
 vec4 outColor
 }
 }
}

```

here is the shader Node code (Common/MatDefs/ShaderNodes/Basic/colorMult.frag)

```

void main(){
 outColor = color1 * color2;
}

```

### ***ColorMult input mapping***

All inputs are mapped here :

- **color1 = MatParam.Color** : The first color is assigned to the Color Material parameter declared in the MaterialParameter block of the material definition
- **color2 = Global.color** : The second color is assigned with the Global color variable. this is defaulted to vec4(1.0) (opaque white). Note that in a much complex material def this variable could already have been assigned with a previous Shader Node output

### ***ColorMult output mapping***

- **Global.color = outColor** : the resulting color is assigned to the Global color variable.

Note that the Global.color variable will be assigned to gl\_FragColor (glsl < 1.5) or declared

as a Global output of the shader (glsl >= 1.5).

Also note that in case several Global variables are declared, the generator will assign them gl\_FragData[i](glsl < 1.5) i being the order the variable has been found in the material def. For glsl >= 1.5 the variable will just all be declared as shader output in the order they've been found in the declaration

### **Generated shader code**

Don't take this code as carved in stone, the generated code can change as optimization of the shader generator goes on

Vertex Shader (glsl 1.0)

```
uniform mat4 g_WorldViewProjectionMatrix;

attribute vec4 inPosition;

void main(){
 vec4 Global_position = inPosition;

 //CommonVert : Begin
 vec3 CommonVert_modelPosition = Global_position.xyz;
 vec4 CommonVert_projPosition;
 vec2 CommonVert_texCoord1;
 vec2 CommonVert_texCoord2;
 vec4 CommonVert_vertColor;

 CommonVert_projPosition = g_WorldViewProjectionMatrix * vec
 Global_position = CommonVert_projPosition;
 //CommonVert : End

 gl_Position = Global_position;
}
```

All materials parameter, world parameters, attributes varying are declared first. then for each shader node, the declarative part is appended.

For the main function, for each shader node, the input mappings are declared and assigned, the output are declared.

Then the variable names are replaced in the sahder node code with there complete name (NameSpace\_varName), material parameters are replaced in the shader code as is.

Then, the output are mapped.

As you can see `texCoord1/2` and `vertColor` are declared but never used. That's because the generator is not aware of that. By default it will declare all inputs in case they are used in the `shaderNode` code. Note that most glsl compiler will optimize this when compiling the shader on the GPU.

### Fragment Shader (glsl 1.0)

```
uniform vec4 m_Color;

void main(){
 vec4 Global_color = vec4(1.0);

 //ColorMult : Begin
 vec4 ColorMult_color2 = Global_color;
 vec4 ColorMult_outColor;

 ColorMult_outColor = m_Color * ColorMult_color2;
 Global_color = ColorMult_outColor;
 //ColorMult : End

 gl_FragColor = Global_color;
}
```

Same as for the Vertex shader. note that the `color1` is not declared, because it's directly replaced by the material parameter.

As a rule of thumb you should not assign a value to an input. Input are likely to be material parameters or outputs from other shaders and modifying them may cause unexpected behavior, even failure in your resulting shader.

For more explanations and design decisions please refer to the [spec](#) here

[https://docs.google.com/document/d/1S6xO3d1TBz0xcKe\\_MPTqY9V-QI59AKdg1OGy3U-HeVY/edit?usp=sharing](https://docs.google.com/document/d/1S6xO3d1TBz0xcKe_MPTqY9V-QI59AKdg1OGy3U-HeVY/edit?usp=sharing)

Thank you for the brave ones that came through all this reading. i'm not gonna offer you a prize in exchange of a password, because we ran out of JME thongs...

</div>

## **title: JME3 and Shaders**

# **JME3 and Shaders**

</div>

## **Shaders Basics**

Shaders are sets of instructions that are executed on the GPU. They are used to take advantage of hardware acceleration available on the GPU for rendering purposes.

This paper only covers Vertex and Fragment shaders because they are the only ones supported by JME3 for the moment. But be aware that there are some other types of shaders (geometry, tessellation,...).

There are multiple frequently used languages that you may encounter to code shaders but as JME3 is based on OpenGL, shaders in JME use GLSL and any example in this paper will be written in GLSL.

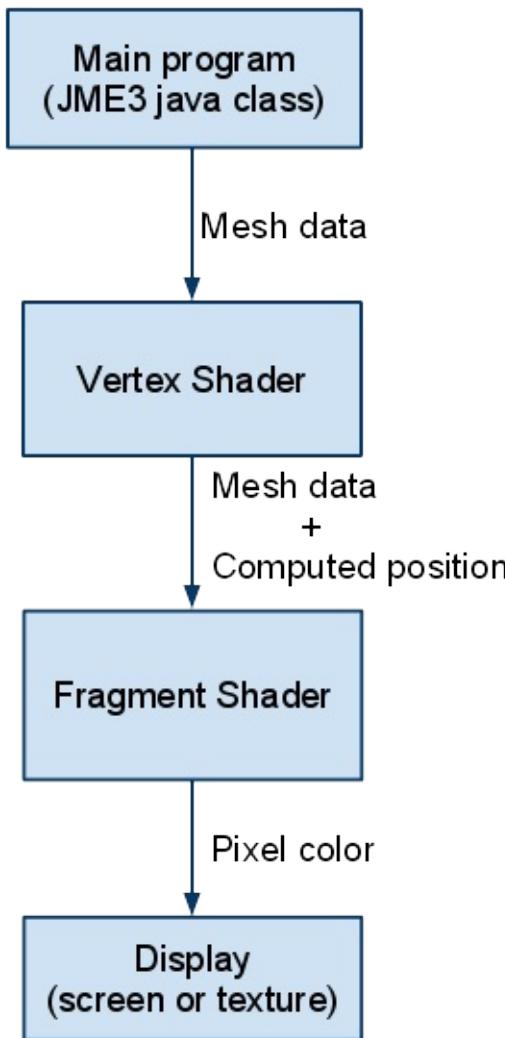
</div>

## **How Does it work?**

To keep it Simple: The Vertex shader is executed once for each vertex in the view, then the Fragment shader (also called the Pixel shader) is executed once for each pixel on the screen.

The main purpose of the Vertex shader is to compute the screen coordinate of a vertex (where this vertex will be displayed on screen) while the main purpose of the Fragment shader is to compute the color of a pixel.

This is a very simplified graphic to describe the call stack:



The main program sends mesh data to the vertex shader (vertex position in object space, normals, tangents, etc..). The vertex shader computes the screen position of the vertex and sends it to the Fragment shader. The fragment shader computes the color, and the result is displayed on screen or in a texture.

</div>

## Variables scope

There are different types of scope for variables in a shader :

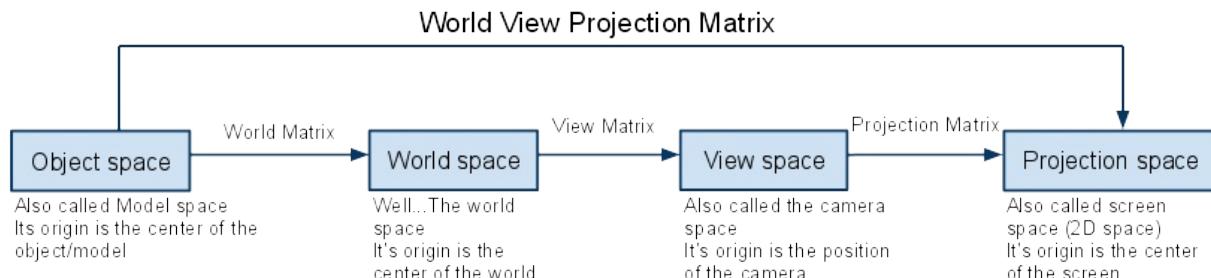
- uniform : User defined variables that are passed by the main program to the vertex and fragment shader, these variables are global for a given execution of a shader.
- attribute : Per-vertex variables passed by the engine to the shader, like position, normal, etc (Mesh data in the graphic)
- varying : Variables passed from the vertex shader to the fragment shader.

There is a large panel of variable types to be used, for more information about it I recommend reading the GLSL specification [here](#).

</div>

## Spaces and Matrices

To understand the coming example you must know about the different spaces in 3D computer graphics, and the matrices used to translate coordinate from one space to another.



The engine passes the object space coordinates to the vertex shader. We need to compute its position in projection space. To do that we transform the object space position by the WorldViewProjectionMatrix which is a combination of the World, View, Projection matrices (who would have guessed?).

</div>

## Simple example : rendering a solid color on an object

Here is the simplest application to shaders, rendering a solid color.

Vertex Shader :

```

//the global uniform World view projection matrix
//(more on global uniforms below)
uniform mat4 g_WorldViewProjectionMatrix;
//The attribute inPosition is the Object space position of the vert
attribute vec3 inPosition;
void main(){
 //Transformation of the object space coordinate to projection s
 //coordinates.
 // - gl_Position is the standard GLSL variable holding projectio
 //position. It must be filled in the vertex shader
 // - To convert position we multiply the worldViewProjectionMatr
 //by the position vector.
 //The multiplication must be done in this order.
 gl_Position = g_WorldViewProjectionMatrix * vec4(inPosition, 1.
}

```

Fragment Shader :

```

void main(){
 //returning the color of the pixel (here solid blue)
 // - gl_FragColor is the standard GLSL variable that holds the p
 //color. It must be filled in the Fragment Shader.
 gl_FragColor = vec4(0.0, 0.0, 1.0, 1.0);
}

```

For example applying this shader to a sphere would render a solid blue sphere on screen.

</div>

## How to use shaders in JME3

You probably heard that JME3 is “shader oriented”, but what does that mean?

Usually to use shaders you must create what is called a program. This program specify the vertex shader and the fragment shader to use.

JME3 encloses this in the material system. Every material in JME3 uses shaders.

For example let's have a look at the SolidColor.j3md file :

```

MaterialDef Solid Color {
 //This is the complete list of user defined uniforms to be used
 //in shaders
 MaterialParameters {
 Vector4 Color
 }
 Technique {
 //This is where the vertex and fragment shader files are
 //specified
 VertexShader GLSL100: Common/MatDefs/Misc/SolidColor.vert
 FragmentShader GLSL100: Common/MatDefs/Misc/SolidColor.frag
 //This is where you specify which global uniform you need for
 //shaders
 WorldParameters {
 WorldViewProjectionMatrix
 }
 }
 Technique FixedFunc {
 }
}

```

For more information on JME3 material system, i suggest you read this [topic](#).

</div>

## JME3 Global uniforms

JME3 can expose pre-computed global uniforms to your shaders. You must specify the ones that are required for your shader in the WorldParameters section of the material definition file (.j3md).

Note that in the shader the uniform names will be prefixed by a “g\_”.

In the example above, WorldViewProjectionMatrix is declared as uniform mat4 g\_WorldViewProjectionMatrix in the shader.

The complete list of global uniforms that can be used in JME3 can be found [here](#).

</div>

## JME3 Lighting Global uniforms

JME3 uses some global uniforms for lighting :

- g\_LightDirection (vec4) : the direction of the light
  - use for SpotLight : x,y,z contain the world direction vector of the light, the w component contains the spotlight angle cosine
- g\_LightColor (vec4) : the color of the light
- g\_LightPosition : the position of the light
  - use for SpotLight : x,y,z contain the world position of the light, the w component contains 1/lightRange
  - use for PointLight : x,y,z contain the world position of the light, the w component contains 1/lightRadius
  - use for DirectionalLight : strangely enough it's used for the direction of the light... this might change though. The fourth component contains -1 and it's used in the lighting shader to know if it's a directionalLight or not.
- g\_AmbientLightColor the color of the ambient light.

These uniforms are passed to the shader without having to declare them in the j3md file, but you have to specify in the technique definition “ LightMode MultiPass” see lighting.j3md for more information.

</div>

## JME3 attributes

Those are different attributes that are always passed to your shader.

You can find a complete list of those attribute in the Type enum of the VertexBuffer [here](#).

Note that in the shader the attributes names will be prefixed by an “in”.

When the enumeration lists some usual types for each attribute (for example texCoord specifies two floats) then that is the format expected by all standard JME3 shaders that use that attribute. When writing your own shaders though you can use alternative formats such as placing three floats in texCoord simply by declaring the attribute as vec3 in the shader and passing 3 as the component count into the mesh setBuffer call.

</div>

## User's uniforms

At some point when making your own shader you'll need to pass your own uniforms

Any uniform has to be declared in the material definition file (.j3md) in the

“MaterialParameters” section.

```
MaterialParameters {
 Vector4 Color
 Texture2D ColorMap
}
```

You can also pass some define to your vertex/fragment programs to know if an uniform has been declared.

You simply add it in the Defines section of your Technique in the definition file.

```
Defines {
 COLORMAP : ColorMap
}
```

For integer and floating point parameters, the define will contain the value that was set.

For all other types of parameters, the value 1 is defined.

If no value is set for that parameter, the define is not declared in the shader.

Those material parameters will be sent from the engine to the shader as follows, there are setXXXX methods for any type of uniform you want to pass.

```
material.setColor("Color", new ColorRGBA(1.0f, 0.0f, 0.0f, 1.0f)
material.setTexture("ColorMap", myTexture); // bind myTexture fo
```

To use this uniform in the shader, you need to declare it in the .frag or .vert files (depending on where you need it). You can make use of the defines here and later in the code: **Note that the “m\_” prefix specifies that the uniform is a material parameter.**

```
uniform vec4 m_Color;
#ifndef COLORMAP
 uniform sampler2D m_ColorMap;
#endif
```

The uniforms will be populated at runtime with the value you sent.

</div>

## Example: Adding Color Keying to the Lighting.j3md Material Definition

Color Keying is useful in games involving many players. It consists of adding some player-specific color on models textures.

The easiest way of doing this is to use a keyMap which will contain the amount of color to add in its alpha channel.

Here I will use this color map: <http://wstaw.org/m/2011/10/24/plasma-desktopxB2787.jpg> to blend color on this texture: <http://wstaw.org/m/2011/10/24/plasma-desktopbq2787.jpg>

We need to pass 2 new parameters to the Lighting.j3md definition, MaterialParameters section :

```
// Keying Map
Texture2D KeyMap

// Key Color
Color KeyColor
```

Below, add a new Define in the main Technique section:

```
KEYMAP : KeyMap
```

In the Lighting.frag file, define the new uniforms:

```
#ifdef KEYMAP
 uniform sampler2D m_KeyMap;
 uniform vec4 m_KeyColor;
#endif
```

Further, when obtaining the diffuseColor from the DiffuseMap texture, check if we need to blend it:

```
#ifdef KEYMAP
 vec4 keyColor = texture2D(m_KeyMap, newTexCoord);
 diffuseColor.rgb = (1.0-keyColor.a) * diffuseColor.rgb + keyC
#endif
```

This way, a transparent pixel in the KeyMap texture doesn't modify the color.

A black pixel replaces it for the m\_KeyColor and values in between are blended.

A result preview can be seen here: <http://wstaw.org/m/2011/10/24/plasma-desktoppuV2787.jpg>

</div>

## Step by step

- Create a vertex shader (.vert) file
- Create a fragment shader (.frag) file
- Create a material definition (j3md) file specifying the user defined uniforms, path to the shaders and the global uniforms to use
- In your initSimpleApplication, create a material using this definition, apply it to a geometry
- That's it!!

</ul>

```
// A cube
Box box= new Box(Vector3f.ZERO, 1f,1f,1f);
Geometry cube = new Geometry("box", box);
Material mat = new Material(assetManager,"Path/To/My/materialDef.j3md");
cube.setMaterial(mat);
rootNode.attachChild(cube);
```

</div>

## JME3 and OpenGL 3 & 4 compatibility

GLSL 1.0 to 1.2 comes with built in attributes and uniforms (ie, gl\_Vertex, gl\_ModelViewMatrix, etc...).

Those attributes are deprecated since GLSL 1.3 (opengl 3), hence JME3 global uniforms and attributes. Here is a list of deprecated attributes and their equivalent in JME3

GLSL 1.2 attributes	JME3 equivalent
gl_Vertex	inPosition
gl_Normal	inNormal
gl_Color	inColor
gl_MultiTexCoord0	inTexCoord
gl_ModelViewMatrix	g_WorldViewMatrix
gl_ProjectionMatrix	g_ProjectionMatrix
gl_ModelViewProjectionMatrix	g_WorldViewProjectionMatrix
gl_NormalMatrix	g_NormalMatrix

## Useful links

<http://www.eng.utah.edu/~cs5610/lectures/GLSL-ATI-Intro.pdf>

</div>

# **title: Gamma Correction or sRGB pipeline**

## **Gamma Correction or sRGB pipeline**

### **Overview**

Here is a quick overview of what lies under the “Gamma Correction” term.

More in depth rundowns on the matter can be found here :

[http://http.developer.nvidia.com/GPUGems3/gpugems3\\_ch24.html](http://http.developer.nvidia.com/GPUGems3/gpugems3_ch24.html) and here

<http://www.arcsynthesis.org/gltut/Texturing/Tutorial%2016.html>

We consider color values to be linear when computing lighting. What does that means? That means that we assume that the color 0.5,0.5,0.5 is half way between black and white.

The problem is that it's not the case, or at least not when you look at the color through a monitor.

CRT monitors had physical limitations that prevented them to have a linear way of representing colors. that means that 0.5,0.5,0.5 through a monitor is not half way between black and white (it's darker). Note that black and white remains the same though.

If we do not take that into account, the rendered images are overly darken and feels dull.

LCD monitors still mimic this physical limitation (I guess for backward compatibility).

#### **Output correct colors**

Gamma Correction is the technique that tends to correct the issue. Gamma is an power factor applied to the color by the monitor when lighting a pixel on screen (or at least a simplification of the function applied). So when we output the color, we have to apply the inverse power of this factor to nullify the effect : `finalColor = pow(computedColor, 1/gamma);`

#### **Knowing what colors we have as input**

The other aspect of gamma correction is the colors we get as input in the rendering process that are stored in textures or in ColorRGBA material params. Almost all image editors are storing color data in images already gamma corrected (so that colors are correct when you display the picture in a viewer or in a browser). Also most hand picked colors (in a color picker) can be assumed as gamma corrected as you most probably chose this color through a monitor display. Such images or color are said to be in sRGB color space, meaning

“standard RGB” (which is not the standard one would guess). That means that textures and colors that we use as input in our shaders are not in a linear space. The issue is that we need them in linear space when we compute the lighting, else the lighting is wrong. To avoid this we need to apply some gamma correction to the colors :  $(\text{pow}(\text{color}, \text{gamma}))$ ; This only apply to textures that will render colors on screen (basically diffuse map, specular, light maps). Normal maps, height maps don’t need the correction.

this is the kind of difference you can have :

left is non corrected output, right is gamma corrected output.



## Implementation

- To handle proper gamma corrected ouput colors, Opengl expose an ARB extension that allows you to output a color in linear space and have the GPU automatically correct it :  
[https://www.opengl.org/registry/specs/ARB/framebuffer\\_sRGB.txt](https://www.opengl.org/registry/specs/ARB/framebuffer_sRGB.txt)
- To handle the input, instead of classic RGBA8 image format, one can use SRGB8\_ALPHA8\_EXT which is basically RGBA in sRGB. Using this you specify the GPU that the texture is in sRGB space and when fetching a color from it, the GPU will linearize the color value for you (for free). There are sRGB equivalent to all 8 bits formats (even compressed format like DXT).
- But all textures don't need this. For example, normal maps, height maps colors are most probably generated and not hand picked by an artist looking through a monitor. The implementation needs to account for it and expose a way to exclude some textures from the sRGB pipeline.

Gamma Correction in jME 3.0 is based on those three statements.

Note that Gamma Correction is only available on desktop with LWJGL or JOGL renderer.

They are not yet supported on Android or iOS renderers

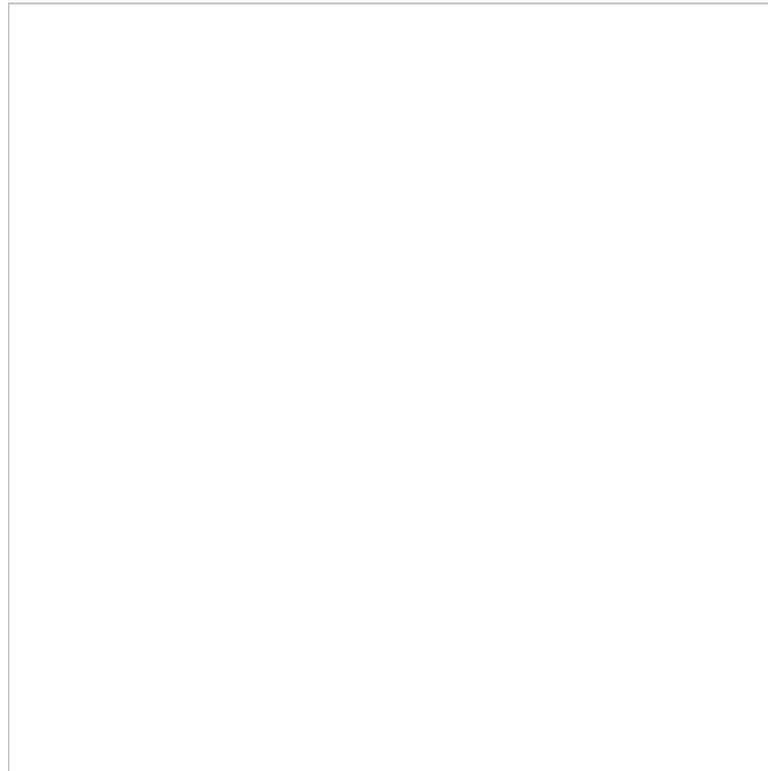
</div>

## Turning Gamma Correction on/off

You can turn Gamma Correction on and off using the AppSettings. There is a method `setGammaCorrection(boolean)` that changes the setting. use this in the `main()` method of your application :

```
AppSettings settings = new AppSettings(true);
settings.setGammaCorrection(true);
app.setSettings(settings);
```

This setting is also exposed in the Settings dialog displayed when you launch a jME application.



This is a short hand to enable both linearization of input textures and Gamma correction of the rendered output on screen. **But both can be enabled separately.**

</div>

### Enabling output Gamma Correction

You can enable or disable the Gamma correction of the rendered output by using

```
renderer.setMainFrameBufferSrgb(boolean srgb)
```

This will be ignored if the hardware doesn't have the GL\_ARB\_framebuffer\_sRGB or the GL\_EXT\_texture\_sRGB. This can be toggled at run time.

This uses Opengl hardware gamma correction that uses an approximated Gamma value of 2.2 and uses the following formula : color = pow(color,1/gamma)

Not that this will not yield exact results, as the real gamma can vary depending on the

monitor.

If this is a problem, please refer to the “handling gamma correction in a post process” section.

### Enabling texture linearization

You can enable or disable texture linearization by using

```
renderer.setLinearizeSrgbImages(boolean linearize)
```

This will be ignored if the hardware doesn't have the GL\_ARB\_framebuffer\_sRGB or the GL\_EXT\_texture\_sRGB.

Toggling this setting at runtime will produce unexpected behavior for now. A change in this setting would need a proper reload of the context to work.

All images marked as in sRGB color space will be uploaded to the GPU using a sRGB image format. Opengl hardware texture linearization also uses an approximated Gamma value of 2.2 and linearize the fetched texel color using the following formula :  $\text{color} = \text{pow}(\text{color}, \text{gamma})$

As with output gamma correction this will not give exact result, but the error is less important since most image editor uses the same approximation to correct images and save them in sRGB color space.

Not all image format have their sRGB equivalent, and only 8bit formats. Here is an exhaustive list of the supported format and there equivalent :

- RGB8 : GL\_SRGB8
- RGBA8 : GL\_SRGB8\_ALPHA8
- BGR8 : GL\_SRGB8
- ABGR8 : GL\_SRGB8\_ALPHA8
- Luminance8 : GL\_SLUMINANCE8
- Luminance8Alpha8 : GL\_SLUMINANCE8\_ALPHA8
- DXT1 : GL\_COMPRESSED\_SRGB\_S3TC\_DXT1
- DXT1A : GL\_COMPRESSED\_SRGB\_ALPHA\_S3TC\_DXT1
- DXT3 : GL\_COMPRESSED\_SRGB\_ALPHA\_S3TC\_DXT3
- DXT5 : GL\_COMPRESSED\_SRGB\_ALPHA\_S3TC\_DXT5

Conventionally only the rgb channels are gamma corrected, as the alpha channel does not represent a color value

## Excluding images from the sRGB pipeline

Only loaded images will be marked as in sRGB color space, when using

`assetManager.loadTexture` or `loadAsset`.

The color space of an image created by code will have to be specified in the constructor or will be assumed as Linear if not specified.

Not all images need to be linearized. Some images don't represent color information that will be displayed on screen, but more different sort of data packed in a texture.

The best example is a Normal map that will contain normal vectors for each pixel. Height maps will contain elevation values. These textures must not be linearized.

There is no way to determine the real color space of an image when loading it, so we must deduce the color space from the usage it's loaded for. The usage is dictated by the material, those textures are used for, and by the material parameter they are assigned to. One can now specify in a material definition file (`j3md`) if a texture parameter must be assumed as in linear color space, and thus, must not be linearized, by using the keyword `-LINEAR` next to the parameter (case does not matter).

For example here is how the `NormalMap` parameter is declared in the lighting material definition.

```
// Normal map
Texture2D NormalMap -LINEAR
```

When a texture is assigned to this material param by using `material.setTexture("NormalMap", myNormalTexture)`, the color space of this texture's image will be forced to linear. So if you make your own material and want to use Gamma Correction, make sure you properly mark your textures as in the proper color space.

This can sound complicated, but you just have to answer this question : Does my image represent color data? if the answer is no, then you have to set the `-Linear` flag.

</div>

## ColorRGBA as sRGB

The `r`, `g`, `b` attributes of a `ColorRGBA` object are **ALWAYS** assumed in Linear color space. If you want to set a color that you hand picked in a color picker, you should use the `setAsSRGB` method of `ColorRGBA`. This will convert the given values to linear color space by using the same formula as before : `color = pow (color, gamma)` where `gamma = 2.2`;

If you want to retrieve those values from a `ColorRGBA`, you can call the `getAsSRGB` method. The values will be converted back to sRGB color Space.

Note that the return type of that method is a `Vector4f` and not a `ColorRGBA`, because as stated before, all `ColorRGBA` objects `r,g,b` attributes are assumed in Linear color space.

</div>

## Handling rendered output Gamma Correction with a post process filter

As stated before, the hardware gamma correction uses an approximated gamma value of 2.2. Some may not be satisfied with that approximation and may want to pick a more appropriate gamma value. You can see in some games some Gamma calibration screens, that are here to help the player pick a correct gamma value for the monitor he's using.

For this particular case, you can do as follows :

1. Enable Gamma Correction global app setting.
2. Disable rendered output correction : `renderer.setMainFrameBufferSrgb(false);` (for example in the `simpleInit` method of your `SimpleApplication`).
3. Use the `GammaCorrectionFilter` in a `FilterPostProcessor`, and set the proper gamma value on it (default is 2.2).

## Should you use this?

Yes. Mostly because it's the only way to have proper lighting. If you're starting a new project it's a no brainer...use it, period. And don't allow the player to turn it off.

Now if you already spent time to adjust lighting in your scenes, without gamma correction, turning it on will make everything too bright, and you'll have to adjust all your lighting and colors again. That's why we kept a way to turn it off, for backward compatibility.

</div>

## **title: Level of Detail (LOD) Optimization**

# **Level of Detail (LOD) Optimization**

A mesh with a high level of detail has lots of polygons and looks good close up. But when the mesh is further away (and the detail is not visible), the high-polygon count slows down performance unnecessarily.

One solution for this problem is to use high-detail meshes for objects close to the camera, and low-detail meshes for objects far from the camera. As the player moves through the scene, you must keep replacing close objects by more detailed meshes, and far objects by less detailed meshes. The goal is to keep few high-quality slow-rendering objects in the foreground, and many low-quality fast-rendering objects in the background. (Experienced users can compare this approach to [JME's TerraMonkey terrain system](#), which internally uses the specialized GeoMipMapping algorithm to generate a terrain's Levels of Detail.)

You see now why you may want to be able to generate Levels of Detail for complex Geometries automatically. JME3 supports a Java implementation of the Ogre engine's LOD generator (originally by Péter Szücs and Stan Melax): You use [jme3tools.optimize.LodGenerator](#) in conjunction with [com.jme3.scene.control.LodControl](#).

For a demo, run [TestLodGeneration.java](#) from [JmeTests](#), then press +/- and spacebar to experiment. The following screenshots show a monkey model with three reduced Levels of



## Usage

To activate this optimization:

1. Pick a reduction method and values for the Geometry. (Trial and error...)
2. Generate LODs for the Geometry, either in the SDK or in code.
3. Add an LOD control to the Geometry.

## Pick Reduction Methods and Values

There are several reduction methods to generate a low-polygon version from a high-polygon model. Don't worry, the reduction does not modify the original model.

Reduction Method	Description	Reduction Value
LodGenerator.TriangleReductionMethod.COLLAPSE_COST	Collapses polygon vertices from the mesh until the reduction cost (= amount of ugly artifacts caused) exceeds the given threshold.	0.0f - 1.0f
LodGenerator.TriangleReductionMethod.PROPORTIONAL	Removes the given percentage of polygons from the mesh.	0.0f - 1.0f
LodGenerator.TriangleReductionMethod.CONSTANT	Removes the given number of polygons from the mesh.	integer

If you don't know which to choose, experiment. For example start by trying COLLAPSE\_COST and .5f-.9f.

## Generate LOD

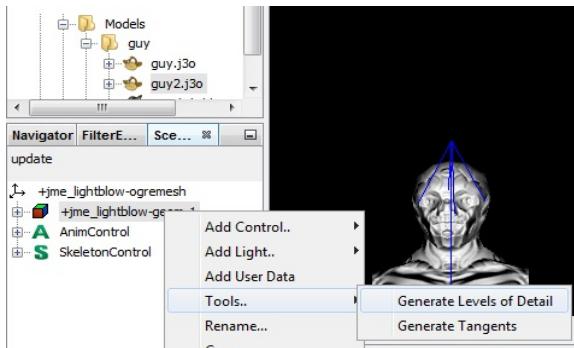
You must generate and cache several LODs for each mesh, ranging from many to few polygons. The LOD generator algorithm attempts to collapse vertices automatically, while avoiding ugly artifacts. The LOD generator doesn't generate new meshes, it only creates separate reduced index buffers for the more highly reduced levels.

- If you create geometries manually (3D models), use the SDK to generate LODs.
- If you create geometries programmatically, generate LODs from your Java code.

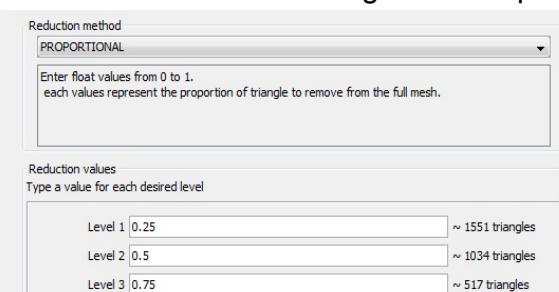
## Generating LODs in the SDK

The SDK contains a user-friendly interface to generate LODs for a model (.j3o file).

1. Open the Projects or Files window.
2. Select the .j3o file in the Project Assets > Models directory.
3. Choose Window>SceneExplorer if the SceneExplorer is not open. Info about the selected model is now displayed in the SceneExplorer.
4. Right-click the model in SceneExplorer. Choose the “Tools > Generate Levels of Detail” menu.



5. The “Generate LOD” settings wizard opens:



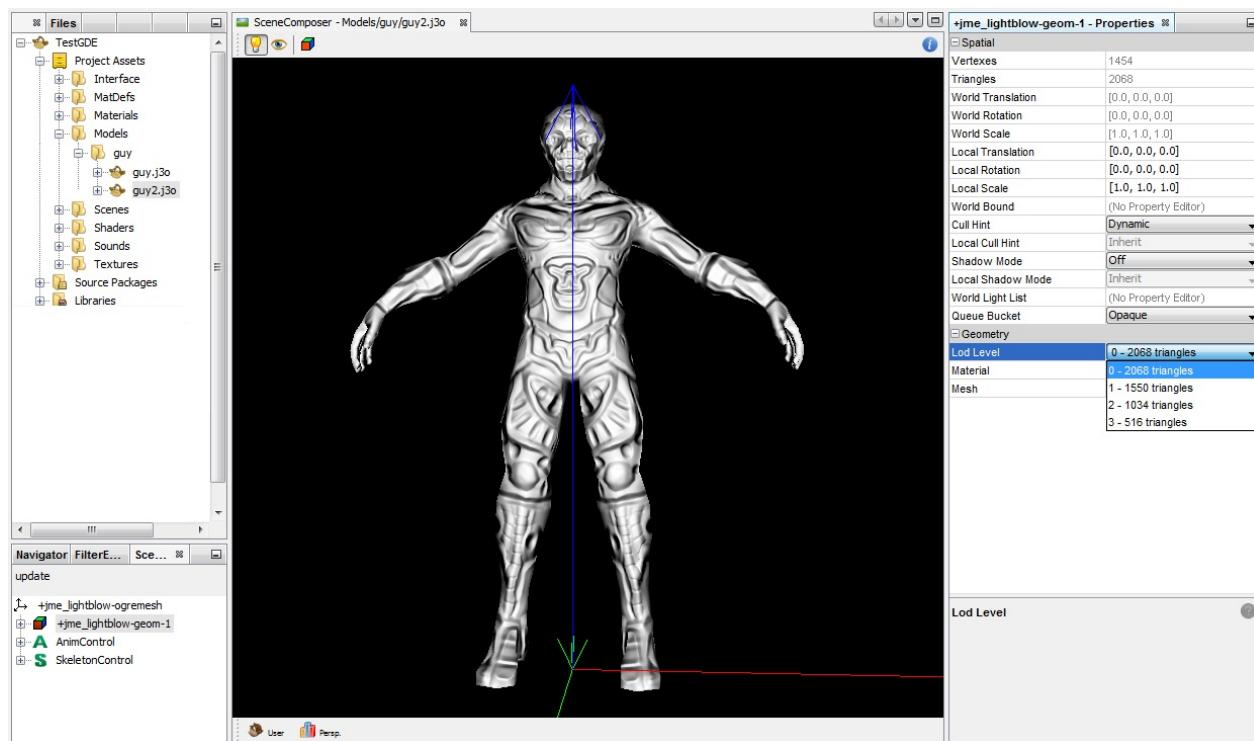
6. Choose a reduction method and reduction values for one or more levels.

Tip: Enter higher reduction values for higher levels.

7. Click Finish to generate the LODs for this model.

The LODs are saved in the .j3o model file.

Choose Window>Properties if the Properties window is not open. Choose the generated LODs from the dropdown in the Properties window, and verify their quality in the SceneComposer.



&lt;/div&gt;

## Generating LODs in Code

The `jme3tools.optimize.LodGenerator` utility class helps you generate LODs for an arbitrary mesh (a Geometry object) programmatically from your Java code. You create and bake one LodGenerator for each Geometry.

```
LodGenerator lod = new LodGenerator(geometry);
lod.bakeLods(reductionMethod, reductionValue);
```

The LODs are stored inside the Geometry object.

**Example:** How to generate an LOD of myPrettyGeo's mesh with 50% fewer polygons:

```
LodGenerator lod = new LodGenerator(myPrettyGeo);
lod.bakeLods(LodGenerator.TriangleReductionMethod.PROPORTIONAL, 0.5f)
```

## Activate the LOD Control

After generating the LODs for the geometry, you create and add a `com.jme3.scene.control.LodControl` to the geometry. Adding the LodControl activates the LOD optimization for this geometry.

```
LodControl lc = new LodControl();
myPrettyGeo.addControl(lc);
rootNode.attachChild(myPrettyGeo);
```

The LodControl internally gets the camera from the game's viewport to calculate the distance to this geometry. Depending on the distance, the LodControl selects an appropriate level of detail, and passes more (or less) detailed vertex data to the renderer.

</div>

## Impact on Quality and Speed

Level number	Purpose	Distance	Rendering Speed	Rendering Quality
“Level 0”	The original mesh is used automatically for close-ups, and it's the default if no LODs have been generated.	Closest	Slowest.	Best.
“Level 1” “Level 2” “Level 3” ...	If you generated LODs, JME3 uses them automatically as soon as the object moves into the background.	The higher the level, the further away.	The higher the level, the faster.	The higher the level, the lower the quality.

</div>

## See also

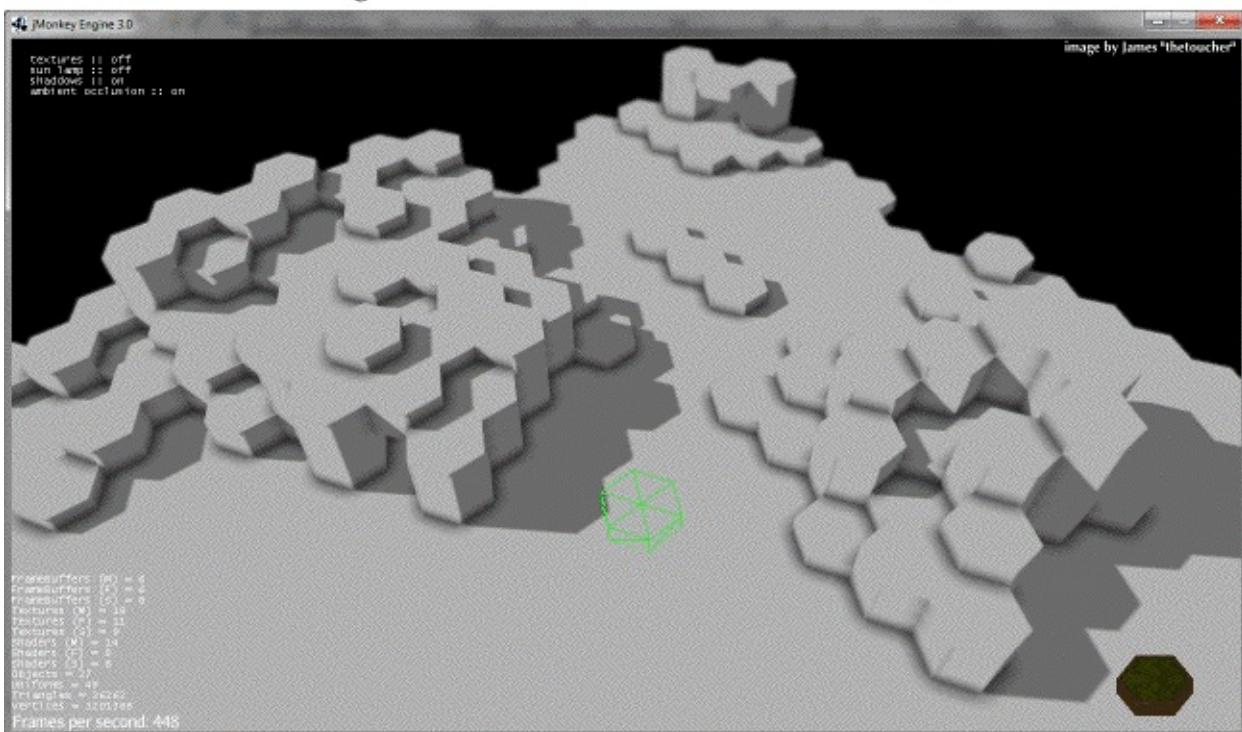
- <http://hub.jmonkeyengine.org/forum/topic/brand-new-lod-generator/>
- <https://github.com/worldforge/ember/tree/master/src/components/ogre/lod>
- <http://www.melax.com/polychop>
- <http://sajty.elementfx.com/progressivemesh/GSoC2012.pdf>
- [JME3 TerraMonkey Terrain](#)

</div>

# **title: Light and Shadow**

# **Light and Shadow**

### Colorized +Ambient Light



#### + Shadows + Ambient Occlusion

Light and Shadow are two separate things in 3D engines, although we perceive them together in real life:

- Lighting means that an object is brighter on the side facing the light direction, and darker on the backside. Computationally, this is relatively easy.
  - Lighting does not mean that objects cast a shadow on the floor or other objects: Activating shadow processing is an additional step described here. Since casting shadows has an impact on performance, drop shadows and ambient occlusion shading are not activated by default.

A light source with a direction or location is required for all Geometries with Lighting.j3md-based Materials. An ambient light is not sufficient. In a scene with no appropriate light sources, Geometries with Lighting.j3md-based Materials do not render. Only Geometries with Unshaded.j3md-based Materials are visible independent of any light sources.

</div>

# Light Sources and Colors

A lit scene with multiple light sources

You can add several types of light sources to a scene using `rootNode.addLight(mylight)`.

The available light sources in `com.jme3.light` are:

- `SpotLight`
- `PointLight`
- `AmbientLight`
- `DirectionalLight`

You control the color and intensity of each light source. Typically you set the color to white (`new ColorRGBA(1.0f, 1.0f, 1.0f, 1.0f)` or `ColorRGBA.White`), which makes all scene elements appear in their natural color.

You can choose to use lights in other colors than white, or darker colors. This influences the scene's atmosphere and will make the scene appear colder (e.g. `ColorRGBA.Cyan`) or warmer (`ColorRGBA.Yellow`), brighter (higher values) or darker (lower values).

You can get a list of all lights added to a Spatial by calling `getWorldLightList()` (includes inherited lights) or `getLocalLightList()` (only directly added lights), and iterating over the result.

## PointLight



A PointLight has a location and shines from there in all directions as far as its radius reaches. The light intensity decreases with increased distance from the light source. A PointLight can be used to cast shadows along with a PointLightShadowRenderer (see the Casting Shadows section)

**Typical example:** Lamp, lightbulb, torch, candle.

```
PointLight lamp_light = new PointLight();
lamp_light.setColor(ColorRGBA.Yellow);
lamp_light.setRadius(4f);
lamp_light.setPosition(new Vector3f(lamp_geo.getLocalTranslation())
rootNode.addLight(lamp_light);
```

## DirectionalLight



A Directionallight has no position, only a direction. It sends out parallel beams of light and is considered “infinitely” far away. You typically have one directional light per scene. A DirectionalLight can be used together with shadows.

**Typically example:** Sun light.

```
DirectionalLight sun = new DirectionalLight();
sun.setColor(ColorRGBA.White);
sun.setDirection(new Vector3f(-.5f, -.5f, -.5f).normalizeLocal());
rootNode.addLight(sun);
```

## SpotLight



A SpotLight sends out a distinct beam or cone of light. A SpotLight has a direction, a position, distance (range) and two angles. The inner angle is the central maximum of the light cone, the outer angle the edge of the light cone. Everything outside the light cone's angles is not affected by the light.

### Typical Example: Flashlight

```
SpotLight spot = new SpotLight();
spot.setSpotRange(100f); // distance
spot.setSpotInnerAngle(15f * FastMath.DEG_TO_RAD); // inner light c
spot.setSpotOuterAngle(35f * FastMath.DEG_TO_RAD); // outer light c
spot.setColor(ColorRGBA.White.mult(1.3f)); // light color
spot.setPosition(cam.getLocation()); // shine from ca
spot.setDirection(cam.getDirection()); // shine forward
rootNode.addLight(spot);
```

If you want the spotlight to follow the flycam, repeat the `setDirection(...)` and `setPosition(...)` calls in the update loop, and keep syncing them with the camera position and direction.

## AmbientLight

An AmbientLight simply influences the brightness and color of the scene globally. It has no direction and no location and shines equally everywhere. An AmbientLight does not cast any shadows, and it lights all sides of Geometries evenly, which makes 3D objects look unnaturally flat; this is why you typically do not use an AmbientLight alone without one of the other lights.

**Typical example:** Regulate overall brightness, tinge the whole scene in a warm or cold color.

```
AmbientLight al = new AmbientLight();
al.setColor(ColorRGBA.White.mult(1.3f));
rootNode.addLight(al);
```

You can increase the brightness of a light source gradually by multiplying the light color to values greater than 1.0f.

Example: `mylight.setColor(ColorRGBA.White.mult(1.3f));`  
`</div>`

## Light Follows Spatial

You can use a `com.jme3.scene.control.LightControl` to make a SpotLight or PointLight follow a Spatial. This can be used for a flashlight being carried by a character, or for car headlights, or an aircraft's spotlight, etc.

```
PointLight myLight = new PointLight();
rootNode.addLight(myLight);
LightControl lightControl = new LightControl(myLight);
spatial.addControl(lightControl); // this spatial controls the posi
```

Obviously, this does not apply to AmbientLights, which have no position.

`</div>`

## BasicShadowRenderer (deprecated)

Full code sample

- [TestShadow.java](#)

## Casting Shadows

For each type of non-ambient light source, JME3 implements two ways to simulate geometries casting shadows on other geometries:

- a shadow renderer (which you apply to a viewport) and
- a shadow filter (which you can add to a viewport's filter post-processor).

<b>light source class</b>	<b>shadow renderer class</b>	<b>shadow filter class</b>
DirectionalLight	DirectionalLightShadowRenderer	DirectionalLightShadowFilter
PointLight	PointLightShadowRenderer	PointLightShadowFilter
SpotLight	SpotLightShadowRenderer	SpotLightShadowFilter
AmbientLight	(not applicable)	(not applicable)

You only need one shadow simulation per light source: if you use shadow rendering, you won't need a shadow filter and vice versa. Which way is more efficient depends partly on the complexity of your scene. All six shadow simulation classes have similar interfaces, so once you know how to use one, you can easily figure out the rest.

Shadow calculations (cast and receive) have a performance impact, so use them sparingly. With shadow renderers, you can turn off shadow casting and/or shadow receiving for individual geometries, for portions of the scene graph, or for the entire scene:

```

spatial.setShadowMode(ShadowMode.Inherit); // This is the default
rootNode.setShadowMode(ShadowMode.Off); // Disable shadows for the entire scene
wall.setShadowMode(ShadowMode.CastAndReceive); // The wall can cast and receive shadows
floor.setShadowMode(ShadowMode.Receive); // Any shadows cast by the wall will be received by the floor
airplane.setShadowMode(ShadowMode.Cast); // There's nothing to receive shadows from
ghost.setShadowMode(ShadowMode.Off); // The ghost is transparent and doesn't cast or receive shadows

```

Both shadow renderers and shadow filters use shadow modes to determine which objects can cast shadows. However, only the shadow renderers pay attention to shadow modes when determining which objects receive shadows. With a shadow filter, shadow modes have no effect on which objects receive shadows.

Here's a sample application which demonstrates both DirectionalLightShadowRenderer and DirectionalLightShadowFilter:

- [TestDirectionalLightShadow.java](#)

Here is the key code fragment:

```

 DirectionalLight sun = new DirectionalLight();
 sun.setColor(ColorRGBA.White);
 sun.setDirection(cam.getDirection());
 rootNode.addLight(sun);

 /* Drop shadows */
 final int SHADOWMAP_SIZE=1024;
 DirectionalLightShadowRenderer dlsr = new DirectionalLightShadowRenderer(SHADOWMAP_SIZE);
 dlsr.setLight(sun);
 viewPort.addProcessor(dlsr);

 DirectionalLightShadowFilter dlsf = new DirectionalLightShadowFilter(SHADOWMAP_SIZE);
 dlsf.setLight(sun);
 dlsf.setEnabled(true);
 FilterPostProcessor fpp = new FilterPostProcessor(assetManager);
 fpp.addFilter(dlsf);
 viewPort.addProcessor(fpp);
 }
}

```

Constructor arguments:

- \* your AssetManager object
- \* size of the rendered shadow maps, in pixels per side (512, 1024, 2048, etc...)
- \* the number of shadow maps rendered (more shadow maps = better quality, but slower)

Properties you can set:

- \* `setDirection(Vector3f)` – the direction of the light
- \* `setLambda(0.65f)` – to reduce the split size
- \* `setShadowIntensity(0.7f)` – shadow darkness (1=black, 0=invisible)
- \* `setShadowZExtend(float)` – distance from camera to which shadows will be computed

</div>

## Parallel-Split Shadow Map (deprecated)

Full sample code

- [TestPssmShadow.java](#)

### A lit scene with PSSM drop shadows

```
private PssmShadowRenderer pssmRenderer;
...
public void simpleInitApp() {
 ...
 pssmRenderer = new PssmShadowRenderer(assetManager, 1024, 3);
 pssmRenderer.setDirection(new Vector3f(-.5f, -.5f, -.5f).normalize());
 viewPort.addProcessor(pssmRenderer);
```

## Screen Space Ambient Occlusion

Full sample code

- [jme3/src/test/jme3test/post/TestSSAO.java](#) – Screen-Space Ambient Occlusion shadows
- [jme3/src/test/jme3test/post/TestTransparentSSAO.java](#) – Screen-Space Ambient Occlusion shadows plus transparency
- [Screen Space Ambient Occlusion for jMonkeyEngine \(article\)](#)

Ambient Occlusion refers to the shadows which nearby objects cast on each other under an ambient lighting. Screen Space Ambient Occlusion (SSAO) approximates how light radiates in real life.

In JME3, SSAO is implemented by adding an instance of `com.jme3.post.SSAOFilter` to a viewport which already simulates shadows using another method such as `DirectionalLightShadowRenderer`.

```
FilterPostProcessor fpp = new FilterPostProcessor(assetManager);
SSAOFilter ssaoFilter = new SSAOFilter(12.94f, 43.92f, 0.33f, 0.61f
fpp.addFilter(ssaoFilter);
viewPort.addProcessor(fpp);
```



&lt;/div&gt;

## **title: Nifty Loading Screen (Progress Bar)**

### **Nifty Loading Screen (Progress Bar)**

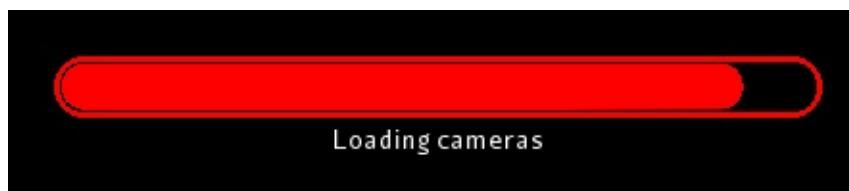
There is a good tutorial about creating a nifty progress bar here:

[http://sourceforge.net/apps/mediawiki/nifty-gui/index.php?  
title>Create\\_your\\_own\\_Control\\_%28A\\_Nifty\\_Progressbar%29](http://sourceforge.net/apps/mediawiki/nifty-gui/index.php?title/Create_your_own_Control_%28A_Nifty_Progressbar%29)

This example will use the existing hello terrain as an example. It will require these 2 images inside Assets/Interface/ (save them as border.png and inner.png respectively)



This is the progress bar at 90%:



nifty\_loading.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<nifty>
 <useStyles filename="nifty-default-styles.xml" />
 <useControls filename="nifty-default-controls.xml" />

 <controlDefinition name = "loadingbar" controller = "jme3test.TestLoadingScreen">
 <image filename="Interface/border.png" childLayout="absolute" imageMode="resize:15,2,15,15,15,2,15,2,15,2,15,15">
 <image id="progressbar" x="0" y="0" filename="Interface/loadingbar.png" imageMode="resize:15,2,15,15,15,2,15,2,15,2,15,15">
 </image>
 </image>
 </controlDefinition>

 <screen id="start" controller = "jme3test.TestLoadingScreen">
 <layer id="layer" childLayout="center">
 <panel id = "panel2" height="30%" width="50%" align="center" visibleToMouse="true">
 <control id="startGame" name="button" backgroundColor="white" >
 <interact onClick="showLoadingMenu()" />
 </control>
 </panel>
 </layer>
 </screen>

 <screen id="loadlevel" controller = "jme3test.TestLoadingScreen">
 <layer id="loadinglayer" childLayout="center" backgroundColor="white">
 <panel id = "loadingpanel" childLayout="vertical" align="center" valign="middle">
 <control name="loadingbar" align="center" valign="middle" >
 <control id="loadingtext" name="label" align="center" text="Loading..." >
 </panel>
 </layer>
 </screen>

 <screen id="end" controller = "jme3test.TestLoadingScreen">
 </screen>
</nifty>
```

## Understanding Nifty XML

The progress bar and text is done statically using nifty XML. A custom control is created, which represents the progress bar.

```
<controlDefinition name = "loadingbar" controller = "jme3test.TestLoadingScreen">
 <image filename="Interface/border.png" childLayout="absolute"
 imageMode="resize:15,2,15,15,15,2,15,2,15,2,15,15">
 <image id="progressbar" x="0" y="0" filename="Interface/progressbar.png"
 imageMode="resize:15,2,15,15,15,2,15,2,15,2,15,15">
 </image>
</controlDefinition>
```



This screen simply displays a button in the middle of the screen, which could be seen as a simple main menu UI.

```
<screen id="start" controller = "jme3test.TestLoadingScreen">
 <layer id="layer" childLayout="center">
 <panel id = "panel2" height="30%" width="50%" align="center"
 visibleToMouse="true">
 <control id="startGame" name="button" backgroundColor="white">
 <interact onClick="showLoadingMenu()" />
 </control>
 </panel>
 </layer>
</screen>
```



This screen displays our custom progress bar control with a text control

```

<screen id="loadlevel" controller = "jme3test.TestLoadingScreen">
 <layer id="loadinglayer" childLayout="center" backgroundCol="white">
 <panel id = "loadingpanel" childLayout="vertical" align="center" valign="center">
 <control name="loadingbar" align="center" valign="center" type="progress">
 <control id="loadingtext" name="label" align="center" valign="center" text="Loading..." type="text">
 </control>
 </panel>
 </layer>
</screen>

```



## Creating the bindings to use the Nifty XML

There are 3 main ways to update a progress bar. To understand why these methods are necessary, an understanding of the graphics pipeline is needed.

Something like this in a single thread will not work:

```

load_scene();
update_bar(30%);
load_characters();
update_bar(60%);
load_sounds();
update_bar(100%);

```

If you do all of this in a single frame, then it is sent to the graphics card only after the whole code block has executed. By this time the bar has reached 100% and the game has already begun – for the user, the progressbar on the screen would not have visibly changed.

The 2 main good solutions are:

1. Updating explicitly over many frames
2. Multi-threading

## Updating progress bar over a number of frames

The idea is to break down the loading of the game into discrete parts

```

package jme3test;

import com.jme3.niftygui.NiftyJmeDisplay;

```

```
import de.lessvoid.nifty;
import de.lessvoid.nifty.elements.Element;
import de.lessvoid.nifty.input.NiftyInputEvent;
import de.lessvoid.nifty.screen.Screen;
import de.lessvoid.nifty.screen.ScreenController;
import de.lessvoid.nifty.tools.SizeValue;
import com.jme3.app.SimpleApplication;
import com.jme3.material.Material;
import com.jme3.renderer.Camera;
import com.jme3.terrain.geomipmap.TerrainLodControl;
import com.jme3.terrain.heightmap.AbstractHeightMap;
import com.jme3.terrain.geomipmap.TerrainQuad;
import com.jme3.terrain.heightmap.ImageBasedHeightMap;
import com.jme3.texture.Texture;
import com.jme3.texture.Texture.WrapMode;
import de.lessvoid.nifty.controls.Controller;
import de.lessvoid.nifty.elements.render.TextRenderer;
import de.lessvoid.xml.xpp3.Attributes;
import java.util.ArrayList;
import java.util.List;
import java.util.Properties;
import jme3tools.converters.ImageToAwt;

public class TestLoadingScreen extends SimpleApplication implements

 private NiftyJmeDisplay niftyDisplay;
 private Nifty nifty;
 private Element progressBarElement;
 private TerrainQuad terrain;
 private Material mat_terrain;
 private float frameCount = 0;
 private boolean load = false;
 private TextRenderer textRenderer;

 public static void main(String[] args) {
 TestLoadingScreen app = new TestLoadingScreen();
 app.start();
 }

 @Override
```

```
public void simpleInitApp() {
 flyCam.setEnabled(false);
 niftyDisplay = new NiftyJmeDisplay(assetManager,
 inputManager,
 audioRenderer,
 guiViewPort);
 nifty = niftyDisplay.getNifty();

 nifty.fromXml("Interface/nifty_loading.xml", "start", this)

 guiViewPort.addProcessor(niftyDisplay);
}

@Override
public void simpleUpdate(float tpf) {

 if (load) { //loading is done over many frames
 if (frameCount == 1) {
 Element element = nifty.getScreen("loadlevel").findElement("text");
 textRenderer = element.getRenderer(TextRenderer.class);

 mat_terrain = new Material(assetManager, "Common/MatDefs/Light/Lighting.j3md");
 mat_terrain.setTexture("Alpha", assetManager.loadTexture("Textures/Grass.png"));
 setProgress(0.2f, "Loading grass");

 } else if (frameCount == 2) {
 Texture grass = assetManager.loadTexture("Textures/Grass.png");
 grass.setWrap(WrapMode.Repeat);
 mat_terrain.setTexture("Tex1", grass);
 mat_terrain.setFloat("Tex1Scale", 64f);
 setProgress(0.4f, "Loading dirt");

 } else if (frameCount == 3) {
 Texture dirt = assetManager.loadTexture("Textures/Dirt.png");
 dirt.setWrap(WrapMode.Repeat);
 mat_terrain.setTexture("Tex2", dirt);
 mat_terrain.setFloat("Tex2Scale", 32f);
 setProgress(0.5f, "Loading rocks");
 }
 }
}
```

```
 } else if (frameCount == 4) {
 Texture rock = assetManager.loadTexture("Textures/1"
 rock.setWrap(WrapMode.Repeat);

 mat_terrain.setTexture("Tex3", rock);
 mat_terrain.setFloat("Tex3Scale", 128f);
 setProgress(0.6f, "Creating terrain");

 } else if (frameCount == 5) {
 AbstractHeightMap heightmap = null;
 Texture heightMapImage = assetManager.loadTexture("heightmap =
 new ImageBasedHeightMap(heightMapImage.

 heightmap.load();
 terrain = new TerrainQuad("my terrain", 65, 513, he
 setProgress(0.8f, "Positioning terrain");

 } else if (frameCount == 6) {
 terrain.setMaterial(mat_terrain);

 terrain.setLocalTranslation(0, -100, 0);
 terrain.setLocalScale(2f, 1f, 2f);
 rootNode.attachChild(terrain);
 setProgress(0.9f, "Loading cameras");

 } else if (frameCount == 7) {
 List<Camera> cameras = new ArrayList<Camera>();
 cameras.add(getCamera());
 TerrainLodControl control = new TerrainLodControl(t
 terrain.addControl(control);
 setProgress(1f, "Loading complete");

 } else if (frameCount == 8) {
 nifty.gotoScreen("end");
 nifty.exit();
 guiViewPort.removeProcessor(niftyDisplay);
 flyCam.setEnabled(true);
 flyCam.setMoveSpeed(50);
 }
 }
```

```
 frameCount++;
 }

}

public void setProgress(final float progress, String loadingText) {
 final int MIN_WIDTH = 32;
 int pixelWidth = (int) (MIN_WIDTH + (progressBarElement.getConstraintWidth() - progressBarElement.getConstraintWidth(new SizeValue(pixelWidth, 16))) * progress);
 progressBarElement.setConstraintWidth(new SizeValue(pixelWidth, 16));
 progressBarElement.getParent().layoutElements();

 textRenderer.setText(loadingText);
}

public void showLoadingMenu() {
 nifty.gotoScreen("loadlevel");
 load = true;
}

@Override
public void onStartScreen() {
}

@Override
public void onEndScreen() {
}

@Override
public void bind(Nifty nifty, Screen screen) {
 progressBarElement = nifty.getScreen("loadlevel").findElement("progressBar");
}

// methods for Controller
@Override
public boolean inputEvent(final NiftyInputEvent inputEvent) {
 return false;
}

@Override
public void bind(Nifty nifty, Screen screen, Element elmnt, Proc
```

```

 progressBarElement = elmnt.findElementByName("progressbar")
 }

@Override
public void init(Properties prprts, Attributes atrbts) {
}

public void onFocus(boolean getFocus) {
}
}

```

Note:

- Try and add all controls near the end, as their update loops may begin executing

## Using multithreading

For more info on multithreading: [The jME3 Threading Model](#)

Make sure to change the XML file to point the controller to TestLoadingScreen1

```

package jme3test;

import com.jme3.niftygui.NiftyJmeDisplay;
import de.lessvoid.nifty.Nifty;
import de.lessvoid.nifty.elements.Element;
import de.lessvoid.nifty.input.NiftyInputEvent;
import de.lessvoid.nifty.screen.Screen;
import de.lessvoid.nifty.screen.ScreenController;
import de.lessvoid.nifty.tools.SizeValue;
import com.jme3.app.SimpleApplication;
import com.jme3.material.Material;
import com.jme3.renderer.Camera;
import com.jme3.terrain.geomipmap.TerrainLodControl;
import com.jme3.terrain.heightmap.AbstractHeightMap;
import com.jme3.terrain.geomipmap.TerrainQuad;
import com.jme3.terrain.heightmap.ImageBasedHeightMap;
import com.jme3.texture.Texture;
import com.jme3.texture.Texture.WrapMode;
import de.lessvoid.nifty.controls.Controller;

```

```
import de.lessvoid.nifty.elements.render.TextRenderer;
import de.lessvoid.xml.xpp3.Attributes;
import java.util.ArrayList;
import java.util.List;
import java.util.Properties;
import java.util.concurrent.Callable;
import java.util.concurrent.Future;
import java.util.concurrent.ScheduledThreadPoolExecutor;
import jme3tools.converters.ImageToAwt;

public class TestLoadingScreen1 extends SimpleApplication implements

 private NiftyJmeDisplay niftyDisplay;
 private Nifty nifty;
 private Element progressBarElement;
 private TerrainQuad terrain;
 private Material mat_terrain;
 private boolean load = false;
 private ScheduledThreadPoolExecutor exec = new ScheduledThreadF
 private Future loadFuture = null;
 private TextRenderer textRenderer;

 public static void main(String[] args) {
 TestLoadingScreen1 app = new TestLoadingScreen1();
 app.start();
 }

 @Override
 public void simpleInitApp() {
 flyCam.setEnabled(false);
 niftyDisplay = new NiftyJmeDisplay(assetManager,
 inputManager,
 audioRenderer,
 guiViewPort);
 nifty = niftyDisplay.getNifty();

 nifty.fromXml("Interface/nifty_loading.xml", "start", this)

 guiViewPort.addProcessor(niftyDisplay);
 }
}
```

```

@Override
public void simpleUpdate(float tpf) {
 if (load) {
 if (loadFuture == null) {
 //if we have not started loading yet, submit the Callable
 loadFuture = exec.submit(loadingCallable);
 }
 //check if the execution on the other thread is done
 if (loadFuture.isDone()) {
 //these calls have to be done on the update loop thread
 //especially attaching the terrain to the rootNode
 //after it is attached, it's managed by the update
 //and may not be modified from any other thread anymore
 nifty.gotoScreen("end");
 nifty.exit();
 guiViewPort.removeProcessor(niftyDisplay);
 flyCam.setEnabled(true);
 flyCam.setMoveSpeed(50);
 rootNode.attachChild(terrain);
 load = false;
 }
 }
}

//this is the callable that contains the code that is run on the other thread
//since the assetmanager is threadsafe, it can be used to load assets
//we do *not* attach the objects to the rootNode here!
Callable<Void> loadingCallable = new Callable<Void>() {

 public Void call() {

 Element element = nifty.getScreen("loadlevel").findElement("load");
 TextRenderer textRenderer = element.getRenderer(TextRenderer.class);

 mat_terrain = new Material(assetManager, "Common/MatDefs/Misc/Unshaded.j3m");
 mat_terrain.setTexture("Alpha", assetManager.loadTexture("Textures/Terrain/grass.png"));
 //setProgress is thread safe (see below)
 setProgress(0.2f, "Loading grass");

 Texture grass = assetManager.loadTexture("Textures/Terrain/grass.png");
 }
}

```

```
 grass.setWrap(WrapMode.Repeat);
 mat_terrain.setTexture("Tex1", grass);
 mat_terrain.setFloat("Tex1Scale", 64f);
 setProgress(0.4f, "Loading dirt");

 Texture dirt = assetManager.loadTexture("Textures/Terrain/dirt.png");

 dirt.setWrap(WrapMode.Repeat);
 mat_terrain.setTexture("Tex2", dirt);
 mat_terrain.setFloat("Tex2Scale", 32f);
 setProgress(0.5f, "Loading rocks");

 Texture rock = assetManager.loadTexture("Textures/Terrain/rock.png");

 rock.setWrap(WrapMode.Repeat);

 mat_terrain.setTexture("Tex3", rock);
 mat_terrain.setFloat("Tex3Scale", 128f);
 setProgress(0.6f, "Creating terrain");

 AbstractHeightMap heightmap = null;
 Texture heightMapImage = assetManager.loadTexture("Textures/heightmap.png");
 heightmap = new ImageBasedHeightMap(heightMapImage.getHeight(), heightMapImage.getWidth());
 heightmap.load();
 terrain = new TerrainQuad("my terrain", 65, 513, heightmap);
 setProgress(0.8f, "Positioning terrain");

 terrain.setMaterial(mat_terrain);

 terrain.setLocalTranslation(0, -100, 0);
 terrain.setLocalScale(2f, 1f, 2f);
 setProgress(0.9f, "Loading cameras");

 List<Camera> cameras = new ArrayList<Camera>();
 cameras.add(getCamera());
 TerrainLodControl control = new TerrainLodControl(terrain);
 terrain.addControl(control);
 setProgress(1f, "Loading complete");
```

```
 return null;
 }

};

public void setProgress(final float progress, final String load
 //since this method is called from another thread, we enqueue
 enqueue(new Callable() {

 public Object call() throws Exception {
 final int MIN_WIDTH = 32;
 int pixelWidth = (int) (MIN_WIDTH + (progressBarEle
 progressBarElement.setConstraintWidth(new SizeValue
 progressBarElement.getParent().layoutElements();

 textRenderer.setText/loadingText);
 return null;
 }
 });
});

public void showLoadingMenu() {
 nifty.gotoScreen("loadlevel");
 load = true;
}

@Override
public void onStartScreen() {
}

@Override
public void onEndScreen() {
}

@Override
public void bind(Nifty nifty, Screen screen) {
 progressBarElement = nifty.getScreen("loadlevel").findElement
}

// methods for Controller
```

```
@Override
public boolean inputEvent(final NiftyInputEvent inputEvent) {
 return false;
}

@Override
public void bind(Nifty nifty, Screen screen, Element elmnt, Prc
 progressBarElement = elmnt.findElementByName("progressbar")
}

@Override
public void init(Properties prprts, Attributes atrbts) {
}

public void onFocus(boolean getFocus) {
}

@Override
public void stop() {
 super.stop();
 //the pool executor needs to be shut down so the application
 exec.shutdown();
}
}
```

&lt;/div&gt;

# title: Localizing jME 3 Games

## Localizing jME 3 Games

</div>

### Scope

Localizing an application can mean several things:

- At minimum you translate all messages and dialogs in the user interface to your target languages.
- You should also translate the “read me”, help, and other documentation.
- Also translating web content related to the application makes sure international users find out about your localized game.
- If you go the whole way of internationalization, you also “translate” metaphors in icons or symbols used.

E.g. For localizations to right-to-left languages, you must also adjust the whole flow of the UI (order of menus and buttons).

There are tools that assist you with localizing Java Swing GUIs. jME3 applications do not typically have a Swing GUI, so those tools are not of much help. Just stick to the normal Java rules about using Bundle Properties:

</div>

## Preparing the Localization

**Tip:** The jMonkeyEngine SDK supports opening and editing Bundle.properties files. Also note the Tools > Localization menu.

To prepare the application for localization, you have to first identify all hard-coded messages.

1. Find every line in your jME3 game where you hard-coded message strings, e.g.

```
System.out.print("Hello World!");
UiText.setText("Score: "+score);
```

2. Create one file named `Bundle.properties` in each directory where there are Java files that contain messages.
3. For every hard-coded message, you add one line to the `Bundle.properties` file: First specify a unique key that identifies this string; then an equal sign; and the literal string itself.

```
greeting=Hello World!
score.display=Score:
```

4. In the source code, replace every occurrence of a hard-coded message with the appropriate Resource Bundle call to its unique key:

```
System.out.print(ResourceBundle.getBundle("Bundle").getString("greeting"));
UiText.setText(ResourceBundle.getBundle("Bundle").getString("score.display"));
```

The language used in the `Bundle.properties` files will be the default language for your game.

</div>

## Translating the Messages

Each additional language comes in a set of files that is marked with a (usually) two-letter suffix. Common locales are de for German, en for English, fr for French, ja for Japanese, pt for Portuguese, etc.

To translate the messages to another language, for example, German:

1. Make a copy of the `Bundle.properties` files.
2. Name the copy `Bundle_de.properties` for German. Note the added suffix `_de`.
3. Translate all strings (text on the right side of the equal sign) in the `Bundle_de.properties` to German.

```
greeting=Hallo Welt!
score.display=Spielstand:
```

**Important:** Do not modify any of the keys (text to the left of the equal sign)!

4. To test the German localization, start the application from the command line with `-Duser.language=de`. Note the parameter `de`.

**Tip:** In the jMonkeyEngine SDK, you set this VM Option in the Project properties under Run. Here you can also save individual run configurations for each language you want to test.

To get the full list of language suffixes use

```
System.out.println(Arrays.toString(Locale.getISOLanguages()));
```

## Which Strings Not to Translate

**Important:** In the Bundle.properties file, do not include any strings that are asset paths, node or geometry names, input mappings, or material layers.

- Keep material layers:

```
mat.setTexture("ColorMap", tex);
```

- Keep paths:

```
teapot = assetManager.loadModel("Models/Teapot/Teapot.obj");
```

- Keep geometry and node names:

```
Geometry thing=new Geometry("A thing", mesh);
Node vehicle = new Node("Vehicle");
```

- Keep mappings:

```
inputManager.addMapping("Shoot", trigger);
inputManager.addListener(actionListener, "Shoot");
```

Only localize messages and UI text!

</div>

## Common Localization Problems

Typical problems include:

- Localized strings will be of vastly different lengths and will totally break your UI layout. ⇒ Test every localization.
- Strings with variable text or numbers don't work the same in different languages. ⇒ Either work in grammatical cases/numbers/gender for each language, or use [gettext](#) or [ICU4J](#).

- The localizer only sees the strings, without any context. E.g. does “Search History” mean “display the history of searches”, or “search through the history”? ⇒ Use clear key labels. Work closely with the localizers if they require extra info, and add that info as comments to the translation file.
- Broken international characters ⇒ Make sure the files are saved with the same character encoding as the font file(s) you're using. Nowadays, that usually means UTF-8 since font files tend to come for Unicode.
- Missing international characters ⇒ Make sure that there's a glyph for every needed character in your font, either by using more complete font files or by having the translation changed.

## More Documentation

<http://java.sun.com/developer/technicalArticles/Intl/ResourceBundles/>

<http://sourceforge.net/apps/mediawiki/nifty-gui/index.php?title=Localisation>

</div>

## title: Logging and Monitoring

# Logging and Monitoring

## Logging Like a Newbie

Many developers just use `System.out.println()` to print diagnostic strings to the terminal. The problem with that is that before the release, you have to go through all your code and make certain you removed all these `println()` calls. You do not want your customers to see them, and needlessly worry about ominous outdated debugging diagnostics.

## Logging Like a Pro

Instead of `println()`, use the standard Java logger from `java.util.logging`. It has many advantages for professional game development:

- You tag each message with its **log level**: Severe error, informative warning, etc.
- You can **switch off or on printing of log messages** up to certain log level with just one line of code.
  - During development, you would set the log level to `fine`, because you want all warnings printed.
  - For the release, you set the log level to only report `severe` errors, and never print informative diagnostics.
- The logger message string is **localizable** and can use variables. Optimally, you localize all error messages.

To print comments like a pro, you use the following logger syntax.

1. Declare the logger object once per file. In the following code, replace `HelloWorld` by the name of the class where you are using this line.

```
private static final Logger logger = Logger.getLogger(HelloWorld)
```

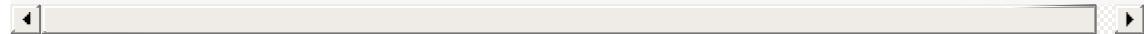
2. Declare the info that you want to include in the message. The variables (here `a, b, c`) can be any printable Java object.

Example: `Vector3f a = cam.getLocation();`

3. Put the variables in a new `Object` array. Refer to the variables as `{0}, {1}, {2}` etc in the message string. Variables are numbered in the order you put them into the `Object` array.
4. Add the logger line and specify the log level:

- Usecase 1: During debugging, a developer uses a warning to remind himself of a bug:

```
logger.log(Level.WARNING, "why is {0} set to {1} again?!",
 new Object[]{a , b});
```



- Usecase 2: For the release, you inform the customer of a problem and how to solve it.

```
logger.log(Level.SEVERE, "MyGame error: {0} must not be {1}
 new Object[]{a , b , c});
```



As you see in the examples, you should phrase potentially “customer facing” errors in a neutral way and offer *a reason and a solution* for the error (if you don't, it has no value to your customer). If your development team uses `WARNINGS` as replacement for casual `printlns`, make sure you deactivate them for the release.

More details about [Java log levels](#) here.

</div>

## Switching the Logger on and off

In the release version you will deactivate the logging output to the terminal.

To deactivate the default logger for a release, you set the log level to only report `severe` messages:

```
Logger.getLogger("").setLevel(Level.SEVERE);
```

During development or a beta test, you can tune down the default logger, and set the log level to only report `warning` s:

```
Logger.getLogger("").setLevel(Level.WARNING);
```

To activate full logging, e.g. for debugging and testing, use the `fine` level:

```
Logger.getLogger("").setLevel(Level.FINE);
```

## Advanced Error Handling

When an uncaught exception reaches certain parts of the jME3 system then the default response is to log the error and then exit the application. This is because an error happening every frame will rapidly fill logs with repeated failings and potentially mask or over-write the original cause of the problem or even the application may continue for a while and then suffer other errors caused by the first and make the root cause hard to determine.

This behaviour can be partially modified by overriding the method `handleError` in `SimpleApplication`, for example to display a custom message to users, or to provide users with information on how to report a bug or even to change the way that the error is logged.

</div>

## title: Making the Camera Follow a 3rd-Person Character

# Making the Camera Follow a 3rd-Person Character

When players steer a game character with 1st-person view, they directly steer the camera (`flyCam.setEnabled(true);`), and they never see the walking character itself. In a game with 3rd-person view, however, the players see the character walk, and you (the game developer) want to make the camera follow the character around when it walks.

There are two ways how the camera can do that:

- Registering a chase camera to the player and the input manager.
- Attaching the camera to the character using a camera node.

**Important:** Using third-person view requires you to deactivate the default flyCam (first-person view). This means that you have to configure your own navigation ([key inputs](#) and [analogListener](#)) that make your player character walk. For moving a physical player character, use `player.setWalkDirection()`, for a non-physical character you can use `player.move()`.

</div>

## Code Samples

Press the WASD or arrow keys to move. Drag with the left mouse button to rotate.

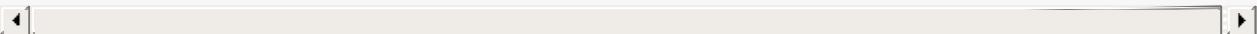
- [TestChaseCamera.java](#)
- [TestCameraNode.java](#)

</div>

## Camera Node

To make the camera follow a target node, add this camera node code to your init method (e.g. `simpleInitApp()`). The `target` spatial is typically the player node.

```
// Disable the default flyby cam
flyCam.setEnabled(false);
//create the camera Node
camNode = new CameraNode("Camera Node", cam);
//This mode means that camera copies the movements of the target:
camNode.setControlDir(ControlDirection.SpatialToCamera);
//Attach the camNode to the target:
target.attachChild(camNode);
//Move camNode, e.g. behind and above the target:
camNode.setLocalTranslation(new Vector3f(0, 5, -5));
//Rotate the camNode to look at the target:
camNode.lookAt(target.getLocalTranslation(), Vector3f.UNIT_Y);
```



**Important:** Where the example says `camNode.setLocalTranslation(new Vector3f(0, 5, -5));`, you have to supply your own start position for the camera. This depends on the size of your target (the player character) and its position in your particular scene. Optimally, you set this to a spot a bit behind and above the target.

Methods	Description
<code>setControlDir(ControlDirection.SpatialToCamera)</code>	User input steers the target spatial, and the camera follows the spatial. The spatial's transformation is copied over the camera's transformation. Example: Use with <a href="#">CharacterControlled</a> spatial.
<code>setControlDir(ControlDirection.CameraToSpatial)</code>	User input steers the camera, and the target spatial follows the camera. The camera's transformation is copied over the spatial's transformation. Use with first-person flyCam.

### Code sample:

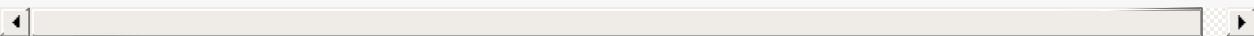
- [TestCameraNode.java](#) – Press the WASD or arrow keys to move. Drag with the left mouse button to rotate.

</div>

## Chase Camera

To activate the chase camera, add the following code to your init method (e.g. `simpleInitApp()` ). The `target` spatial is typically the player node. You will be able to rotate the target by dragging (keeping the left mouse button pressed and moving the mouse).

```
// Disable the default flyby cam
flyCam.setEnabled(false);
// Enable a chase cam for this target (typically the player).
ChaseCamera chaseCam = new ChaseCamera(cam, target, inputManager);
chaseCam.setSmoothMotion(true);
```



Method	Description
setInvertVerticalAxis(true)	Invert the camera's vertical rotation Axis
setInvertHorizontalAxis(true)	Invert the camera's horizontal rotation Axis
setTrailingEnabled(true)	Camera follows the target and flies around and behind when the target moves towards the camera. Trailing only works with smooth motion enabled. (Default)
setTrailingEnabled(false)	Camera follows the target, but does not rotate around the target when the target changes direction.
setSmoothMotion(true)	Activate SmoothMotion when trailing. This means the camera seems to accelerate and fly after the character, when it has caught up, it slows down again.
setSmoothMotion(false)	Disable smooth camera motion. Disabling SmoothMotion also disables trailing.
setLookAtOffset(Vector3f.UNIT_Y.mult(3))	Camera looks at a point 3 world units above the target.
setToggleRotationTrigger(new MouseButtonTrigger(MouseInput.BUTTON_MIDDLE))	Enable rotation by keeping the middle mouse button pressed (like in Blender). This disables the rotation on right and left mouse button click.
setToggleRotationTrigger(new MouseButtonTrigger(MouseInput.BUTTON_MIDDLE), new KeyTrigger(KeyInput.KEY_SPACE))	Activate multiple triggers for the rotation of the camera, e.g. spacebar and middle mouse button, etc.
setRotationSensitivity(5f)	How fast the camera rotates. Use values around <1.0f (all bigger values are ignored).

**Code sample:**

- [TestChaseCamera.java](#) – Press the WASD or arrow keys to move. Drag with the left mouse button to rotate.

```
</div>
```

## Which to Choose?

What is the difference of the two code samples above?

CameraNode	ChaseCam
Camera follows immediately, flies at same speed as target.	Camera moves smoothly and accelerates and decelerates, flies more slowly than the target and catches up.
Camera stays attached to the target at a constant distance.	Camera orbits the target and approaches slowly.
Drag-to-Rotate rotates the target and the camera. You always see the target from behind.	Drag-to-Rotate rotates only the camera. You can see the target from various sides.

```
</div>
```

## title: How to Use Material Definitions (.j3md)

# How to Use Material Definitions (.j3md)

All Geometries need a Material to be visible. Every Material is based on a Material Definition. Material definitions provide the “logic” for the material, and a shader draws the material according to the parameters specified in the definition. The J3MD file abstracts the shader and its configuration away from the user, allowing a simple interface where the user can simply set a few parameters on the material to change its appearance and the way it's handled by the shaders.

The most common Material Definitions are included in the engine, advanced users can create custom ones. In this case you will also be interested in the [in-depth developer specification of the jME3 material system](#).

### Example:

```
Spatial myGeometry = assetManager.loadModel("Models/Teapot/Teapot.j
Material mat = new Material(assetManager, // Create new material a
 "Common/MatDefs/Misc/Unshaded.j3md"); // ... specify a Materia
mat.setColor("Color", ColorRGBA.Blue); // Set one or more mater
myGeometry.setMaterial(mat); // Use material on this
```

If you use one custom material with certain settings very often, learn about storing material settings in [j3m Material Files](#). You either [use the jMonkeyEngine SDK to create .j3m files](#) (user-friendly), or you [write .j3m files in a text editor](#) (IDE-independent).

</div>

## Preparing a Material

In the [Materials Overview](#) list:

1. Choose a Material Definition that has the features that you need.
  - Tip: If you don't know, start with `Unshaded.j3md` or `Lighting.j3md`.
2. Look at the applicable parameters of the Material Definition and determine which parameters you need to achieve the desired effect (e.g. “glow” or “color”). Most

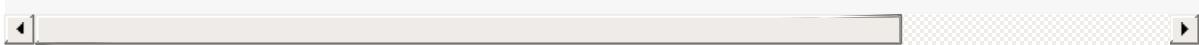
- parameters are optional!
3. Create and save the necessary Texture files to your `assets/Textures` directory.
    - E.g. `mytex_diffuse.png` as ColorMap / DiffuseMap, `mytex_normal.png` as NormalMap, `mytex_alpha.png` as AlphaMap, etc...
  4. Determine the required values to achieve the effect that you want.
    - E.g. set colors, floats, booleans, etc...

## Using a Material

In your Java code,

1. Create a Material object based on the chosen Material Definition (.j3md file):

```
Material mat = new Material(assetManager, "Common/MatDefs/Misc/
```

- 
2. Configure your Material by setting the appropriate values listed in the [Materials Overview](#) table.

```
mat.setColor("Color", ColorRGBA.Yellow); // and more
```

3. Apply your prepared Material to a Geometry:

```
myGeometry.setMaterial(mat);
```

4. (Optional) Adjust the texture scale of the mesh:

```
myGeometryMesh.scaleTextureCoordinates(new Vector2f(2f, 2f));
```

For details see also: [How to Use Materials](#)

## Examples

Here are examples of the methods that set the different data types:

- `mat.setColor("Color", ColorRGBA.White);`
- `mat.setTexture("ColorMap", assetManager.loadTexture("Interface/Logo/Monkey.png"));`
- `mat.SetFloat("Shininess", 5f);`
- `mat.SetBool("SphereMap", true);`
- `mat.setVector3("NormalScale", new Vector3f(1f, 1f, 1f));`

A simpled textured material.

```
Material mat = new Material(assetManager,
 "Common/MatDefs/Misc/Unshaded.j3md");
mat.setTexture("ColorMap", assetManager.loadTexture(
 "Interface/Logo/Monkey.jpg"));
```

A textured material with a color bleeding through transparent areas.

```
Material mat = new Material(assetManager,
 "Common/MatDefs/Misc/Unshaded.j3md");
mat.setTexture("ColorMap", assetManager.loadTexture(
 "Textures/ColoredTex/Monkey.png"));
mat.setColor("Color", ColorRGBA.Blue);
```

You can test these examples within the following code snippet. It creates a box and applies the material:

```
Box b = new Box(Vector3f.ZERO, 1, 1, 1);
Geometry geom = new Geometry("Box", b);
// ... insert Material definition...
geom.setMaterial(mat);
rootNode.attachChild(geom);
```

You can find these and other common code snippets in the jMonkeyEngine SDK Code Palette. Drag and drop them into your source code.

</div>

## Creating a Custom Material Definition

First read the [developer specification of the jME3 material system \(.j3md,.j3m\)](#). Also check out the [engine source code](#) and have a look at how some Material Definitions are implemented.

You can create your own Material Definitions and place them in your project's `assets/MatDefs` directory.

1. Find the existing MatDefs in `engine/src/core-data/Common/MatDefs/`.
2. Open a Something.j3md file in a text editor. You see that this .j3md file defines Material Parameters and Techniques.
  - Material Parameters are the ones that you set in Materials, as shown in the

examples above.

- The Techniques rely on VertexShaders and FragmentShaders: You find those in the files Something.vert and Something.frag in the same directory.
3. Learn about GLSL (OpenGL Shading Language) to understand the .vert and .frag syntax, then write your own.

## Related Links

- [Developer specification of the jME3 material system \(.j3md,.j3m\)](#)

[Material](#), [SDK](#), [MatDef](#), [file](#), [documentation](#)

</div>

# title: jMonkeyEngine3 Material Specification

## jMonkeyEngine3 Material Specification

</div>

### General Syntax

Material definitions and material instance files are formatted similarly to curly-bracket languages, in other words, you have “blocks” and other “blocks” nested in them, surrounded by curly-brackets. There are statements inside the blocks, the next statement begins after a new line, or a semi-colon to allow two statements on the same line. Comments are made by prefixing with two slashes, the `/* */` style comments are not allowed.

#### Example:

```
RootBlock {
 // Comment
 SubBlock NameForTheBlock {
 Statement1 // Another comment
 }
 SubBlock2 {
 SubSubBlock {
 Statement2
 Statement3
 // two statements on the same line
 Statement4; Statement5
 }
 }
 SubBlock3
 { // bracket can be on next line as well
 }
}
```

The syntax for J3MD and J3M files follows from this base format.

</div>

## Material Definition files (J3MD)

Material definitions provide the “logic” for the material. Usually a shader that will handle drawing the object, and corresponding parameters that allow configuration of the shader. The J3MD file abstracts the shader and its configuration away from the user, allowing a simple interface where one can simply set a few parameters on the material to change its appearance and the way it's handled.

Material definitions support multiple techniques, each technique describes a different way to draw the object. For example, currently in jME3, an additional technique is used to render shadow maps for example.

</div>

## Shaders

Shader support inside J3MD files is rather sophisticated. First, shaders may reference shader libraries, in a similar way to Java's “import” statement, or C++'s “include” pre-processor directive. Shader libraries in turn, can also reference other shader libraries this way. In the end, it is possible for a shader to use many functions together from many libraries and combine them in ways to create a more advanced effect. For example, any shader that wishes to take advantage of hardware skinning, can just import the skinning shader library and use the function, without having to write the specific logic needed for hardware skinning.

Shaders can also take advantage of “defines” that are specified inside material definitions. The defines “bind” into material parameters, so that a change in a material parameter can apply or remove a define from the corresponding shader. This allows the shader to completely change in behavior during run-time.

Although it is possible to use shader uniforms for the very same purpose, those may introduce slowdowns in older GPUs, that do not support branching. In that case, using defines can allow changing the way the shader works without using shader uniforms. In order to introduce a define into a shader, however, its source code must be changed, and therefore, it must be re-compiled. It is therefore not recommended to change define bound parameters often.

</div>

## Syntax of a J3MD file

All J3MD files begin with `MaterialDef` as the root block, following that, is the name of the material def (in this example `Test Material 123` ). The name is not used for anything important currently, except for debugging. The name is typed as is without quotes, and can have spaces.

### Example of a first line of a J3MD file:

```
MaterialDef Test Material 123 {
```

Inside a `MaterialDef` block, there can be at most one `MaterialParameters` block, and one or more `Technique` blocks.

Techniques may have an optional name, which specifies the name of the technique. If no name is specified for a technique, then its name is “Default”, and it is used by default if the user does not specify another technique to use for the material.

### Example of J3MD:

```
MaterialDef Test Material 123 {
 MaterialParameters { }
 Technique { }
 Technique NamedTech { }
}
```

Inside the `MaterialParameters` block, the parameters are specified. Every parameter has a type and a name. Material parameters are similar to Java variables in that aspect.

### Example of a `MaterialParameters` block:

```
MaterialParameters {
 Texture2D TexParam
 Color ColorParam
 Vector3 VectorParam
 Boolean BoolParam
 // ...
}
```

Whereas in the J3MD file, the parameter names and types are specified, in a J3M (Material instance) file, the values for these parameters are assigned, as will be shown later. This is how the materials are configured.

At the time of writing, the following types of parameters are allowed inside J3MD files: Int, Boolean, Float, Vector2, Vector3, Vector4, Texture2D, TextureCubeMap.

You can specify a default value for material parameters, inside material definitions, in the case that no value is specified in the material instance.

```
MaterialParameters {
 Float MyParam : 1
 // ...
}
```

1 will be used as the default value and sent to the shader if it has not been set by a material.setFloat() call.

</div>

## Techniques

Techniques are more advanced blocks than the MaterialParameters block. Techniques may have nested blocks, any many types of statements.

In this section, the statements and nested blocks that are allowed inside the Technique block will be described.

The two most important statements, are the `FragmentShader` and `VertexShader` statements. These statements specify the shader to use for the technique, and are required inside the “Default” technique. Both operate in the same way, after the statement, the language of the shader is specified, usually with a version number as well, for example `GLSL100` for OpenGL Shading Language version 1.00. Followed by a colon and an absolute path for an asset describing the actual shader source code. For GLSL, it is permitted to specify `.glsl`, `.frag`, and `.vert` files.

When the material is applied to an object, the shader has its uniforms set based on the material parameter values specified in the material instance. but the parameter is prefixed with an “m\_”.

For example, assuming the parameter `Shininess` is defined in the MaterialParameters block like so:

```
MaterialParameters {
 Float Shininess
}
```

The value of that parameter will map into an uniform with same name with the “m\_” prefix in the GLSL shader:

```
uniform float m_Shininess;
```

The letter `m` in the prefix stands for material.

</div>

## World/Global parameters

An important structure, that also relates to shaders, is the `WorldParameters` structure. It is similar in purpose to the `MaterialParameters` structure; it exposes various parameters to the shader, but it works differently. Whereas the user specified material parameters, world parameters are specified by the engine. In addition, the `WorldParameters` structure is nested in the `Technique`, because it is specific to the shader being used. For example, the `Time` world parameter specifies the time in seconds since the engine started running, the material can expose this parameter to the shader by specifying it in the `WorldParameters` structure like so:

```
WorldParameters {
 Time
 // ...
}
```

The shader will be able to access this parameter through a uniform, also named `Time` but prefixed with `g_`:

```
uniform float g_Time;
```

The `g` letter stands for “global”, which is considered a synonym with “world” in the context of parameter scope.

There are many world parameters available for shaders, a comprehensive list will be specified elsewhere.

</div>

## RenderState

The RenderState block specifies values for various render states in the rendering context. The RenderState block is nested inside the Technique block. There are many types of render states, and a comprehensive list will not be included in this document.

The most commonly used render state is alpha blending, to specify it for a particular technique, including a RenderState block with the statement `Blend Alpha`.

**Example:**

```
RenderState {
 Blend Alpha
}
```

**Full Example of a J3MD**

Included is a full example of a J3MD file using all the features learned:

```
MaterialDef Test Material 123 {
 MaterialParameters {
 Float m_Shininess
 Texture2D m_MyTex
 }
 Technique {
 VertexShader GLSL100 : Common/MatDefs/Misc/MyShader.vert
 FragmentShader GLSL100 : Common/MatDefs/Misc/MyShader.frag
 WorldParameters {
 Time
 }
 RenderState {
 Blend Alpha
 }
 }
}
```

## Material Instance files (J3M)

In comparison to J3MD files, material instance (J3M) files are significantly simpler. In most cases, the user will not have to modify or create his/her own J3MD files.

All J3M files begin with the word `Material` followed by the name of the material (once again, used for debugging only). Following the name, is a colon and the absolute asset path to the material definition (J3MD) file extended or implemented, followed by a curly-bracket.

### **Example:**

```
Material MyGrass : Common/MatDefs/Misc/TestMaterial.j3md {
```

The material definition is a required component, depending on the material definition being used, the appearance and functionality of the material changes completely. Whereas the material definition provided the “logic” for the material, the material instance provides the configuration for how this logic operates.

The J3M file includes only a single structure; `MaterialParameters`, analogous to the same-named structure in the J3MD file. Whereas the J3MD file specified the parameter names and types, the J3M file specifies the values for these parameters. By changing the parameters, the configuration of the parent J3MD changes, allowing a different effect to be achieved.

To specify a value for a parameter, one must specify first the parameter name, followed by a colon, and then followed by the parameter value. For texture parameters, the value is an absolute asset path pointing to the image file. Optionally, the path can be prefixed with the word “Flip” in order to flip the image along the Y-axis, this may be needed for some models.

### **Example of a MaterialParameters block in J3M:**

```
MaterialParameters {
 m_Shininess : 20.0
}
```

Param type	Value example
Int	123
Boolean	true
Float	0.1
Vector2	0.1 5.6
Vector3	0.1 5.6 2.99
Vector4=Color	0.1 5.6 2.99 3
Texture2D=TextureCubeMap	Textures/MyTex.jpg

}

The formatting of the value, depends on the type of the value that was specified in the J3MD file being extended. Examples are provided for every parameter type:

### Full example of a J3M

```
Material MyGrass : Common/MatDefs/Misc/TestMaterial.j3md {
 MaterialParameters {
 m_MyTex : Flip Textures/GrassTex.jpg
 m_Shininess : 20.0
 }
}
```

### Java interface for J3M

It is possible to generate an identical J3M file using Java code, by using the classes in the com.jme3.material package. Specifics of the [Material API](#) will not be provided in this document. The J3M file above is represented by this Java code:

```
// Create a material instance
Material mat = new Material(assetManager, "Common/MatDefs/Misc/
 TestMaterial.j3md");
// Load the texture. Specify "true" for the flip flag in the Texture
Texture tex =
assetManager.loadTexture(new TextureKey("Textures/GrassTex.jpg", tr
// Set the parameters
mat.setTexture("MyTex", tex);
mat.setFloat("Shininess", 20.0f);
```

## Conclusion

Congratulations on being able to read this entire document! To reward your efforts, jMonkeyEngine.com will offer a free prize, please contact Momoko\_Fan aka “Kirill Vainer” with the password “bananapie” to claim.

</div>

## title: Material Definition Properties

# Material Definition Properties

In jMonkeyEngine 3, colors and textures are represented as Material objects.

- All Geometries must have Materials. To improve performance, reuse Materials for similar models, don't create a new Material object for every Geometry. (E.g. use one bark Material for several tree models.)
- Each Material is based on one of jme3's default Material Definitions (.j3md files) that are included in the engine. Advanced users can create additional custom Material Definitions (see how it's done in the [jme3 sources](#)).

Find out quickly [How to Use Materials](#), including the most commonly used code samples and RenderStates.

Or find more background info on [How to use Material Definitions](#).

</div>

## All Materials Definition Properties

The following Materials table shows you the Material Definitions that jMonkeyEngine 3 supports.

Looks confusing?

- 1) Start learning about `Unshaded.j3md` and `Lighting.j3md`, they cover 90% of the cases.
- 2) Use [the SDK's visual material editor](#) to try out and save material settings easily.
- 3) The [SDK's Palette](#) contains drag&drop code snippets for loading materials.

Most Material parameters are optional. For example, it is okay to specify solely the `DiffuseMap` and `NormalMap` when using `Lighting.j3md`, and leave the other texture maps empty. In this case, you are only using a subset of the possible features, but that's fine if it already makes the material look the way that you want. You can always add more texture maps later.

</div>

## Unshaded Coloring and Textures

jMonkeyEngine supports illuminated and unshaded Material Definitions.

- Phong Illuminated materials look more naturalistic.
- Unshaded materials look more abstract.

“Unshaded” materials look somewhat abstract because they ignore lighting and shading.

Unshaded Materials work even if the scene does not include a light source. These Materials can be single-colored or textured. For example, they are used for cards and tiles, for the sky, billboards and UI elements, for toon-style games, or for testing.

Standard Unshaded Material Definition	Usage	Material Parameters
Common/MatDefs/Misc/Unshaded.j3md	Standard, non-illuminated Materials. Use this for simple coloring and texturing, glow, and transparency. See also: <a href="#">Hello Material</a>	<b>Texture Maps</b> <code>setTexture("ColorMap", assetManager.loadTexture(""))</code> <code>setBoolean("SeparateTexCoo</code> <b>Colors</b> <code>setColor("Color", ColorRGBA.</code> <code>setBoolean("VertexColor",true</code> <b>Glow</b> <code>setTexture("GlowMap", assetManager.loadTexture(""))</code> <code>setColor("GlowColor", ColorRGBA.White);</code>

Other useful, but less commonly used material definitions:

Special Unshaded Material Definitions	Usage	Material Parameters
Common/MatDefs/Misc/Sky.j3md	A solid skyblue, or use with a custom SkyDome texture. See also: <a href="#">Sky</a>	<code>setTexture("Texture", assetManager.loadTexture("sky"))</code> <code>setBoolean("SphereLighting", true)</code> <code>setVector3("Normal", new Vector3f(0,0,0))</code>
Common/MatDefs/Terrain/Terrain.j3md	Splat textures for e.g. terrains. See also: <a href="#">Hello Terrain</a>	<code>setTexture("Tex1", assetManager.loadTexture("red"))</code> <code>setFloat("Tex1Scale", 1.0f)</code> <code>setTexture("Tex2", assetManager.loadTexture("green"))</code> <code>setFloat("Tex2Scale", 1.0f)</code> <code>setTexture("Tex3", assetManager.loadTexture("blue"))</code> <code>setFloat("Tex3Scale", 1.0f)</code>

		setFloat("Tex3Scale", setTexture("Alpha", assetManager.load")
Common/MatDefs/Terrain/HeightBasedTerrain.j3md	A multi-layered texture for terrains. Specify four textures and a Vector3f describing the region in which each texture should appear: X = start height, Y = end height, Z = texture scale. Texture regions can overlap. For example: Specify a seafloor texture for the lowest areas, a sandy texture for the beaches, a grassy texture for inland areas, and a rocky texture for mountain tops.	setFloat("terrainSize", setTexture("region1", assetManager.load") setTexture("region2", assetManager.load") setTexture("region3", assetManager.load") setTexture("region4", assetManager.load") setVector3("region1", Vector3f(0,0,0)); setVector3("region2", Vector3f(0,0,0)); setVector3("region3", Vector3f(0,0,0)); setVector3("region4", Vector3f(0,0,0)); <b>Settings for steep slopes</b> setTexture("slopeColor", assetManager.load") setFloat("slopeTiltFactor", 0.5);
	Used with texture	

Common/MatDefs/Misc/Particle.j3md	<p>masks for particle effects, or for point sprites. The Quadratic value scales the particle for perspective view (<a href="#">formula</a>). Does support an optional colored glow effect. See also: <a href="#">Hello Effects</a></p>	<pre>setTexture("Texture" assetManager.load" setTexture("GlowMa assetManager.load" setColor("GlowColo ColorRGBA.White); setFloat("Quadratic" setBoolean("PointS</pre>
-----------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------

## Phong Illuminated

jMonkeyEngine supports illuminated and unshaded Material Definitions.

- Phong Illuminated materials look more naturalistic.
- Unshaded materials look more abstract.

Illuminated materials require a [light source](#) added to at least one of their parent nodes! (e.g. rootNode.) Illuminated materials are darker on the sides facing away from light sources. They use Phong illumination model (default), or the Ward isotropic gaussian specular shader (WardIso) which looks more like plastic. They do not cast [drop shadows](#) unless you use a FilterPostProcessor.

Standard Illuminated Material Definition	Usage	Material Parameters
Common/MatDefs/Light/Lighting.j3md	<p>Commonly used Material with Phong illumination. Use this material together with DiffuseMap, SpecularMap, BumpMap (NormalMaps, ParallaxMap) textures. Supports shininess, transparency, and plain material colors (Diffuse, Ambient, Specular colors). See also: <a href="#">Hello Material</a></p>	<p><b>Texture Maps</b>  <code>setTexture("DiffuseMap", assetManager.loadTexture(""));  setBoolean("UseAlpha",true);<sup>1)</sup>  setTexture("NormalMap", assetManager.loadTexture(""));  setBoolean("LATC",true);<sup>2)</sup>  setTexture("SpecularMap", assetManager.loadTexture(""));  setFloat("Shininess",64f);  setTexture("ParallaxMap", assetManager.loadTexture(""));  setTexture("AlphaMap", assetManager.loadTexture(""));  setFloat("AlphaDiscardThreshold",0.5f);  setTexture("ColorRamp", assetManager.loadTexture(""));  <b>Glow</b>  setTexture("GlowMap", assetManager.loadTexture(""));  setColor("GlowColor", ColorRGBA.White);  <b>Performance and quality</b>  setBoolean("VertexLighting",true);  setBoolean("UseVertexColor",true);  setBoolean("LowQuality",true);  setBoolean("HighQuality",true);  <b>Material Colors</b>  setBoolean("UseMaterialColors",true);  setColor("Diffuse", ColorRGBA.White);  setColor("Ambient", ColorRGBA.White);  setColor("Specular", ColorRGBA.White);  <b>Tangent shading:</b>  setBoolean("VTangent",true);  setBoolean("Minnaert",true);<sup>3)</sup>  setBoolean("WardIso",true);<sup>4)</sup></code></p>

Special Illuminated Material Definitions	Usage	More
Common/MatDefs/Terrain/TerrainLighting.j3md	<p>Same kind of multi-layered splat texture as Terrain.j3md, but with illumination and shading.</p> <p>Typically used for terrains, but works on any mesh.</p> <p>For every 3 splat textures, you need one alpha map.</p> <p>You can use a total of 11 texture maps in the terrain's splat texture: Note that diffuse and normal maps all count against that.</p> <p>For example, you can use a maximum of 9 diffuse textures, two of which can have normal maps; or, five textures with both diffuse and normal maps.</p>	<b>Texture Splat</b> <code>setTexture("assetManager", "TerrainLighting")</code> <code>setFloat("Diffuse", 1.0f)</code> <code>setTexture("assetManager", "TerrainLighting")</code> <code>setTexture("assetManager", "TerrainLighting")</code> <code>setFloat("Diffuse", 0.5f)</code> <code>setTexture("assetManager", "TerrainLighting")</code> <code>setTexture("assetManager", "TerrainLighting")</code> <code>setFloat("Diffuse", 0.0f)</code> <code>setTexture("assetManager", "TerrainLighting")</code> <code>setTexture("assetManager", "TerrainLighting")</code> <code>setFloat("Diffuse", 0.5f)</code> <code>setTexture("assetManager", "TerrainLighting")</code> <code>setTexture("assetManager", "TerrainLighting")</code> <code>setFloat("Diffuse", 0.0f)</code> <code>etc, up to 11</code> <b>Alpha Maps</b> <code>setTexture("assetManager", "TerrainLighting")</code> <code>setTexture("assetManager", "TerrainLighting")</code> <code>setTexture("assetManager", "TerrainLighting")</code> <code>setTexture("assetManager", "TerrainLighting")</code> <code>setTexture("assetManager", "TerrainLighting")</code> <b>Glowing</b> <code>setTexture("assetManager", "TerrainLighting")</code> <code>setColor("GlowColor", ColorRGBA(1.0f, 0.0f, 0.0f, 1.0f))</code> <b>Miscellaneous</b> <code>setColor("DiffuseColor", ColorRGBA(1.0f, 0.0f, 0.0f, 1.0f))</code> <code>setColor("AmbientColor", ColorRGBA(0.0f, 0.0f, 0.0f, 1.0f))</code> <code>setFloat("Shininess", 100.0f)</code> <code>setColor("SpecularColor", ColorRGBA(1.0f, 1.0f, 1.0f, 1.0f))</code> <code>setTexture("assetManager", "TerrainLighting")</code> <code>setBoolean("IsGlowing", true)</code> <code>setBoolean("IsGlowing", false)</code> <code>setBoolean("IsGlowing", true)</code>
Common/MatDefs/Light/Reflection.j3md	<p>Reflective glass material with environment map (CubeMap/SphereMap). See also:</p> <p><a href="#">TestCubeMap.java</a></p>	<code>setTexture("assetManager", "Reflection")</code> <code>setBoolean("IsEnvironmentMap", true)</code>

## Other: Test and Debug

Material Definition	Usage
Common/MatDefs/Misc/ShowNormals.j3md	A color gradient calculated from the model's surface normals. You can use this built-in material to debug the generation of normals in meshes, to preview models that have no material and no lights, or as fall-back default material. This built-in material has no parameters.

# RenderStates

## Transparency

Material Option	Description
<code>getAdditionalRenderState().setBlendMode(BlendMode.Off);</code>	This is the default transparency.
<code>getAdditionalRenderState().setBlendMode(BlendMode.Alpha);</code>	Interpolates the background pixel with the current pixel using the current alpha.
<code>getAdditionalRenderState().setDepthWrite(false);</code>	Disables writing pixel's depth value to the depth buffer.
<code>getAdditionalRenderState().setAlphaFallOff(0.5f); getAdditionalRenderState().setAlphaTest(true)</code>	Enables Alpha Testing with a "AlphaDiscardThreshold" in the AlphaMap
	Additive alpha blending adds colors in a commutative way.

	the result does not depend on the order of transparent layers; it adds the scene's background pixels to the current pixel. This is useful if you have lots of transparent textures overlapping and don't care about their order.
getAdditionalRenderState().setBlendMode(BlendMode.Additive);	<b>Note:</b> Viewed in a white background, Additive textures become fully transparent!
getAdditionalRenderState().setBlendMode(BlendMode.AlphaAdditive);	Same as "Additive" except first it multiplies the current pixel by the pixel alpha.
getAdditionalRenderState().setBlendMode(BlendMode.Color);	Blends by color.
getAdditionalRenderState().setBlendMode(BlendMode.Modulate);	Multiplies the background pixels by the current pixel.
getAdditionalRenderState().setBlendMode(BlendMode.ModulateX2);	Same as "Modulate" except the result is doubled.
getAdditionalRenderState().setBlendMode(BlendMode.PremultAlpha);	Pre-multiplied alpha blending. E.g. if the object has been multiplied by its alpha, this is used instead of "Alpha" mode.

If the DiffuseMap has an alpha channel, use:

```
mat.setBoolean("UseAlpha", true);
```

Later, put the Geometry (not the Material!) in the appropriate render queue

```
geo.setQueueBucket(Bucket.Translucent);
```

```
geo.setQueueBucket(Bucket.Transparent);
```

## Culling

Material Option	Usage
<pre data-bbox="181 601 1111 637">getAdditionalRenderState().setFaceCullMode(FaceCullMode.Back);</pre>	<p>Activates back-face culling. Mesh faces that are facing away from the camera are not rendered, which saves time. *Backface culling is activated by default as major optimization.</p>
<pre data-bbox="181 1156 1079 1192">getAdditionalRenderState().setFaceCullMode(FaceCullMode.Off);</pre>	<p>No meshes are culled. Both mesh faces are rendered, even if the face away from the camera. Slow.</p>
<pre data-bbox="181 1560 1111 1596">getAdditionalRenderState().setFaceCullMode(FaceCullMode.Front);</pre>	<p>Activates front-face culling. Mesh faces facing the camera are not rendered.</p>

getAdditionalRenderState().setFaceCullMode(FaceCullMode.FrontAndBack)	backfaces and frontfaces.
-----------------------------------------------------------------------	---------------------------------

## Miscellaneous

getAdditionalRenderState().setColorWrite(false);	Disable writing the color of pixels.	Use this together with setDepthWrite(true) to write pixels only to the depth buffer, for example.
getAdditionalRenderState().setPointSprite(true);	Enables point-sprite mode, e.g. meshes with "Mode.Points" will be rendered as textured sprites. Note that gl_PointCoord must be set in the shader.	Point sprites are used internally for hardware accelerated particle effects.
getAdditionalRenderState().setPolyOffset();	Enable polygon offset.	Use this when you have meshes that have triangles really close to each other (e.g. <a href="#">Coplanar</a> ), it will shift the depth values to prevent Z-fighting.

### Related Links

- [Developer specification of the jME3 material system \(.j3md,.j3m\)](#)

[material](#), [texture](#), [MatDefs](#), [light](#), [culling](#), [RenderStates](#), [documentation](#)  
</div>

- 1) UseAlpha specifies whether DiffuseMap uses the alpha channel
- 2) LATC Specifies whether NormalMap is BC5/ATI2n/LATC/3Dc-compressed
- 3) Minnaert is a shader type.
- 4) WardIso is a shader type.

## title: Polygon Meshes

# Polygon Meshes



All visible game elements in a scene, whether it is a Model or a Shape, are made up of polygon meshes. JME3 has a com.jme3.scene.Mesh class that represents all meshes.

- Meshes are made up of triangles: `getTriangleCount(...)` and `getTriangle(...)`
- Each mesh has a unique ID: `getId()`
- Meshes have transformations: Location (local translation), rotation, scale.
- Meshes have a bounding volume. jME3 can detect intersections (that is, non-physical collisions) between meshes, or between meshes and 2D elements such as rays:  
`collideWith()` .
- Meshes are locked with `setStatic()` and unlocked with `setDynamic()` .
  - Static Meshes cannot be modified, but are more optimized and faster (they can be precalculated).
  - Dynamic Meshes can be modified live, but are not optimized and slower.

You have several options when [creating Geometries from meshes](#):

- Use built-in [Shapes](#) as meshes;
- Load [3D models](#) (that is, meshes created in external applications); or
- Create free-form [custom meshes](#) programmatically.

## Vertex Buffer

The VertexBuffer contains a particular type of geometry data used by Meshes. Every VertexBuffer set on a Mesh is sent as an attribute to the vertex shader to be processed.

## Mesh Vertex Buffers

Vertex Buffer Type	Description
Type.Position	Position of the vertex (3 floats)
Type.Index	Specifies the index buffer, must contain integer data.
Type.TexCoord	Texture coordinate
Type.TexCoord2	Texture coordinate #2
Type.Normal	Normal vector, normalized.
Type.Tangent	Tangent vector, normalized.
Type.Binormal	Binormal vector, normalized.
Type.Color	Color and Alpha (4 floats)
Type.Size	The size of the point when using point buffers.
Type.InterleavedData	Specifies the source data for various vertex buffers when interleaving is used.
Type.BindPosePosition	Initial vertex position, used with animation.
Type.BindPoseNormal	Initial vertex normals, used with animation
Type.BoneWeight	Bone weights, used with animation
Type.BoneIndex	Bone indices, used with animation

## Mesh Properties

Mesh method	Description
setLineWidth(1)	the thickness of the line if using Mode.Lines
setPointSize(4.0f)	the thickness of the point when using Mode.Points
setBound(boundingVolume)	if you need to specify a custom optimized bounding volume
setStatic()	Locks the mesh so you cannot modify it anymore, thus optimizing its data (faster).
setDynamic()	Unlocks the mesh so you can modify it, but this will un-optimize the data (slower).
setMode(Mesh.Mode.Points)	Used to set mesh rendering modes, see below.
getId()	returns the Mesh ID
getTriangle(int,tri)	returns data of triangle number <code>int</code> into variable <code>tri</code>
scaleTextureCoordinates(Vector2f)	How the texture will be stretched over the whole mesh.

## Mesh Rendering Modes

Mesh Mode	Description
Mesh.Mode.Points	Show only corner points (vertices) of mesh
Mesh.Mode.Lines	Show lines (edges) of mesh
Mesh.Mode.LineLoop	?
Mesh.Mode.LineStrip	?
Mesh.Mode.Triangles	?
Mesh.Mode.TriangleStrip	?
Mesh.Mode.TriangleFan	?
Mesh.Mode.Hybrid	?

## Level of Detail

Optionally, custom meshes can have a LOD (level of detail optimization) that renders more or less detail, depending on the distance of the mesh from the camera. You have to specify several vertex buffers, one for each level of detail you want (very far away with few details, close up with all details, and something in the middle). Use `setLodLevels(VertexBuffer[] lodLevels)`.

```
spatial, node, mesh, geometry, scenegraph
</div>
```

## **title: User Guide for MonkeyBrains**

# **User Guide for MonkeyBrains**

MonkeyBrains is a sophisticated AI Engine for jMonkeyEngine. It uses an agent framework to wrap human and AI controlled characters in plugin-based AI algorithms so that different each game can pick out whichever AI techniques fits best. Download MonkeyBrains from the GitHub [repository](#).

</div>

## **MonkeyBrains Wiki**

User guide for MonkeyBrains is moved to [MonkeyBrains Wiki](#)

</div>

## **Working games:**

For more examples of working games built with MonkeyBrains see:

<https://github.com/QuietOne/MonkeyBrainsDemoGames>

</div>

## **Suggestions and questions:**

If you have suggestion or any questions, please see forum:

<http://hub.jmonkeyengine.org/forum/board/projects/monkeybrains/>

</div>

## **title: Monkey Zone: Multi-player Sample Project**

# **Monkey Zone: Multi-player Sample Project**

MonkeyZone is an multi-player demo game provided by the jME core developer team.

- [Download source code](#) (Subversion Repository)
- [Watch pre-alpha video footage](#) (YouTube Video)
- [Read "MonkeyZone – a jME3 game from the core"](#) (news article)
- Related forum thread: [Open Game Finder](#)

This open-source demo:

1. showcases one possible way to implement a game with jME3, and
2. helps the jME team verify the jME3 [API](#) in terms of usability.

The game idea is based on “BattleZone” arcade game from the 1980s, a first-person shooter with real-time strategy elements. The game was written using the jMonkeyEngine SDK, and it's based off the BasicGame project template. It took us one week to create a playable pre-alpha, including networking. The project design follows best practices that make it possible to edit maps, vehicles, etc, in jMonkeyEngine SDK without having to change the code – This allows 3D graphic designers to contribute models more easily. (If you feel like contributing assets or working on parts of the game code, drop us a note!)

## **Implementation**

MonkeyZone is a multi-player game with a physics simulation. Both, clients and server, run the physics simulation. The clients send input data from the player group to the server, where they control the entities, and also broadcast to the clients. Additionally, the server sends regular synchronization data for all objects in the game to prevent drifting. When a human user or an AI performs an action (presses a button), the actual logic is done on the server. The results are broadcast as data messages to the entities. When the entity is controlled by an AI, the actual AI code (that determines where the entity should move, and when it should perform an action) is executed on the client.

*The way MonkeyZone is implemented is just one of the many possible ways to do a game like this in jME. Some things might be done more efficiently, some might be done in another*

*way completely. MonkeyZone tries to do things the way that are most appropriate to implement the game at hand and it shows nicely how jME3 and the standard Java API can make game development easier and faster. Also note that the way MonkeyZone is designed is not scalable to a MMO style game, it will only work in a FPS style environment where the whole game world can be loaded at once.*

## Terminology

The game uses certain terms that might be familiar to you but maybe used in another way, so heres a quick rundown on the terms being used.

- Player – Logical human or AI player that can enter entities and generally act, only exists as PlayerData “database” with an id.
- Entity – Spatial with UserData, a world object like character, vehicle, box or factory. The base form is defined only by a String pointing to the j3o which already has all userdata like hitpoints, speed etc.
- User – Human player using a client
- Player Group – Group of players that play together (e.g. one human player and one AI companion per client). For now that's the same as client\_id of human player for all AIControl'ed players originating from that client.
- Client – Computer connected to server

## Manager Classes

The WorldManager does the main work of organizing players, entities and the world and synchronizing them between the server and client. Both client and server use this class. Some other managers like ClientEffectsManager only exist on the client or server and manage e.g. effects display. The gameplay is largely controlled by the ServerGameManager which does gameplay logic on the server, combined with the actions issued by the AI and user on the client (see below) it implements the gameplay. It extensively uses the functions exposed by the WorldManager to perform actions and gather data. This is also the class where the actions of the players are actually executed on the server to determine the outcome (ray testing for shooting etc.).

## Use of Controls

Controls are used extensively in MonkeyZone for many aspects of the game. When a player enters an entity, the Spatial Controls are configured based on the player that enters. For example when the human user enters an entity, Controls that update the user interface (DefaultHUDControl) or user input (UserInputControl) are added to the current entity Spatial.

## ...As entity capabilities

Controls attached to Spatial entities are generally used like an “array of capabilities” that the entity possesses. So when an entity has a VehicleControl it’s expected to be a vehicle, when it has a CharacterControl it’s expected to be a character. Other Controls work completely on their own, like CharacterAnimControl which just uses the CharacterControl of the entity to check if the character is running, jumping etc. and then animates the entity if it has an AnimControl.

## ... to abstract

Furthermore there are special interfaces for Controls that allow abstraction of different Controls into one base interface. For example ManualControl and AutonomousControl are interfaces for controls that manage the movement of a spatial in a generalized way. This way AI code and e.g. the UserInputControl only have to check for a valid AutonomousControl or ManualControl on the spatial to control and move it. The details of the movement are handled by classes like ManualVehicleControl and AutonomousCharacterControl.

## ... for AI functions

A special Control called CommandControl handles the Commands that can be executed by user controlled players, see below.

## Artificial Intelligence

MonkeyZone includes simple AI functions based on a command queue.

## Commands

To implement autonomous AI players MonkeyZone uses a system of Commands that are managed by a CommandControl that is attached to each AI player entity controlled by the user. This CommandControl manages a list of Commands that are executed based on priority. For example there are MoveCommand, a FollowCommand and an AttackCommand, Commands can however implement more complete behavior than that, e.g. the complete logic for a scavenging entity.

- Press the WASD keys and use the mouse to move
- press space to jump
- Aim and click to shoot
- Type 1 to select the first Ogre
  - Aim at the floor and press F1 to tell it where to go.
  - Aim at a target and press F2 to tell it who to follow.

- Aim at the car and press F3 to make it drive the car.
- Aim at a target and press F4 to tell it who to attack.
- Walk close to the car and press enter to drive the car.

## Triggers

The SphereTrigger is a TriggerControl that is also attached to each AI players current entity. It consists of a GhostControl that checks the overlapping entities around the entity its attached to. It can be assigned a command that is checked with every entity entering the SphereTrigger and executed if applicable (e.g. normal “attack enemy” mode).

## NavMesh

For each map a navigation mesh is generated that allows the entities to navigate the terrain. Autonomous entities automatically get a NavigationControl based on the current map. The AutonomousControl implementations automatically recognize the NavigationControl attached to the Spatial and use it for navigation. The NavMeshNavigationControl implementation contains a reference to the levels NavMesh and implements a navigation algorithm similar to the A\* algorithm.

## Networking

Networking is realized in the PhysicsSyncManager which we hope to extend to a state where it can serve as a general sync system for physics based network games. The sync manager basically puts a timestamp on every message sent from the server and then buffers all arriving messages on the client within a certain time window. This allows to compensate for messages arriving too soon or too late within the constraints of the buffer, a future version might step the clients physics space different to compensate for network delays without “snapping”.

## Use of jMonkeyEngine SDK tools

All assets used in the game, like entity models and loaded maps can be preconfigured and edited using the jMonkeyEngine SDK. For example, to add a new vehicle type, a vehicle is created in the jMonkeyEngine SDK vehicle editor and UserData like Speed, HitPoints etc. is applied directly in the editor. When the model is loaded in the game it is automatically configured based on these settings, the same accounts for maps that are loaded, special Nodes that mark e.g. player start locations are recognized automatically etc.

## UserData

Entities (Nodes and Geometries) that are loaded from disk have certain UserData like HitPoints, Speed etc. that is used to configure the entity at runtime. The jMonkeyEngine SDK allows adding and editing this UserData, so entity properties are editable visually.

## Physics

VehicleControls, CharacterControls and RigidBodyControls with mesh collision shape for terrain and objects are generated in the jMonkeyEngine SDK and saved in the entity j3o file. When an entity is loaded, the type of entity is identified based on the available controls and UserData and it is configured accordingly.

## API Info

### Designer Infos

Editable UserData of entity Spatials:

- (float) HitPoints
- (float) MaxHitPoints
- (float) Speed

Entity Spatial marking Node names:

- AimNode
- CameraAttachment
- ShootAttachment

Level Spatial marking Node names:

- StartPoint
- PowerSource
- MetalField

### Developer Infos

Programmatic UserData of entities:

- (long) entity\_id
- (int) group\_id
- (long) player\_id

Programmatic PlayerData:

- (long) id

- (int) group\_id
- (long) entity\_id
- (long) character\_entity\_id

## The Future

Have a look at the code and feel free to ask about it, if you want any new features, you are free to implement them. ;) MonkeyZone is hosted at GoogleCode, where you can check out the jMonkeyEngine SDK-ready project via svn:

1. jMonkeyEngine SDK→Team→Subversion→Checkout,
2. Enter the SVN URL <http://monkeyzone.googlecode.com/svn/trunk/>
3. Download, open, and build the project
4. Run the server first (com.jme3.monkeyzone.ServerMain), and then a client (com.jme3.monkeyzone.ClientMain).

## Troubleshooting

1. After download, errors could appear because  
jme3tools.navmesh.util\NavMeshGenerator.java import com.jme3.terrain.Terrain is not known, you should correct this by setting Project Properties > Libraries > Add Library > jme3-libraries-terrain

[network](#), [basegame](#), [physics](#), [inputs](#), [spidermonkey](#)

</div>

## title: MotionPath

# MotionPath

A MotionPath describes the motion of a spatial between waypoints. The path can be linear or rounded. You use MotionPaths to remote-control a spatial, or the camera.

**Tip:** If you want to remote-control a whole cutscene with several spatlals moving at various times, then we recommended you use MotionPaths together with [Cinematics](#).

</div>

## Sample Code

- [TestMotionPath.java](#)
- [TestCameraMotionPath.java](#)

</div>

## What Are Way Points?

When shooting a movie scene, the director tells actors where to walk, for example, by drawing a series of small crosses on the floor. Cameramen often mount the camera on rails (so called dolly track) so they can follow along complex scenes more easily.

In JME3, you use MotionPaths to specify a series of positions for a character or the camera. The MotionPath automatically updates the transformation of the spatial in each frame to make it move from one point to the next.

- **A way point** is one positions on a path.
- **A MotionPath** contains a list of all way points of one path.

The final shape of the path is computed using a linear interpolation or a [Catmull-Rom](#) spline interpolation on the way points.

</div>

# Create a MotionPath

Create a Motionpath object and add way points to it.

```
MotionPath path = new MotionPath();
path.addWayPoint(new Vector3f(10, 3, 0));
path.addWayPoint(new Vector3f(8, -2, 1));
...
...
```

You can configure the path as follows.

MotionPath Method	Usage
path.setCycle(true)	Sets whether the motion along this path should be closed (true) or open-ended (false).
path.addWayPoint(vector)	Adds individual waypoints to this path. The order is relevant.
path.removeWayPoint(vector) removeWayPoint(index)	Removes a way point from this path. You can specify the point that you want to remove as vector or as integer index.
path.setCurveTension(0.83f)	Sets the tension of the curve (Catmull-Rom Spline). A value of 0.0f results in a straight linear line, 1.0 a very round curve.
path.getNbWayPoints()	Returns the number of waypoints in this path.
path.enableDebugShape(assetManager,rootNode)	Shows a line that visualizes the path. Use this during development and for debugging so you see what you are doing.
path.disableDebugShape()	Hides the line that visualizes the path. Use this for the release build.

# MotionPathListener

You can hook interactions into a playing MotionPath. Register a MotionPathListener to the MotionPath to track whether way points have been reached, and then trigger a custom action. The `onWayPointReach()` method of the interface gives you access to the MotionTrack object `control`, and an integer value representing the current wayPointIndex.

In this example, you just print the status at every way point. In a game you could trigger actions here: Transformations, animations, sounds, game actions (attack, open door, etc).

```
path.addListener(new MotionPathListener() {
 public void onWayPointReach(MotionTrack control, int wayPointIndex) {
 if (path.getNbWayPoints() == wayPointIndex + 1) {
 println(control.getSpatial().getName() + " has finished moving");
 } else {
 println(control.getSpatial().getName() + " has reached way point " + wayPointIndex);
 }
 }
});
```

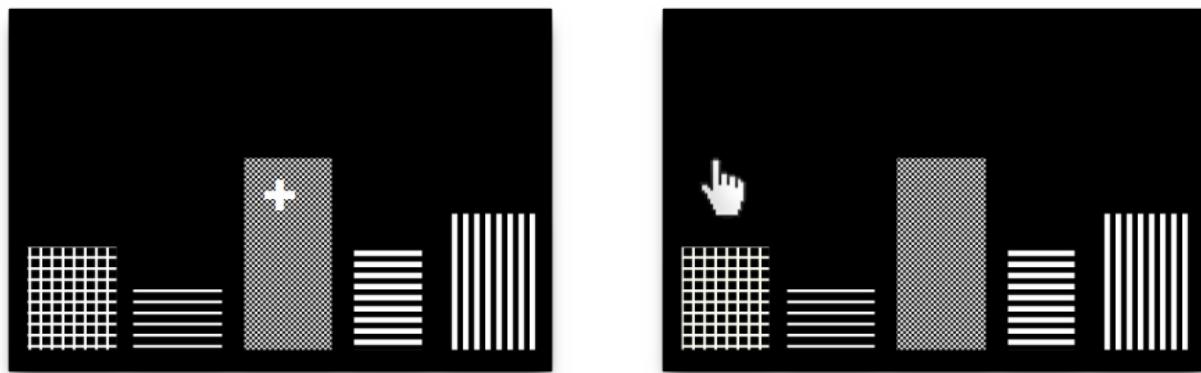


</div>

## title: Mouse Picking

# Mouse Picking

Mouse picking means that the user clicks an object in the scene to select it, or to interact with it otherwise. Games use picking to implement aiming and shooting, casting spells, picking up objects, selecting targets, dragging and moving objects, etc. Mouse picking can be done using fixed crosshairs, or using the mouse pointer.



See [Input Handling](#) for details on how to define the necessary input triggers, input mappings, and input listeners.

## Pick a Target Using Fixed Crosshairs

The following `pick target` input mapping implements an action that determines what a user clicked. It assumes that the mouse pointer is invisible and there are crosshairs painted in the center of the screen. It assumes that the user aims the crosshairs at an object in the scene and clicks. You use Ray Casting to identify the geometry that was picked by the user. Use this method together with a first-person flyCam.

1. Activate the first-person camera: `flyCam.setEnabled(true);`
2. Keep mouse pointer invisible using `inputManager.setCursorVisible(false)`.
3. Map the `pick target` action to a `MouseButtonTrigger`.
4. Implement the action in the Listener.

The following example rotates Spatial named “Red Box” or “Blue Box” when they are clicked. Modify this code to do whatever your game needs to do with the identified target (shoot it, take it, move it, etc).

```
private AnalogListener analogListener = new AnalogListener() {
 public void onAnalog(String name, float intensity, float tpf) {
 if (name.equals("pick target")) {
 // Reset results list.
 CollisionResults results = new CollisionResults();
 // Aim the ray from camera location in camera direction
 // (assuming crosshairs in center of screen).
 Ray ray = new Ray(cam.getLocation(), cam.getDirection());
 // Collect intersections between ray and all nodes in resu
 rootNode.collideWith(ray, results);
 // Print the results so we see what is going on
 for (int i = 0; i < results.size(); i++) {
 // For each "hit", we know distance, impact point, geom
 float dist = results.getCollision(i).getDistance();
 Vector3f pt = results.getCollision(i).getContactPoint();
 String target = results.getCollision(i).getGeometry().ge
 System.out.println("Selection #" + i + ": " + target + "
 }
 // 5. Use the results -- we rotate the selected geometry.
 if (results.size() > 0) {
 // The closest result is the target that the player pick
 Geometry target = results.getClosestCollision().getGeom
 // Here comes the action:
 if(target.getName().equals("Red Box"))
 target.rotate(0, - intensity, 0);
 else if(target.getName().equals("Blue Box"))
 target.rotate(0, intensity, 0);
 }
 } // else if ...
 }
};
```

## Pick a Target Using the Mouse Pointer

The following `pick target` input mapping implements an action that determines what a user clicked. It assumes that the mouse pointer is visible, and the user aims the cursor at an object in the scene. You use ray casting to determine the geometry that was picked by the user.

**Note:** Picking with a visible mouse pointer implies that your application can no longer use the default flyCam where the MouseAxisTrigger rotates the camera. You have to deactivate the flyCam mappings and provide custom mappings. Either different inputs rotate the camera, or the camera is fixed.

1. Map the `pick target` action to a MouseButtonTrigger.
2. Make the mouse pointer visible using `inputManager.setCursorVisible(true)`.
3. Remap the inputs for camera rotation, or deactivate camera rotation.
4. Implement the action in the Listener.

The following example rotates Spatial named “Red Box” or “Blue Box” when they are clicked. Modify this code to do whatever your game needs to do with the identified target (shoot it, take it, move it, etc).

```

private AnalogListener analogListener = new AnalogListener() {
 public void onAnalog(String name, float intensity, float tpf) {
 if (name.equals("pick target")) {
 // Reset results list.
 CollisionResults results = new CollisionResults();
 // Convert screen click to 3d position
 Vector2f click2d = inputManager.getCursorPosition();
 Vector3f click3d = cam.getWorldCoordinates(new Vector2f(click2d));
 Vector3f dir = cam.getWorldCoordinates(new Vector2f(click2d));
 // Aim the ray from the clicked spot forwards.
 Ray ray = new Ray(click3d, dir);
 // Collect intersections between ray and all nodes in results
 rootNode.collideWith(ray, results);
 // (Print the results so we see what is going on:)
 for (int i = 0; i < results.size(); i++) {
 // (For each "hit", we know distance, impact point, geometry)
 float dist = results.getCollision(i).getDistance();
 Vector3f pt = results.getCollision(i).getContactPoint();
 String target = results.getCollision(i).getGeometry().getLocalName();
 System.out.println("Selection #" + i + ": " + target + " at " + dist);
 }
 // Use the results -- we rotate the selected geometry.
 if (results.size() > 0) {
 // The closest result is the target that the player picked
 Geometry target = results.getClosestCollision().getGeometry();
 // Here comes the action:
 if (target.getName().equals("Red Box")) {
 target.rotate(0, -intensity, 0);
 } else if (target.getName().equals("Blue Box")) {
 target.rotate(0, intensity, 0);
 }
 }
 } // else if ...
 }
};

```

[documentation](#), [node](#), [ray](#), [click](#), [collision](#), [keyinput](#), [input](#)

</div>

## title: Multiple Camera Views

# Multiple Camera Views

You can split the screen and look into the 3D scene from different camera angles at the same time. E.g. you can have two rootnodes with different scene graphs, and two viewPorts, each of which can only see its own subset of the scene with its own subset of port-processing filters, so you get two very different views of the scene.

The packages used in this example are `com.jme3.renderer.Camera` and `com.jme3.renderer.ViewPort`. You can get the full sample code here: [TestMultiViews.java](#)

## How to resize and Position ViewPorts

The default viewPort is as big as the window. If you have several, they must be of different sizes, either overlapping or adjacent to one another. How do you tell jME which of the ViewPorts should appear where on the screen, and how big they should be?

Imagine the window as a 1.0f x 1.0f rectangle. The default cam's viewPort is set to

```
cam.setViewPort(0f, 1f, 0f, 1f);
```

This setting makes the ViewPort take up the whole rectangle.

The four values are read in the following order:

```
cam.setViewPort(x1,x2 , y1,y2);
```

- **X-axis** from left to right
- **Y-axis** upwards from bottom to top

Here are a few examples:

```
cam1.setViewPort(0.0f , 1.0f , 0.0f , 1.0f);
cam2.setViewPort(0.5f , 1.0f , 0.0f , 0.5f);
```

These viewport parameters are, (in this order) the left-right extend, and the bottom-top extend of a views's rectangle on the screen.

```
0.0 , 1.0 1.0 , 1.0
+-----+
|cam1 |
| |
| +----+
| | |
| |cam2 |
+-----+
0.0 , 0.0 1.0 , 0.0
```

Example: Cam2's rectangle is in the bottom right: It extends from mid ( $x1=0.5f$ ) bottom ( $y1=0.0f$ ), to right ( $x2=1.0f$ ) mid ( $y2=0.5f$ )

If you scale the views in a way so that the aspect ratio of a ViewPort is different than the window's aspect ratio, then the ViewPort appears distorted. In these cases, you must recreate (not clone) the ViewPort's cam object with the right aspect ratio. For example:

```
Camera cam5 = new Camera(100,100);
</div>
```

## Four-Time Split Screen

In this example, you create four views ( $2\times 2$ ) with the same aspect ratio as the window, but each is only half the width and height.

### Set up the First View

You use the preconfigured Camera `cam` and `viewPort` from `SimpleApplication` for the first view. It's in the bottom right.

```
cam.setViewPort(.5f, 1f, 0f, 0.5f); // Resize the viewPort to half
```

Optionally, place the main camera in the scene and rotate it in its start position.

```
cam.setLocation(new Vector3f(3.32f, 4.48f, 4.28f));
cam.setRotation(new Quaternion (-0.07f, 0.92f, -0.25f, -0.27f));
```

## Set Up Three More Views

Here is the outline for how you create the three other cams and viewPorts ([Full code sample is here.](#)) In the code snippet, `cam_n` stand for `cam_2` - `cam_4`, respectively, same for `view_n`.

1. Clone the first cam to reuse its settings
2. Resize and position the cam's viewPort with `setViewPort()`.
3. (Optional) Move the cameras in the scene and rotate them so they face what you want to see.
4. Create a ViewPort for each camera
5. Reset the camera's enabled statuses
6. Attach the Node to be displayed to this ViewPort.

The camera doesn't have to look at the `rootNode`, but that is the most common use case.

Here is the abstract code sample for camera `n`:

```
Camera cam_n = cam.clone();
cam_n.setViewPort(...); // resize the viewPort
cam_n.setLocation(new Vector3f(...));
cam_n.setRotation(new Quaternion(...));

ViewPort view_n = renderManager.createMainView("View of camera #n",
view_n.setClearEnabled(true);
view_n.attachScene(rootNode);
view_n.setBackgroundColor(ColorRGBA.Black);
```

To visualize what you do, use the following drawing of the viewport positions:

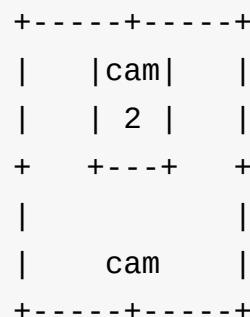
```
0.0 , 1.0 1.0 , 1.0
+-----+
| | |
|cam3 |cam4 |
+-----+
| | |
|cam2 |cam1 |
+-----+
0.0 , 0.0 1.0 , 0.0
```

This are the lines of code that set the four cameras to create a four-times split screen.

```
cam1.setViewPort(0.5f , 1.0f , 0.0f , 0.5f);
...
cam2.setViewPort(0.0f , 0.5f , 0.0f , 0.5f);
...
cam3.setViewPort(0.0f , 0.5f , 0.5f , 1.0f);
...
cam4.setViewPort(0.5f , 1.0f , 0.5f , 1.0f);
```

## Picture in Picture

The following code snippet sets up two views, one covers the whole screen, and the second is a small view in the top center.



```
// Setup first full-window view
cam.setViewPort(0f, 1f, 0f, 1f);
cam.setLocation(new Vector3f(3.32f, 4.48f, 4.28f));
cam.setRotation(new Quaternion(-0.07f, 0.92f, -0.25f, -0.27f));

// Setup second, smaller PiP view
Camera cam2 = cam.clone();
cam2.setViewPort(.4f, .6f, 0.8f, 1f);
cam2.setLocation(new Vector3f(-0.10f, 1.57f, 4.81f));
cam2.setRotation(new Quaternion(0.00f, 0.99f, -0.04f, 0.02f));
ViewPort viewPort2 = renderManager.createMainView("PiP", cam2);
viewPort2.setClearFlags(true, true, true);
viewPort2.attachScene(rootNode);
```

# ViewPort Settings

You can customize the camera and the viewPort of each view individually. For example, each view can have a different background color:

```
viewPort.setBackgroundColor(ColorRGBA.Blue);
```

You have full control to determine which Nodes the camera can see! It can see the full rootNode...

```
viewPort1.attachScene(rootNode);
```

... or you can give each camera a special node whose content it can see:

```
viewPort2.attachScene(spookyGhostDetectorNode);
```

[camera documentation](#)

</div>

# title: The jME3 Threading Model

## The jME3 Threading Model

jME3 is similar to Swing in that, for speed and efficiency, all changes to the scene graph must be made in a single update thread. If you make changes only in Control.update(), AppState.update(), or SimpleApplication.simpleUpdate(), this will happen automatically. However, if you pass work to another thread, you may need to pass results back to the main jME3 thread so that scene graph changes can take place there.

```
public void rotateGeometry(final Geometry geo, final Quaternion rot
 mainApp.enqueue(new Callable<Spatial>() {
 public Spatial call() throws Exception {
 return geo.rotate(rot);
 }
 });
}
```

Note that this example does not fetch the returned value by calling `get()` on the Future object returned from `enqueue()`. This means that the example method `rotateGeometry()` will return immediately and will not wait for the rotation to be processed before continuing.

If the processing thread needs to wait or needs the return value then `get()` or the other methods in the returned Future object such as `isDone()` can be used.

## Multithreading Optimization

First, make sure you know what [Application States](#) and [Custom Controls](#) are.

More complex games may feature complex mathematical operations or artificial intelligence calculations (such as path finding for several NPCs). If you make many time-intensive calls on the same thread (in the update loop), they will block one another, and thus slow down the game to a degree that makes it unplayable. If your game requires long running tasks, you should run them concurrently on separate threads, which speeds up the application considerably.

Often multithreading means having separate detached logical loops going on in parallel, which communicate about their state. (For example, one thread for AI, one Sound, one Graphics). However we recommend to use a global update loop for game logic, and do multithreading within that loop when it is appropriate. This approach scales way better to multiple cores and does not break up your code logic.

Effectively, each for-loop in the main update loop might be a chance for multithreading, if you can break it up into self-contained tasks.

## Java Multithreading

The `java.util.concurrent` package provides a good foundation for multithreading and dividing work into tasks that can be executed concurrently (hence the name). The three basic components are the Executor (supervises threads), Callable Objects (the tasks), and Future Objects (the result). You can [read about the concurrent package more here](#), I will give just a short introduction.

- A Callable is one of the classes that gets executed on a thread in the Executor. The object represents one of several concurrent tasks (e.g, one NPC's path finding task). Each Callable is started from the `updateloop` by calling a method named `call()`.
- The Executor is one central object that manages all your Callables. Every time you schedule a Callable in the Executor, the Executor returns a Future object for it.
- A Future is an object that you use to check the status of an individual Callable task. The Future also gives you the return value in case one is returned.

## Multithreading in jME3

So how do we implement multithreading in jME3?

Let's take the example of a Control that controls an NPC Spatial. The NPC Control has to compute a lengthy pathfinding operation for each NPC. If we would execute the operations directly in the `simpleUpdate()` loop, it would block the game each time a NPC wants to move from A to B. Even if we move this behaviour into the `update()` method of a dedicated NPC Control, we would still get annoying freeze frames, because it still runs on the same update loop thread.

To avoid slowdown, we decide to keep the pathfinding operations in the NPC Control, *but execute it on another thread*.

## Executor

You create the executor object in a global AppState (or the initSimpleApp() method), in any case in a high-level place where multiple controls can access it.

```
/* This constructor creates a new executor with a core pool size of
ScheduledThreadPoolExecutor executor = new ScheduledThreadPoolExecutor(4);
```

Pool size means the executor will keep four threads alive at any time. Having more threads in the pool means that more tasks can run concurrently. But a bigger pool only results in a speed gain if the PC can handle it! Allocating a pool that is uselessly large just wastes memory, so you need to find a good compromise: About the same to double the size of the number of cores in the computer makes sense.

!!! Executor needs to be shut down when the application ends, in order to make the process die properly In your simple application you can override the destroy method and shutdown the executor:

```
@Override
public void destroy() {
 super.destroy();
 executor.shutdown();
}
```

## Control Class Fields

In the NPC Control, we create the individual objects that the thread manipulates. In our example case (the pathfinding control), the task is about locations and path arrays, so we need the following variables:

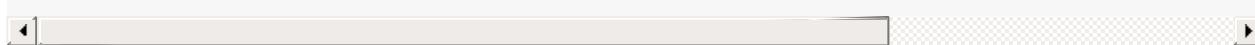
```
//The vector to store the desired location in:
Vector3f desiredLocation = new Vector3f();
//The MyWayList object that contains the result waylist:
MyWayList wayList = null;
//The future that is used to check the execution status:
Future future = null;
```

Here we also created the Future variable to track the state of this task.

## Control Update() Method

Next let's look at the update() call of the Control where the time-intensive task starts. In our example, the task is the `findway` Callable (which contains the pathfinding process). So instead of spelling out the pathfinding process in the Control's update() loop, we start the process via `future = executor.submit(findWay);`.

```
public void update(float tpf) {
 try{
 //If we have no waylist and not started a callable yet, do
 if(wayList == null && future == null){
 //set the desired location vector, after that we should
 //because it's being accessed on the other thread!
 desiredLocation.set(getGoodNextLocation());
 //start the callable on the executor
 future = executor.submit(findWay); // Thread starts
 }
 //If we have started a callable already, we check the status
 else if(future != null){
 //Get the waylist when its done
 if(future.isDone()){
 wayList = future.get();
 future = null;
 }
 else if(future.isCancelled()){
 //Set future to null. Maybe we succeed next time...
 future = null;
 }
 }
 }
 catch(Exception e){
 Exceptions.printStackTrace(e);
 }
 if(wayList != null){
 //.... Success! Let's process the wayList and move the NPC.
 }
}
```



Note how this logic makes its decision based on the Future object.

Remember not to mess with the class fields after starting the thread, because they are being accessed and modified on the new thread. In more obvious terms: You cannot change the “desired location” of the NPC while the path finder is calculating a different path. You have to cancel the current Future first.

## The Callable

The next code sample shows the Callable that is dedicated to performing the long-running task (here, wayfinding). This is the task that used to block the rest of the application, and is now executed on a thread of its own. You implement the task in the Callable always in an inner method named `call()`.

The task code in the Callable should be self-contained! It should not write or read any data of objects that are managed by the scene graph or OpenGL thread directly. Even reading locations of Spatial can be problematic! So ideally all data that is needed for the wayfinding process should be available to the new thread when it starts already, possibly in a cloned version so no concurrent access to the data happens.

In reality, you might need access to the game state. If you must read or write a current state from the scene graph, you must have a clone of the data in your thread. There are only two ways:

- Use the execution queue `application.enqueue()` to create a sub-thread that clones the info. Only disadvantage is, it may be slower.

The example below gets the `vector3f location` from the scene object `mySpatial` using this way.

- Create a separate World class that allows safe access to its data via synchronized methods to access the scene graph. Alternatively it can also internally use `application.enqueue()`.

The following example gets the object `Data data = myWorld.getData();` using this way.

These two ways are thread-safe, they don't mess up the game logic, and keep the Callable code readable.

```

// A self-contained time-intensive task:
private Callable<MyWayList> findWay = new Callable<MyWayList>(){
 public MyWayList call() throws Exception {

 //Read or write data from the scene graph -- via the execut
 Vector3f location = application.enqueue(new Callable<Vector
 public Vector3f call() throws Exception {
 //we clone the location so we can use the variable
 return mySpatial.getLocalTranslation().clone();
 }
 }).get();

 // This world class allows safe access via synchronized met
 Data data = myWorld.getData();

 //... Now process data and find the way ...

 return wayList;
 }
};

```

## Useful Links

High level description which describes how to manage the game state and the rendering in different threads - [link](#) Outdated link. A C++ example can be found at [link](#)

## Conclusion

The cool thing about this approach is that every entity creates one self-contained Callable for the Executor, and they are all executed in parallel. In theory, you can have one thread per entity without changing anything else but the settings of the executor.

[loop](#), [game](#), [performance](#), [state](#), [states](#), [documentation](#)  
 </div>

## **title:**

### **jME3 Networking Video Tutorials**

- Video: General Introduction to the networking series
- Video: Introduction to the internet
- Video: Introduction to the web
- Video: Cheating in games
- Video: Introduction to SpiderMonkey (Part 1/2)
- Video: Introduction to SpiderMonkey (Part 2/2)
- Video: Our first Networked game (Part 1/3)
- Video: Our first Networked game (Part 2/3)
- Video: Our first Networked game (Part 3/3)

# **title: SpiderMonkey: Multi-Player Networking**

## **SpiderMonkey: Multi-Player Networking**

This document introduces you to the SpiderMonkey networking [API](#). You use this [API](#) when you develop games where several players compete with one another in real time. A multi-player game is made up of several clients connecting to a server:

- The central server (one headless SimpleApplication) coordinates the game in the background.
- Each player runs a game client (a standard SimpleApplication) and connects to the central server.

Each Client keeps the Server informed about its player's moves and actions. The Server centrally maintains the game state and broadcasts the state info back to all connected clients. This network synchronization allows all clients to share the same game world. Each client then displays the game state to one player from this player's perspective.

## **SpiderMonkey API Overview**

The SpiderMonkey [API](#) is a set of interfaces and helper classes in the 'com.jme3.network' package. For most users, this package and the 'message' package is all they need to worry about. (The 'base' and 'kernel' packages only come into play when implementing custom network transports or alternate client/server protocols, which is now possible).

The SpiderMonkey [API](#) assists you in creating a Server, Clients, and Messages. Once a Server instance is created and started, the Server accepts remote connections from Clients, and you can send and receive Messages. Client objects represent the client-side of the client-server connection. Within the Server, these Client objects are referred to as HostedConnections. HostedConnections can hold application-defined client-specific session attributes that the server-side listeners and services can use to track player information, etc.

Seen from the Client	=	Seen from the Server
com.jme3.network.Client		com.jme3.network.HostedConnection

You can register several types of listeners to be notified of changes.

- MessageListeners on both the Client and the Server are notified when new messages

arrive. You can use MessageListeners to be notified about only specific types of messages.

- ClientStateListeners inform the Client of changes in its connection state, e.g. when the client gets kicked from the server.
- ConnectionListeners inform the Server about HostedConnection arrivals and removals, e.g. if a client joins or quits.
- ErrorListeners inform the Client about network exceptions that have happened, e.g. if the server crashes, the client throws a ConnectorException, this can be picked up so that the application can do something about it.

## Client and Server

### Creating a Server

The game server is a “headless” com.jme3.app.SimpleApplication:

```
public class ServerMain extends SimpleApplication {
 public static void main(String[] args) {
 ServerMain app = new ServerMain();
 app.start(JmeContext.Type.Headless); // headless type for server
 }
}
```

A `Headless` `SimpleApplication` executes the `simpleInitApp()` method and runs the update loop normally. But the application does not open a window, and it does not listen to user input. This is the typical behavior for a server application.

Create a `com.jme3.network.Server` in the `simpleInitApp()` method and specify a communication port, for example 6143.

```
public void simpleInitApp() {
 ...
 Server myServer = Network.createServer(6143);
 myServer.start();
 ...
}
```

When you run this app on a host, the server is ready to accept clients. Let's create a client next.

```
</div>
```

## Creating a Client

A game client is a standard com.jme3.app.SimpleApplication.

```
public class ClientMain extends SimpleApplication {
 public static void main(String[] args) {
 ClientMain app = new ClientMain();
 app.start(JmeContext.Type.Display); // standard display type
 }
}
```

A standard SimpleApplication in `Display` mode executes the `simpleInitApp()` method, runs the update loop, opens a window for the rendered video output, and listens to user input. This is the typical behavior for a client application.

Create a com.jme3.network.Client in the `simpleInitApp()` method and specify the servers IP address, and the same communication port as for the server, here 6143.

```
public void simpleInitApp() {
 ...
 Client myClient = Network.connectToServer("localhost", 6143);
 myClient.start();
 ...
}
```

The server address can be in the format “localhost” or “127.0.0.1” (for local testing), or an IP address of a remote host in the format “123.456.78.9”. In this example, we assume the server is running on the localhost.

When you run this client, it connects to the server.

```
</div>
```

## Getting Info About a Client

The server refers to a connected client as com.jme3.network.HostedConnection objects. The server can get info about clients as follows:

Accessor	Purpose
myServer.getConnections()	Server gets a collection of all connected HostedConnection objects (all connected clients).
myServer.getConnections().size()	Server gets the number of all connected HostedConnection objects (number of clients).
myServer.getConnection(0)	Server gets the first (0), second (1), etc, connected HostedConnection object (one client).

Your game can define its own game data based on whatever criteria you want, typically these include player ID and state. If the server needs to look up player/client-specific information, you can store this information directly on the HostedConnection object. The following examples read and write a custom Java object `Mystate` in the HostedConnection object `conn`:

Accessor	Purpose
<code>conn.setAttribute("MyState", new MyState());</code>	Server can change an attribute of the HostedConnection.
<code>MyState state = conn.getAttribute("MyState")</code>	Server can read an attribute of the HostedConnection.

## Messaging

### Creating Message Types

Each message represents data that you want to transmit between client and server. Common message examples include transformation updates or game actions. For each message type, create a message class that extends `com.jme3.network.AbstractMessage`. Use the `@Serializable` annotation from `com.jme3.network.serializing.Serializable` and create an empty default constructor. Custom constructors, fields, and methods are up to you and depend on the message data that you want to transmit.

```

@Serializable
public class HelloMessage extends AbstractMessage {
 private String hello; // custom message data
 public HelloMessage() {} // empty constructor
 public HelloMessage(String s) { hello = s; } // custom constructor
}

```

You must register each message type to the `com.jme3.network.serializing.Serializer`, in both server and client!

```
Serializer.registerClass(HelloMessage.class);
```

## Responding to Messages

After a Message was received, a Listener responds to it. The listener can access fields of the message, and send messages back, start new threads, etc. There are two listeners, one on the server, one on the client. For each message type, you implement the responses in either Listeners' `messageReceived()` method.

### ClientListener.java

Create one `ClientListener.java` and make it extend `com.jme3.network.MessageListener`.

```
public class ClientListener implements MessageListener<Client> {
 public void messageReceived(Client source, Message message) {
 if (message instanceof HelloMessage) {
 // do something with the message
 HelloMessage helloMessage = (HelloMessage) message;
 System.out.println("Client #" + source.getId() + " received: " + helloMessage.getMessage());
 } // else...
 }
}
```

For each message type, register a client listener to the client.

```
myClient.addMessageListener(new ClientListener(), HelloMessage.class);
```

### ServerListener.java

Create one `ServerListener.java` and make it extend `com.jme3.network.MessageListener`.

```
public class ServerListener implements MessageListener<HostedConnection>
 public void messageReceived(HostedConnection source, Message message) {
 if (message instanceof HelloMessage) {
 // do something with the message
 HelloMessage helloMessage = (HelloMessage) message;
 System.out.println("Server received '" + helloMessage.getMessage());
 } // else....
 }
}
```

For each message type, register a server listener to the server:

```
myServer.addMessageListener(new ServerListener(), HelloMessage.class)
```

## Creating and Sending Messages

Let's create a new message of type HelloMessage:

```
Message message = new HelloMessage("Hello World!");
```

Now the client can send this message to the server:

```
myClient.send(message);
```

Or the server can broadcast this message to all HostedConnection (clients):

```
Message message = new HelloMessage("Welcome!");
myServer.broadcast(message);
```

Or the server can send the message to a specific subset of clients (e.g. to HostedConnection conn1, conn2, and conn3):

```
myServer.broadcast(Filters.in(conn1, conn2, conn3), message);
```

Or the server can send the message to all but a few selected clients (e.g. to all HostedConnections but conn4):

```
myServer.broadcast(Filters.notEqualTo(conn4), message);
```

The last two broadcasting methods use com.jme3.network.Filters to select a subset of recipients. If you know the exact list of recipients, always send the messages directly to them using the Filters; avoid flooding the network with unnecessary broadcasts to all.

## Identification and Rejection

The ID of the Client and HostedConnection are the same at both ends of a connection. The ID is given out authoritatively by the Server.

```
... myClient.getId() ...
```

A server has a game version and game name property. Each client expects to communicate with a server with a certain game name and version. Test first whether the game name matches, and then whether game version matches, before sending any messages! If they do not match, SpiderMoney will reject it for you, you have no choice in the matter. This is so the client and server can avoid miscommunication.

Typically, your networked game defines its own attributes (such as player ID) based on whatever criteria you want. If you want to look up player/client-specific information beyond the game version, you can set this information directly on the Client/HostedConnection object (see Getting Info About a Client).

</div>

## Closing Clients and Server Cleanly

### Closing a Client

You must override the client's `destroy()` method to close the connection cleanly when the player quits the client:

```
@Override
public void destroy() {
 ... // custom code
 myClient.close();
 super.destroy();
}
```

## Closing a Server

You must override the server's `destroy()` method to close the connection when the server quits:

```
@Override
public void destroy() {
 ... // custom code
 myServer.close();
 super.destroy();
}
```

## Kicking a Client

The server can kick a `HostedConnection` to make it disconnect. You should provide a String with further info (an explanation to the user what happened, e.g. "Shutting down for maintenance") for the server to send along. This info message can be used (displayed to the user) by a `ClientStateListener`. (See below)

```
conn.close("We kick cheaters.");
```

## Listening to Connection Notification

The server and clients are notified about connection changes.

### ClientStateListener

The `com.jme3.network.ClientStateListener` notifies the Client when the Client has fully connected to the server (including any internal handshaking), and when the Client is kicked (disconnected) from the server.

The ClientStateListener when it receives a network exception applies the default close action. This just stops the client and you'll have to build around it so your application knows what to do. If you need more control when a network exception happens and the client closes, you may want to investigate in a ErrorListener.

<b>ClientStateListener interface method</b>	<b>Purpose</b>
public void clientConnected(Client c){}	Implement here what happens as soon as this client has fully connected to the server.
public void clientDisconnected(Client c, DisconnectInfo info){}	Implement here what happens after the server kicks this client. For example, display the DisconnectInfo to the user.

First implement the ClientStateListener interface in the Client class. Then register it to myClient in MyGameClient's simpleInitApp() method:

```
myClient.addClientStateListener(this);
```

## ConnectionListener

The com.jme3.network.ConnectionListener notifies the Server whenever new HostedConnections (clients) come and go. The listener notifies the server after the Client connection is fully established (including any internal handshaking).

<b>ConnectionListener interface method</b>	<b>Purpose</b>
public void connectionAdded(Server s, HostedConnection c){}	Implement here what happens after a new HostedConnection has joined the Server.
public void connectionRemoved(Server s, HostedConnection c){}	Implement here what happens after a HostedConnection has left. E.g. a player has quit the game and the server removes his character.

First implement the ConnectionListener interface in the Server class. Then register it to myServer in MyGameServer's simpleInitApp() method.

```
myServer.addConnectionListener(this);
```

## ErrorListener

The com.jme3.network.ErrorListener is a listener for when network exception happens. This listener is built so that you can override the default actions when a network exception happens.

If you intend on using the default network mechanics, **don't** use this! If you do override this, make sure you add a mechanic that can close the client otherwise your client will get stuck open and cause errors.

ErrorListener interface method	Purpose
public void handleError(Client c, Throwable t){}	Implementen here what happens after a exception affects the network .

This interface was built for the client and server, but the code has never been put on the server to handle this listener.

First implement the ErrorListener interface in the client class. Then you need to register it to myClient in MyGameClients's simpleInitApp() method.

```
myClient.addErrorListener(this);
```

In the class that implements the ErrorListener, a method would of been added call handleError(Client s, Throwable t). Inside this method to get you started, you going to want to listen for an error. To do this you're going to want a bit of code like this.

```
if(t instanceof exception) {
 //Add your own code here
}
```

Replace **exception** part in the **if** statement for the type of exception that you would like it to handle.

</div>

## UDP versus TCP

SpiderMonkey supports both UDP (unreliable, fast) and TCP (reliable, slow) transport of messages.

```
message1.setReliable(true); // TCP
message2.setReliable(false); // UDP
```

- Choose reliable and slow transport for messages, if you want to make certain the message is delivered (resent) when lost, and if the order of a series of messages is relevant. E.g. game actions such as “1. wield weapon, 2. attack, 3. dodge”.
- Choose unreliable and fast transport for messages if the next message makes any previously delayed or lost message obsolete and synchronizes the state again. E.g. a series of new locations while walking.

## Important: Use Multi-Threading

**You cannot modify the scenegraph directly from the network thread.** A common example for such a modification is when you synchronize the player's position in the scene. You have to use Java Multithreading.

Multithreading means that you create a Callable. A Callable is a Java class representing any (possibly time-intensive) self-contained task that has an impact on the scene graph (such as positioning the player). You enqueue the Callable in the Executor of the client's OpenGL thread. The Callable ensures to executes the modification in sync with the update loop.

```
app.enqueue(callable);
```

Learn more about using [multithreading](#) in jME3 [here](#).

For general advice, see the articles [MultiPlayer Networking](#) and [Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization](#) by the Valve Developer Community.

```
</div>
```

## Troubleshooting

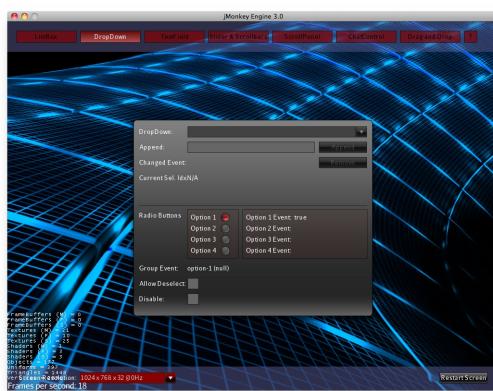
If you have set up a server in your home network, and the game clients cannot reach the server from the outside, it's time to learn about [port forwarding](#).

[documentation](#), [network](#), [spidermonkey](#)

```
</div>
```

## title: Creating JME3 User Interfaces with Nifty GUI

# Creating JME3 User Interfaces with Nifty GUI



You may want your players to press a button to save a game, you want a scrolling text field for highscores, a text label to display the score, drop-downs to select keymap preferences, or checkboxes to specify multi-media options. Usually you solve these tasks by using Swing controls. Although it is possible to embed a [jME3 canvas](#) in a [Swing GUI](#), a 3D game typically runs full-screen, or in a window of its own.

This document introduces you to [Nifty GUI](#), a Java library for building interactive graphical user interfaces (GUIs) for games or similar applications. [Nifty GUI](#) (the `de.lessvoid.nifty` package) is well integrated with jME3 through the `com.jme3.niftygui` package. You define the base [GUI](#) layout in XML, and control it dynamically from your Java code. The necessary JAR libraries are included in your jME3 download, you do not need to install anything extra. (Just make sure they are on the classpath.)

- [Video demo of Nifty GUI 1.3](#)

## Tutorial Overview

Learn to add a [Nifty GUI](#) to your jME3 game by going through this multi-part tutorial:

1. [Understand the Nifty GUI Concepts](#) described on this page.
2. [Browse this short list of Best Practices](#)

3. Lay out your graphical user interface:
  - [Lay out the GUI in XML](#) – or –
  - [Lay out the GUI in Java](#)
4. Integrate the GUI into the game:
  - [Overlay the User Interface Over the Screen](#) – or –
  - [Project the User Interface Onto a Texture](#)
5. Interact with the GUI from Java

## Must Know: Nifty GUI Concepts



Nifty GUIs are made up of the following **elements**:

- A Nifty GUI contains one or more **screens**.
  - Only one screen is visible at a time.
  - Name the first screen `start`. Name any others whatever you like.
  - Screens are [controlled by a Java Controller class](#).
- A screen contains one or more **layers**.
  - Layers are containers that impose alignment on their contents (vertical, horizontal, or centered)
  - Layers can overlap (z-order), and cannot be nested.
- A layer contains **panels**.
  - Panels are containers that impose alignment on their contents (vertical, horizontal, or centered)
  - Panels can be nested, and cannot overlap.
- A panel contains **images, text, or controls (buttons, etc)**.

## Resources

- [Browse this short list of Best Practices before you start](#)

## JME-Nifty Sample Code

- XML examples
  - [HelloJme.xml](#)
- Java examples
  - [TestNiftyGui.java](#)
  - [TestNiftyToMesh.java](#)
- jME3-ready version of the Nifty GUI 1.3 demo (sample code, Java)

- [NiftyGuiDemo.zip](#)
- Find more sample code in the [Nifty GUI source repository](#)

## External Documentation

Learn more from the NiftyGUI page:

- [Nifty GUI - the Manual](#)
- [Nifty 1.3 JavaDoc](#)
- [Nifty 1.3 Controls JavaDoc](#)
- [Forum post: Changing Text in Nifty GUIs programmatically](#)
- [Official Nifty GUI Editor](#)
- [New Nifty GUI Editor](#)
- [Nifty GUI with Groovy](#)

## Next Steps

Now that you understand the concepts and know where to find more information, learn how to lay out a simple graphical user interface. Typically, you start doing this in XML.

- [Lay out the GUI in XML](#) (recommended)
- [Lay out the GUI in Java](#) (optional)
- [Lay out the GUI in Editor](#) (experimental)

## Nifty Logging (Nifty 1.3.1)

If you want to disable the nifty log lines, add this code after you created nifty:

```
Logger.getLogger("de.lessvoid.nifty").setLevel(Level.SEVERE);
Logger.getLogger("NiftyInputEventHandlingLog").setLevel(Level.SEVERE)
```

[gui](#), [documentation](#), [nifty](#), [hud](#)  
</div>

## title: Nifty GUI - Best Practices

# Nifty GUI - Best Practices

This page is a short list of best practices that you should know of when starting to use Nifty [GUI](#). The JME3 tutorials focus on JME3-Nifty integration related details. You will find more features in the [Nifty GUI Manual](#).

1. [Nifty GUI Concepts](#)
2. [Nifty GUI Best Practices](#)
3. [Nifty GUI XML Layout](#) or [Nifty GUI Java Layout](#)
4. [Nifty GUI Overlay](#) or [Nifty GUI Projection](#)
5. [Nifty GUI Java Interaction](#)

## XML or Java?

You can build Nifty GUIs using XML or Java syntax. Which one should you choose? The XML and Java syntax are equivalent, so is it an either-or choice? Not quite. You typically use XML and Java together.

- Build your basic static UI layout using XML - it's cleaner to write and read.
- Use Java syntax to control the dynamic parts of the [GUI](#) at runtime - it's easier to interact with object-oriented code.
- **Example:** You design two UIs with slightly different XML layouts for mobile and desktop. If you use the same IDs for equivalent elements, your dynamic Java code works the same no matter which of the two base XML layout you use it on. This allows you to switch between a phone and a desktop UI by simply swapping one base XML file.

## Edit and Preview XML in the SDK

- Use the [jMonkeyEngine SDK](#) New File wizard to create a new XML file (from the [GUI](#) category, "Empty Nifty File").
- The [jMonkeyEngine SDK](#) includes an XML editor and a special previewer for Nifty [GUI](#) files.
- When you open an XML file, you can switch between XML Editor and [GUI](#) Preview mode.

Tip: The GUI category in the New File wizard also contains Nifty code samples.

## Validate the XML before loading

The [Nifty class](#) has `validateXml()` method that takes the same input XML argument as `fromXml()`. Nifty does not validate the XML by default, and will blow up in surprising ways if your XML does not conform to the schema. Adding the validation step will save you debugging time. You can validate right before loading, or in your unit tests.

## Use Code Completion

- Include the following XML schema in the first line of your NiftyGUI XML files

```
<?xml version="1.0" encoding="UTF-8"?>
<nifty xmlns="http://nifty-gui.sourceforge.net/nifty-1.3.xsd" xsi:schemaLocation="http://nifty-gui.sourceforge.net/nifty-1.3.xsd">
 <!-- Your IDE now tells you that one <screen></screen> element is missing -->
</nifty>
```



- Now your IDE (including the jMonkeyEngine SDK) will display extra info and do code completion for the Nifty GUI API.

## Use the ID String Right

- If you want to interact with an element, give it an ID (String).
- Use transparent ID-less panels as anonymous spacers.

## Sizing Layers and Panels

- Specify widths and heights in percent to allow the GUI to scale.
- Use `*` instead of a fixed value to make the element fill the remaining space automatically.
- Be cautious when specifying fixed sizes, and test the outcome in various resolutions.

## Colorcode for Clarity

## Screens, layers, and panels...

- ... can have an RGBA background color.

You can use temporary colors during the design phase to highlight which container is which.

- ... can be transparent.

In the finished GUI, screens, layers, and panels are typically transparent; the visible elements are the images, text fields, and controls, inside the panels.

During development (and during a tutorial), the following debug code makes all panels visible. This helps you arrange them and find errors.

```
nifty.setDebugOptionPanelColors(true);
```

Before the release, and during testing, set the debug view to false again.

```
</div>
```

## **title: Relucri's Nifty Editor**

### **Relucri's Nifty Editor**

Forum:

<http://hub.jmonkeyengine.org/forum/topic/nifty-gui-editor/>

Videos:

Download: <http://niftyeditor.it/>

Manual : <http://niftyeditor.it/home/>

### **Atomix's Nifty Editor**

Base on recluci's Editor, ported to Netbean plugin, with more features included.

### **Why port?**

To integrate better with the SDK toolset.

Notice that even it from the same author of Atom framework but it can work separately and not depend on Atom framework. The Extended version of this Nifty Editor, which depend on Atom's component is called TeeHeeGUI!

Forum:

Videos:

Download:

The below Manual belong to Atomix's Nifty Editor!!! You should find the Manual for the orginal Relucri's Nifty Editor above

</div>

## **Usage**

</div>

## **Open XML file**

</div>

## **Link to Style Def file**

</div>

## **Link to Control Def file**

</div>

## **Style pack file**

</div>

## **Drag and drop Components**

</div>

## **Java side**

</div>

## **Wire up . Event & Control & Binding**

</div>

## **Prototyping your UI**

</div>

## title: Interacting with the GUI from Java

# Interacting with the GUI from Java

1. [Nifty GUI Concepts](#)
2. [Nifty GUI Best Practices](#)
3. [Nifty GUI XML Layout or Nifty GUI Java Layout](#)
4. [Nifty GUI Overlay or Nifty GUI Projection](#)
5. [\*\*Nifty GUI Java Interaction\*\*](#)

In the previous parts of the tutorial, you created a two-screen user interface. But it is still static, and when you click the buttons, nothing happens yet. The purpose of the GUI is to communicate with your Java classes: Your game needs to know what the users clicked, which settings they chose, which values they entered into a field, etc. Similarly, the user needs to know what the currently game state is (score, health, etc).

## Connect GUI to Java Controller

To let a Nifty screen communicate with the Java application, you register a `ScreenController` to every NiftyGUI screen. You create a ScreenController by creating a Java class that implements the `de.lessvoid.nifty.screen.ScreenController` interface and its abstract methods.

Create an AppState **MyStartScreen.java** file in your package. ( Rightclick on your package → New → Other... → JME3 Classes → New AppState)

**Pro Tip:** Since you are writing a jME3 application, you can additionally make the ScreenController class extend the `AbstractAppState` class! This gives the ScreenController access to the application object and to the update loop!

Now add **implements ScreenController** to `public class MyStartScreen extends AbstractAppState{` and add `import de.lessvoid.nifty.screen.ScreenController;`

Continue with adding:

```

import de.lessvoid.nifty.screen.Screen;

...

public void bind(Nifty nifty, Screen screen) {
 throw new UnsupportedOperationException("Not supported yet.");
}

public void onStartScreen() {
 throw new UnsupportedOperationException("Not supported yet.");
}

public void onEndScreen() {
 throw new UnsupportedOperationException("Not supported yet.");
}

```

```

/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package mygame;

import com.jme3.app.Application;
import com.jme3.app.state.AbstractAppState;
import com.jme3.app.state.AppStateManager;
import de.lessvoid.nifty.Nifty;
import de.lessvoid.nifty.screen.Screen;
import de.lessvoid.nifty.screen.ScreenController;

public class MyStartScreen extends AbstractAppState implements ScreenController {

 @Override
 public void initialize(AppStateManager stateManager, Application app) {
 super.initialize(stateManager, app);
 //TODO: initialize your AppState, e.g. attach spatlials to renderers
 //this is called on the OpenGL thread after the AppState has been initialized
 }
}

```

```

@Override
public void update(float tpf) {
 //TODO: implement behavior during runtime
}

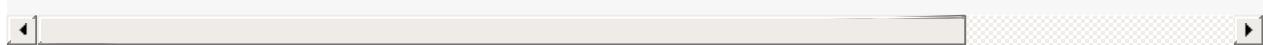
@Override
public void cleanup() {
 super.cleanup();
 //TODO: clean up what you initialized in the initialize method
 //e.g. remove all spatlials from rootNode
 //this is called on the OpenGL thread after the AppState has been updated
}

public void bind(Nifty nifty, Screen screen) {
 throw new UnsupportedOperationException("Not supported yet.");
}

public void onStartScreen() {
 throw new UnsupportedOperationException("Not supported yet.");
}

public void onEndScreen() {
 throw new UnsupportedOperationException("Not supported yet.");
}

```



The name and package of your custom ScreenController class (here `mygame.MyStartScreen`) goes into the controller parameter of the respective XML screen it belongs to. For example:

```

<nifty>
 <screen id="start" controller="mygame.MyStartScreen">
 <!-- layer and panel code ... -->
 </screen>
</nifty>

```

Or the same in a Java syntax, respectively:

```
nifty.addScreen("start", new ScreenBuilder("start") {{
 controller(new mygame.MyStartScreen())}});
```

Now the Java class `MyStartScreen` and this GUI screen (`start`) are connected. For this example you can also connect the `hud` screen to `MyStartScreen`.

## Make GUI and Java Interact

In most cases, you will want to pass game data in and out of the `ScreenController`. Note that you can pass any custom arguments from your Java class into your `ScreenController` constructor (`public MyStartScreen(GameData data) {}`).

Use any combination of the three following approaches to make Java classes interact with the GUI.

### GUI Calls a Void Java Method

This is how you respond to an GUI interaction such as clicks in XML GUIs:

1. Add `visibleToMouse="true"` to the parent element!
2. Embed the `<interact />` element into the parent element.
3. Specify the Java methods that you want to call when the users performs certain actions, such as clicking.

Example: `<interact onclick="startGame(hud)" />`

Or this is how you respond to an GUI interaction such as clicks in Java GUIs:

1. Add `visibleToMouse(true);` to the parent element!
2. Embed one of the `interact...()` elements into the parent element
3. Specify the Java method that you want to call after the interaction.

Example: `interactonClick("startGame(hud)");`

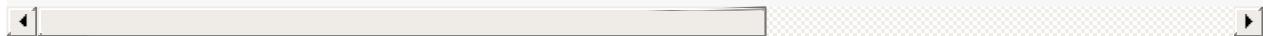
In the following example, we call the `startGame()` method when the player clicks the Start button, and `quitGame()` when the player clicks the Quit button.

```

<panel id="panel_bottom_left" height="50%" width="50%" vali
 <control name="button" label="Start" id="StartButton" ali
 visibleToMouse="true" >
 <interact onClick="startGame(hud)"/>
 </control>
 </panel>

<panel id="panel_bottom_right" height="50%" width="50%" vali
 <control name="button" label="Quit" id="QuitButton" alignr
 visibleToMouse="true" >
 <interact onClick="quitGame()"/>
 </control>
 </panel>

```



Or the same in a Java syntax, respectively:

```

control(new ButtonBuilder("StartButton", "Start") {{
 alignCenter();
 valignCenter();
 height("50%");
 width("50%");
 visibleToMouse(true);
 interactOnClick("startGame(hud)");
}});

...
control(new ButtonBuilder("QuitButton", "Quit") {{
 alignCenter();
 valignCenter();
 height("50%");
 width("50%");
 visibleToMouse(true);
 interactOnClick("quitGame()");
}});

```

Back in the `MyStartScreen` class, you specify what the `startGame()` and `quitGame()` methods do. As you see, you can pass String arguments (here `hud`) in the method call. You also see that you have access to the `app` object.

```

public class MyStartScreen implements ScreenController {
 ...

 /** custom methods */
 public void startGame(String nextScreen) {
 nifty.gotoScreen(nextScreen); // switch to another screen
 // start the game and do some more stuff...
 }

 public void quitGame() {
 app.stop();
 }

 ...
}

```

The `startGame()` example simply switches the GUI to the `hud` screen when the user clicks Start. Of course, in a real game, you would perform more steps here: Load the game level, switch to in-game input and navigation handling, set a custom `running` boolean to true, attach custom in-game AppStates – and lots more.

The `quitGame()` example shows that you have access to the application `app` object because you made the `ScreenController` extend `AbstractAppState`. (If you're creating code from this example, note that you'll need to make sure `app` is initialized before you can successfully call its methods.)

## GUI Gets Return Value from Java Method

When the Nifty GUI is initialized, you can get data from Java. In this example, the Java class `get playerName()` in `MyStartScreen` defines the Text that is displayed in the textfield before the words `'s Cool Game`.

First define a Java method in the screen controller, in this example, `get playerName()`.

```

public class MySettingsScreen implements ScreenController {
 ...
 public String getPlayerName(){
 return System.getProperty("user.name");
 }
}

```

Nifty uses  `${CALL.getPlayerName()}`` to get the return value of the `getPlayerName()` method from your ScreenController Java class.

```
<text text="${CALL.getPlayerName()}'s Cool Game" font="Interface/Fc
```

Or the same in a Java syntax, respectively:

```
text(new TextBuilder() {{
 text("${CALL.getPlayerName()}'s Cool Game");
 font("Interface/Fonts/Default.fnt");
 height("100%");
 width("100%");
}});
```

You can use this for Strings and numeric values (e.g. when you read settings from a file, you display the results in the [GUI](#)) and also for methods with side effects.

## Java Modifies Nifty Elements and Events

You can also alter the appearance and functions of your nifty elements from Java. Make certain that the element that you want to alter has its  `id="name"` attribute set, so you can identify and address it.

Here's an example of how to change an image called  `playerhealth` :

```
// load or create new image
NiftyImage img = nifty.getRenderEngine().createImage("Interface/Ima
// find old image
Element niftyElement = nifty.getCurrentScreen().findElementByName(")
// swap old with new image
niftyElement.getRenderer(ImageRenderer.class).setImage(img);
```

The same is valid for other elements, for example a text label "score":

```
// find old text
Element niftyElement = nifty.getCurrentScreen().findElementByName(" // swap old with new text
niftyElement.getRenderer(TextRenderer.class).setText("124");
```

Similarly, to change the onClick() event of an element, create an `ElementInteraction` object:

```
Element niftyElement = nifty.getCurrentScreen().findElementByName(" niftyElement.getElementInteraction().getPrimary().setOnMouseOver(ne
```

For this to work, there already needs to be a (possibly inactive) `<interact />` tag inside your xml element:

```
<interact onClick="doNothing()">
```

## Next Steps

You're done with the basic Nifty GUI for jME3 tutorial. You can proceed to advanced topics and learn how add controls and effects:

- [Nifty GUI Scenarios](#)
- [Nifty GUI - the Manual](#)

[gui](#), [documentation](#), [input](#), [control](#), [hud](#), [nifty](#)

</div>

--- title: Laying Out the GUI in Java ---

# Laying Out the GUI in Java

- 1.
1.
  1. [Nifty GUI Concepts](#)
  1.
    1. [Nifty GUI Best Practices](#)
    1.
      1. [Nifty GUI XML Layout or Nifty GUI Java Layout](#)
      1.
        1. [Nifty GUI Overlay or Nifty GUI Projection](#)
        1.
          1. [Interact with the GUI from Java](#)

**Work in progress** You can “draw” the GUI to the screen by writing Java code – alternatively to using XML. Typically you lay out the static base GUI in XML, and use Java commands if you need to change the GUI dynamically at runtime. In theory, you can also lay out the whole GUI in Java (but we don’t cover that here).

## Sample Code

Sample project

\*

\*

- **Original Source Code:** </nifty-default-controls-examples/trunk/src/main/java/de/lessvoid/nifty/examples/.>
- **Download demo project:** <http://files.seapegasus.org/NiftyGuiDemo.zip> (jme3-ready) \\ The full demo ZIP is based on `de.lessvoid.nifty.examples.controls.ControlsDemo.java`.

- 1.
  - 1.
    1. The demo is a SimpleApplication-based game (use e.g. the BasicGame template in the jMonkeyEngine SDK).
  - 1.
    1. Copy images and sound files into your project's `assets/Interface/` directory. (In this example, I copied them from `nifty-default-controls-examples/trunk/src/main/resources/` to `assets/Interface/` ).
  - 1.
  - 1. Make sure to use paths relative to your project's `assets/` directory.
1. \*
2. \*
- E.g. for `.fnt/.png/.jpg` files use `filename("Interface.yang.png");` (not `filename("yang.png");` ).

```
*

* E.g. for .wav/.ogg files use `filename("Interface/sounds/gong.wav");` (not `filename(`
[] [] []
```

Just so you get a quick picture what Nifty GUI's Java Syntax looks like, here is the most basic example. It creates a screen with a layer and a panel that contains a button.

```
package mygame;

import com.jme3.app.SimpleApplication;
import com.jme3.niftygui.NiftyJmeDisplay;
import de.lessvoid.nifty.Nifty;
import de.lessvoid.nifty.builder.ScreenBuilder;
import de.lessvoid.nifty.builder.LayerBuilder;
import de.lessvoid.nifty.builder.PanelBuilder;
import de.lessvoid.nifty.controls.button.builder.ButtonBuilder;
import de.lessvoid.nifty.screen.DefaultScreenController;

/**
 * @author iamcreasy
 */
public class Main extends SimpleApplication {
```

```

public static void main(String[] args) {
 Main app = new Main();
 app.start();
}

@Override
public void simpleInitApp() {
 NiftyJmeDisplay niftyDisplay = new NiftyJmeDisplay(
 assetManager, inputManager, audioRenderer, guiViewPort);
 Nifty nifty = niftyDisplay.getNifty();
 guiViewPort.addProcessor(niftyDisplay);
 flyCam.setDragToRotate(true);

 nifty.loadStyleFile("nifty-default-styles.xml");
 nifty.loadControlFile("nifty-default-controls.xml");

 // <screen>
 nifty.addScreen("Screen_ID", new ScreenBuilder("Hello Nifty Screen"){{{
 controller(new DefaultScreenController()); // Screen properties

 // <layer>
 layer(new LayerBuilder("Layer_ID") {{{
 childLayoutVertical(); // layer properties, add more...

 // <panel>
 panel(new PanelBuilder("Panel_ID") {{{
 childLayoutCenter(); // panel properties, add more...

 // GUI elements
 control(new ButtonBuilder("Button_ID", "Hello Nifty"){{{
 alignCenter();
 valignCenter();
 height("5%");
 width("15%");
 }});

 // ... add more GUI elements here
 }});

 // </panel>
 }});

 // </layer>
 }}}.build(nifty));
 // </screen>

 nifty.gotoScreen("Screen_ID"); // start the screen
}
}

```

## Implement Your GUI Layout



In this tutorial, you recreate the same screen as in the Nifty GUI XML example.

Create an Screen.Java file in the `assets/Interfaces/` directory of your project. One Java file can contain several, or even all screens. As a reminder: Nifty displays one screen at a time; a screen contains several layers on top of one another; each layer contains panels that are embedded into another; the panels contain the actual content (text, images, or controls).

## Make Screens

The following minimal Java file contains a start screen and a HUD screen. (Neither has been defined yet.)

```
nifty.addScreen("start", new ScreenBuilder("start")){
 controller(new DefaultScreenController());
 // <!-- ... -->
}.build(nifty));

nifty.addScreen("hud", new ScreenBuilder("hud")){
 controller(new DefaultScreenController());
 // <!-- ... -->
}.build(nifty));
```

Every Nifty GUI must have a start screen. The others (in this example, the HUD screen) are optional.

## Make Layers

The following Java code shows how we add layers to the start screen and HUD screen:

```
nifty.addScreen("start", new ScreenBuilder("start"){{
 controller(new DefaultScreenController());

 // layer added
 layer(new LayerBuilder("background") {{
 childLayoutCenter();
 backgroundColor("#000f");

 // <!-- ... -->
 }});

 layer(new LayerBuilder("foreground") {{
 childLayoutVertical();
 backgroundColor("#0000");

 // <!-- ... -->
 }});
 // layer added

}}.build(nifty));
```

Repeat the same, but use

```
nifty.addScreen("hud", new ScreenBuilder("hud"){{
```

for the HUD screen.

In a layer, you can now add panels and arrange them. Panels are containers that mark the areas where you want to display text, images, or controls (buttons etc) later.

## Make Panels

A panel is the inner-most container (that will contain the actual content: text, images, or controls). You place panels inside layers. The following panels go into in the `start` screen:

```
nifty.addScreen("start", new ScreenBuilder("start") {{
 controller(new DefaultScreenController());
 layer(new LayerBuilder("background") {{
 childLayoutCenter();
 backgroundColor("#000f");
 // <!-- ... -->
 }});

 layer(new LayerBuilder("foreground") {{
 childLayoutVertical();
 backgroundColor("#0000");

 // panel added
 }});
```

```

panel(new PanelBuilder("panel_top") {{
 childLayoutCenter();
 alignCenter();
 backgroundColor("#f008");
 height("25%");
 width("75%");
}});

panel(new PanelBuilder("panel_mid") {{
 childLayoutCenter();
 alignCenter();
 backgroundColor("#0f08");
 height("50%");
 width("75%");
}});

panel(new PanelBuilder("panel_bottom") {{
 childLayoutHorizontal();
 alignCenter();
 backgroundColor("#00f8");
 height("25%");
 width("75%");

 panel(new PanelBuilder("panel_bottom_left") {{
 childLayoutCenter();
 valignCenter();
 backgroundColor("#44f8");
 height("50%");
 width("50%");
}});

 panel(new PanelBuilder("panel_bottom_right") {{
 childLayoutCenter();
 valignCenter();
 backgroundColor("#88f8");
 height("50%");
 width("50%");
}});
}}); // panel added
}});

}}.build(nifty));

```

The following panels go into in the `hud` screen:

```

nifty.addScreen("hud", new ScreenBuilder("hud") {{
 controller(new DefaultScreenController());

 layer(new LayerBuilder("background") {{
 childLayoutCenter();
 backgroundColor("#000f");

```

```

// <!-- ... -->
});

layer(new LayerBuilder("foreground") {{
 childLayoutHorizontal();
 backgroundColor("#0000");

 // panel added
 panel(new PanelBuilder("panel_left") {{
 childLayoutVertical();
 backgroundColor("#0f08");
 height("100%");
 width("80%");
 // <!-- spacer -->
 }});
}

panel(new PanelBuilder("panel_right") {{
 childLayoutVertical();
 backgroundColor("#00f8");
 height("100%");
 width("20%");

 panel(new PanelBuilder("panel_top_right1") {{
 childLayoutCenter();
 backgroundColor("#00f8");
 height("15%");
 width("100%");
 }});
}

panel(new PanelBuilder("panel_top_right2") {{
 childLayoutCenter();
 backgroundColor("#44f8");
 height("15%");
 width("100%");
}});

panel(new PanelBuilder("panel_bot_right") {{
 childLayoutCenter();
 valignCenter();
 backgroundColor("#88f8");
 height("70%");
 width("100%");
}});
}}); // panel added
}});

}}.build(nifty));

```

Try the sample. Remember to activate a screen using `nifty.gotoScreen("start");` or `hud` respectively. The result should look as follows:



# Adding Content to Panels

See also [Layout Introduction](#) on the Nifty GUI site.

## Add Images

The start-background.png image is a fullscreen background picture. In the `start` screen, add the following image element:

```
nifty.addScreen("start", new ScreenBuilder("start") {{
 controller(new DefaultScreenController());
 layer(new LayerBuilder("background") {{
 childLayoutCenter();
 backgroundColor("#000f");

 // add image
 image(new ImageBuilder() {{
 filename("Interface/tutorial/start-background.png");
 }});
 }});

}});
```

The hud-frame.png image is a transparent frame that we use as HUD decoration. In the `hud` screen, add the following image element:

```
nifty.addScreen("hud", new ScreenBuilder("hud") {{
 controller(new DefaultScreenController());

 layer(new LayerBuilder("background") {{
 childLayoutCenter();
 backgroundColor("#000f");

 // add image
 image(new ImageBuilder() {{
 filename("Interface/tutorial/hud-frame.png");
 }});
 }});

}});
```

The face1.png image is an image that you want to use as a status icon. In the `hud` screen's `foreground` layer, add the following image element:

```

 panel(new PanelBuilder("panel_top_right2") {{
 childLayoutCenter();
 backgroundColor("#44f8");
 height("15%");
 width("100%");

 // add image
 image(new ImageBuilder() {{
 filename("Interface/tutorial/face1.png");
 valignCenter();
 alignCenter();
 height("50%");
 width("30%");
 }});
 }});

 });
}

```

This image is scaled to use 50% of the height and 30% of the width of its container.

## Add Static Text

The game title is a typical example of static text. In the `start` screen, add the following text element:

```

// panel added
panel(new PanelBuilder("panel_top") {{
 childLayoutCenter();
 alignCenter();
 backgroundColor("#f008");
 height("25%");
 width("75%");

 // add text
 text(new TextBuilder() {{
 text("My Cool Game");
 font("Interface/Fonts/Default.fnt");
 height("100%");
 width("100%");
 }});
}}
```

For longer pieces of static text, such as an introduction, you can use `wrap="true"`. Add the following text element to the `Start` screen :

```

 panel(new PanelBuilder("panel_mid") {{
 childLayoutCenter();
 alignCenter();
 backgroundColor("#0f08");
 height("50%");
 width("75%");
 // add text
 text(new TextBuilder() {{
 text("Here goes some text describing the game and the rules and stuff
 "Incidentally, the text is quite long and needs to wrap at the e
 font("Interface/FONTs/Default.fnt");
 wrap(true);
 height("100%");
 width("100%");
 }});

 }});

 });

```

The font used is jME3's default font "Interface/FONTs/Default.fnt" which is included in the jMonkeyEngine.JAR. You can add your own fonts to your own `assets/Interface` directory.

## Add Controls

Before you can use any control, you must load a Control Definition first. Add the following two lines *before* your screen definitions:

```

nifty.loadStyleFile("nifty-default-styles.xml");
nifty.loadControlFile("nifty-default-controls.xml");

```

## Label Control

Use label controls for text that you want to edit dynamically from Java. One example for this is the score display. In the `hud` screen's `foreground` layer, add the following text element:

```
panel(new PanelBuilder("panel_top_right1") {{
 childLayoutCenter();
 backgroundColor("#00f8");
 height("15%");
 width("100%");

 control(new LabelBuilder(){{
 color("#000");
 text("123");
 width("100%");
 height("100%");
 }});
```

Note that the width and height do not scale the bitmap font, but make indirectly certain it is centered. If you want a different size for the font, you need to provide an extra bitmap font (they come with fixes sizes and don't scale well).

## Button Control

Our GUI plan asks for two buttons on the start screen. You add the Start and Quit buttons to the bottom panel of the `start` screen using the `<control>` element:

```
panel(new PanelBuilder("panel_bottom_left") {{
 childLayoutCenter();
 valignCenter();
 backgroundColor("#44f8");
 height("50%");
 width("50%");

 // add control
 control(new ButtonBuilder("StartButton", "Start") {{
 alignCenter();
 valignCenter();
 height("50%");
 width("50%");
 }});

});

panel(new PanelBuilder("panel_bottom_right") {{
 childLayoutCenter();
 valignCenter();
 backgroundColor("#88f8");
 height("50%");
 width("50%");

 // add control
 control(new ButtonBuilder("QuitButton", "Quit") {{
 alignCenter();
 valignCenter();
 height("50%");
 width("50%");
 }});

});
```

Note that these controls don't do anything yet – we'll get to that soon.

## Other Controls

Nifty additionally offers many customizable controls such as check boxes, text fields, menus, chats, tabs, ... See also [Elements](#) on the Nifty GUI site.

## Intermediate Result

When you preview this code in the jMonkeyEngine SDK, our tutorial demo should looks as follows: A start screen with two buttons, and a game screen with a simple HUD frame and a blue cube (which stands for any jME3 game content).

**Tip:** Remove all lines that set background colors, you only needed them to see the arrangement.



## Nifty Java Settings

Before initializing the nifty screens, you set up properties and register media.

```
Nifty MethodDescriptionregisterSound("mysound",
"Interface/abc.wav");registerMusic("mymusic",
"Interface/xyz.ogg");registerMouseCursor("mypointer", "Interface/abc.png", 5,
4);registerEffect(?);?setDebugOptionPanelColors(true);Highlight all panels, makes it easier
to arrange them.
```

Example:

```
nifty.registerMouseCursor("hand", "Interface/mouse-cursor-hand.png", 5, 4);
```

## Next Steps

Integrate the GUI into the game. Typically, you will overlay the GUI.

- [Nifty GUI Overlay](#) (recommended)
- [Nifty GUI Projection](#) (optional)

## title: Integrating Nifty GUI: Overlay

# Integrating Nifty GUI: Overlay

1. [Nifty GUI Concepts](#)
2. [Nifty GUI Best Practices](#)
3. [Nifty GUI XML Layout or Nifty GUI Java Layout](#)
4. **Nifty GUI Overlay or Nifty GUI Projection**
5. [Interact with the GUI from Java](#)



Typically, you define a key (for example escape) that switches the GUI on and off. The GUI can be a StartScreen, OptionsScreen, CharacterCreationScreen, etc. While the GUI is up, you pause the running game, and then overlay the GUI. You also must switch to a different set of user inputs while the game is paused, so the player can use the mouse pointer and keyboard to interact with the GUI.

You can also [project](#) the GUI as a texture onto a mesh texture (but then you cannot click to select). On this page, we look at the overlay variant, which is more commonly used in games.

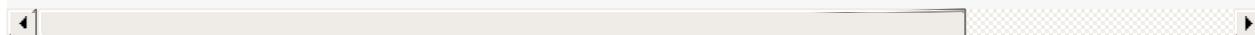
## Sample Code

- [TestNiftyGui.java](#)

## Overlaying the User Interface Over the Screen

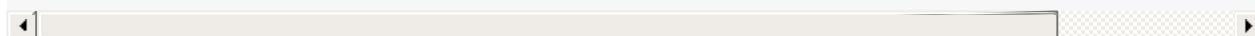
This code shows you how to overlay anything on the screen with the GUI. This is the most common usecase.

```
NiftyJmeDisplay niftyDisplay = new NiftyJmeDisplay(
 assetManager, inputManager, audioRenderer, guiViewPort);
/** Create a new NiftyGUI object */
Nifty nifty = niftyDisplay.getNifty();
/** Read your XML and initialize your custom ScreenController */
nifty.fromXml("Interface/tutorial/step2/screen.xml", "start");
// nifty.fromXml("Interface/helloworld.xml", "start", new MySettingsScreen());
// attach the Nifty display to the gui view port as a processor
guiViewPort.addProcessor(niftyDisplay);
// disable the fly cam
flyCam.setDragToRotate(true);
```



Currently you do not have a ScreenController – we will create one in the next exercise. As soon as you have a screen controller, you will use the commented variant of the XML loading method:

```
nifty.fromXml("Interface/helloworld.xml", "start", new MySettingsScreen());
```



The `MySettingsScreen` class is a custom `de.lessvoid.nifty.screen.ScreenController` in which you will implement your GUI behaviour.

If you have many screens or you want to keep them organized in separate files there is a method available that will just load an additional XML file. The content of the files are simply added to whatever XML data has been loaded before.

```
nifty.addXml("Interface/mysecondscreen.xml");
```



## Next Steps

Now that you have layed out and integrated the GUI in your app, you want to respond to user input and display the current game. Time to create a ScreenController!

- [Interact with the GUI from Java](#)

[gui](#), [documentation](#), [nifty](#), [hud](#)

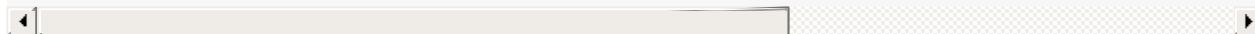
</div>

## title: Nifty GUI: Create a PopUp Menu

# Nifty GUI: Create a PopUp Menu

Even though you create and populate the popup menu in Java, you still need a “placeholder” in your XML file. The popup element needs to be placed *outside* of any screen!

```
<useControls filename="nifty-default-controls.xml"/>
...
<popup id="niftyPopupMenu" childLayout="absolute-inside"
 controller="ControllerOfYourChoice" width="10%">
 <interact onClick="closePopup()" onSecondaryClick="closePopup()"\>
 <control id="#menu" name="niftyMenu" />
 </popup>
...
```



A brief explanation of some the attributes above:

- The popup id is used within your Java code so that nifty knows which popup placeholder to create.
- The Controller tells Nifty which Java class handles MenuItemActivatedEvent.
- The on(Secondary/Tertiary)Click tells Nifty to close the popup if the user clicks anywhere except on the menu items (in this example; you have to define the closePopup()-method yourself, in the screen controller)
- The control id is used by the Java class to define a control type (i.e. Menu)

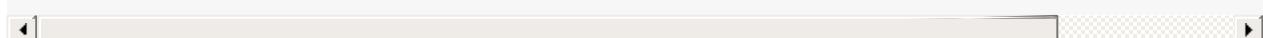
The Java code within your defined ScreenController implementation:

```

private Element popup;
...
public void createMyPopupMenu(){
 popup = nifty.createPopup("niftyPopupMenu");
 Menu myMenu = popup.findNiftyControl("#menu", Menu.class);
 myMenu.setWidth(new SizeValue("100px")); // must be set
 myMenu.addMenuItem("Click me!", "menuItemIcon.png",
 new MenuItem("menuItemid", "blah blah")); // menuItem is a cust
 nifty.subscribe(
 nifty.getCurrentScreen(),
 myMenu.getId(),
 MenuItemActivatedEvent.class,
 new MenuItemActivatedEventSubscriber());
}
public void showMenu() { // the method to trigger the menu
 // If this is a menu that is going to be used many times, then
 // call this in your constructor rather than here
 createMyPopupMenu()
 // call the popup to screen of your choice:
 nifty.showPopup(nifty.getCurrentScreen(), popup.getId(), null);
}

private class menuItem {
 public String id;
 public String name;
 public menuItem(String id, String name){
 this.id= id;
 this.name = name;
 }
}

```



- The `createMyPopupMenu()` method creates the menu with set width so that you can populate it.
- The `showMenu()` method is called by something to trigger the menu (i.e. could be a Key or some other method).
- Note: if you want to be able to access the popup via your id, use `createPopupWithId(id, id)` instead.

To handle menu item events (i.e. calling a method when you click on a menu item), you register (subscribe) a EventTopicSubscriber<MenuItemActivatedEvent> class implementation to a nifty screen and element.

```
private class MenuItemActivatedEventSubscriber
 implements EventTopicSubscriber<MenuItemActivatedEvent> {

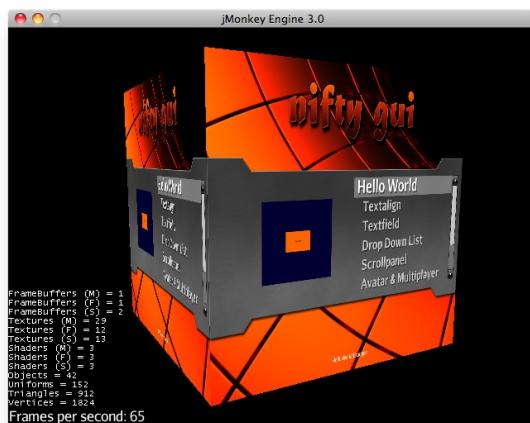
 @Override
 public void onEvent(final String id, final MenuItemActivatedEvent event) {
 menuItem item = (menuItem) event.getItem();
 if ("menuItemid".equals(item.id)) {
 //do something !!!
 }
 }
};
```



## title: Integrating Nifty GUI: Projection

# Integrating Nifty GUI: Projection

1. [Nifty GUI Concepts](#)
2. [Nifty GUI Best Practices](#)
3. [Nifty GUI XML Layout or Nifty GUI Java Layout](#)
4. [Nifty GUI Overlay or Nifty GUI Projection](#)
5. [Interact with the GUI from Java](#)



Typically you define a key (for example escape) to switch the GUI on and off. Then you [overlay](#) the running game with the GUI (you will most likely pause the game then).

Alternatively, you can also project the GUI as a texture onto a mesh textures inside the game. Although this looks cool and “immersive”, this approach is rarely used since it is difficult to record clicks this way. You can only interact with this projected GUI by keyboard, or programmatically. You can select input fields using the arrow keys, and trigger actions using the return key.

This GUI projection variant is less commonly used than the GUI overlay variant. Use cases for GUI projection are, for example, a player avatar using an in-game computer screen.

## Sample Code

- [TestNiftyToMesh.java](#)

# Projecting the User Interface Onto a Texture

You can project the Nifty [GUI](#) onto a texture, load the texture into a material, and assign it to a Geometry (Quads or Boxes are best).

```
/** Create a special viewport for the Nifty GUI */
ViewPort niftyView = renderManager.createPreView("NiftyView", new C
niftyView.setClearEnabled(true);
/** Create a new NiftyJmeDisplay for the integration */
NiftyJmeDisplay niftyDisplay = new NiftyJmeDisplay(
 assetManager, inputManager, audioRenderer, niftyView);
/** Create a new NiftyGUI object */
Nifty nifty = niftyDisplay.getNifty();
/** Read your XML and initialize your custom ScreenController */
nifty.fromXml("Interface/helloworld.xml", "start", new MySettingsSc

/** Prepare a framebuffer for the texture niftytex */
niftyView.addProcessor(niftyDisplay);
FrameBuffer fb = new FrameBuffer(1024, 768, 0);
fb.setDepthBuffer(Format.Depth);
Texture2D niftytex = new Texture2D(1024, 768, Format.RGB8);
fb.setColorTexture(niftytex);
niftyView.setClearEnabled(true);
niftyView.setOutputFrameBuffer(fb);

/** This is the 3D cube we project the GUI on */
Box b = new Box(Vector3f.ZERO, 1, 1, 1);
Geometry geom = new Geometry("Box", b);
Material mat = new Material(assetManager, "Common/MatDefs/Misc/Unsh
mat.setTexture("m_ColorMap", niftytex); /** Here comes the texture!
geom.setMaterial(mat);
rootNode.attachChild(geom);
```

The `MySettingsScreen` class is a custom `de.lessvoid.nifty.screen.ScreenController` in which you implement your [GUI](#) behaviour. The variable `data` contains an object that you use to exchange state info with the game. See [Nifty GUI Java Interaction](#) for details on how to create this class.

Run the code sample. You select buttons on this GUI with the arrow keys and then press return. Note that clicking on the texture will not work!

## Next Steps

Now that you have layed out and integrated the GUI in your app, you want to respond to user input and display the current game.

- [Interact with the GUI from Java](#)

[gui](#), [documentation](#), [nifty](#), [hud](#), [texture](#)

</div>

## **title: Nifty GUI 1.3 - Usecase Scenarios**

# **Nifty GUI 1.3 - Usecase Scenarios**

This document contains typical NiftyGUI usecase scenarios, such as adding effects, game states, and creating typical game screens.

Requirements: These tips assume that you have read and understood the [Creating JME3 User Interfaces with Nifty GUI](#) tutorial, and have already laid out a basic GUI that interacts with your JME3 application. Here you learn how you integrate the GUI better, and add effects and advanced controls.

## **Switch Game States**

In a JME game, you typically have three game states:

1. Stopped: The game is stopped, a StartScreen is displayed.
2. Running: The game is running, the in-game HudScreen is displayed.
3. Paused: The game is paused, a PausedScreen is displayed.

(Aside: Additionally, the Stopped state often contains a LoadScreen, LogonScreen, OptionsScreen, CharacterCreationScreen, HighScoreScreen, CreditsScreen, etc. Some games let you access the OptionsScreen in the Paused state as well. The Running state can also contain an InventoryScreen, ItemShopScreen, StatsScreen, SkillScreen, etc.)

In JME, game states are implemented as custom [AppState](#) objects. Write each AppState so it brings its own input mappings, rootNode content, update loop behaviour, etc with it.

1. Stopped: StartScreen AppState + GuiInputs AppState
2. Paused: PausedScreen AppState + GuiInputs AppState
3. Running: HudScreen AppState + InGameInputs AppState + BulletAppState (jme physics), ...

When the player switches between game states, you detach one set of AppStates, and attach another. For example, when the player pauses the running game, you use a boolean switch to pause the game loop and deactivate the game inputs (shooting, navigation). The

screen is overlayed with a PausedScreen, which contains a visible mouse pointer and a Continue button. When the player clicks Continue, the mouse pointer is deactivated, the in-game input and navigational mappings are activated, and the game loop continues.

## Get Access to Application and Update Loop

Since you are writing a jME3 application, you can additionally make any ScreenController class extend the [AbstractAppState](#) class. This gives the ScreenController access to the application object and to the update loop!

```
public class StartScreenState extends AbstractAppState {

 private ViewPort viewPort;
 private Node rootNode;
 private Node guiNode;
 private AssetManager assetManager;
 private Node localRootNode = new Node("Start Screen RootNode");
 private Node localGuiNode = new Node("Start Screen GuiNode");
 private final ColorRGBA backgroundColor = ColorRGBA.Gray;

 public StartScreenState(SimpleApplication app){
 this.rootNode = app.getRootNode();
 this.viewPort = app.getViewPort();
 this.guiNode = app.getGuiNode();
 this.assetManager = app.getAssetManager();
 }

 @Override
 public void initialize(AppStateManager stateManager, Application app)
 {
 super.initialize(stateManager, app);

 rootNode.attachChild(localRootNode);
 guiNode.attachChild(localGuiNode);
 viewPort.setBackgroundColor(backgroundColor);

 /** init the screen */
 }

 @Override
```

```
public void update(float tpf) {
 /** any main loop action happens here */
}

@Override
public void cleanup() {
 rootNode.detachChild(localRootNode);
 guiNode.detachChild(localGuiNode);

 super.cleanup();
}

}
```

It is not sufficient to just inherit from AbstractAppState. You need to instantiate your controller class, register it with app's stateManager and then pass it to nifty. See code sample below.

```
public class TestNiftyGui extends SimpleApplication {
 public void simpleInitApp() {
 StartScreenState startScreenState = new StartScreenState(this)
 stateManager.attach(startScreenState);
 // [...] boilerplate init nifty omitted
 nifty.fromXml("Interface/myGui.xml", "start", startScreenState)
 }
}
```

</div>

## Know Your Variables

Variable	Description
<code> \${CALL.myMethod()}</code>	Calls a method in the current ScreenController and gets the method's return String. The method can also be void and have a side effect, e.g. play a sound etc.
<code> \${ENV.HOME}</code>	Returns the path to user's home directory.
<code> \${ENV.key}</code>	Looks up <code>key</code> in the environment variables. Use it like Java's <code>System.getenv("key")</code> .
<code> \${PROP.key}</code>	looks up <code>key</code> in the Nifty properties. Use <code>Nifty.setGlobalProperties(properties)</code> and <code>Nifty.getGlobalProperties("key")</code> . Or <code>SystemGetProperties(key)</code> ;

See also: <http://sourceforge.net/apps/mediawiki/nifty-gui/index.php?title=MarkUp>

## Use ScreenControllers for Mutually Exclusive Functionality

Technically you are free to create one ScreenController class for each screen, or reuse the same ScreenController for all or some of them. In the end it may be best to create individual ScreenControllers for functionality that is mutually exclusive.

For example, create a `MyHudScreen.java` for the `hud` screen, and a `MyStartScreen.java` for the `start` screen.

- Include all user interface methods that are needed during the game (while the HUD is up) in `MyHudScreen.java`. Then make this class control all screens that can be up during the game (the HUD screen, a MiniMap screen, an Inventory screen, an Abilities or Skills screen, etc). All these screens possibly share data (game data, player data), so it makes sense to control them all with methods of the same `MyHudScreen.java` class.
- The start screen, however, is mostly independent of the running game. Include all user interface methods that are needed outside the game (while you are on the start screen) in `MyStartScreen.java`. Then make this class control all screens that can be up outside the game (the Start screen, a Settings/Options screen, a HighScore screen, etc). All these classes need to read and write saved game data, so it makes sense to control them all with methods of the same `MyStartScreen.java` class.

## Create a "Loading..." Screen

Get the full [Loading Screen](#) tutorial here.

## Create a Popup Menu

Get the full [Nifty GUI PopUp Menu](#) tutorial here.

## Add Visual Effects

You can register effects to screen elements.

- Respond to element events such as onStartScreen, onEndScreen, onHover, onFocus, onActive,
- Trigger effects that change movement, blending, size, color, fading, and much more.

Here is an example that moves a panel when the startScreen opens. You place an `<effect>` tag inside the element that you want to be affected.

```
<panel height="25%" width="35%" ...>
 <effect>
 <onStartScreen name="move" mode="in" direction="top" length="3000" />
 </effect>
</panel>
```

Learn more from the NiftyGUI page:

- <http://sourceforge.net/apps/mediawiki/nifty-gui/index.php?title=Effects>

## Add Sound Effects

Playing sounds using Nifty is also possible with a `playSound` effect as trigger. Remember to first register the sound that you want to play:

```
<registerSound id="myclick" filename="Interface/sounds/ButtonClick.mp3" />
...
<label>
 <effect>
 <onClick name="playSound" sound="myclick"/>
 </effect>
</label>
```

## Pass ClickLoc From Nifty to Java

After a mouse click, you may want to record the 2D clickLoc and send this info to your Java application. Typical ScreenController methods however only have a String argument. You'd have to convert the String to ints.

To pass the clickLoc as two ints, you can use the special `(int x, int y)` syntax in the ScreenController:

```
public void clicked(int x, int y) {
 // here you can use the x and y of the clickLoc
}
```

In the Nifty GUI screen code (e.g. XML file) you must call the `(int x, int y)` method *without* any parameters!

```
<interact onClick="clicked()"/>
```

You can name the method (here `clicked`) whatever you like, as long as you keep the argument syntax.

## Load Several XML Files

The basic Nifty GUI example showed how to use the `nifty.fromXML()` method to load one XML file containing all Nifty GUI screens. The following code sample shows how you can load several XML files into one nifty object. Loading several files with `nifty.addXml()` allows you to split up each screen into one XML file, instead of all into one hard-to-read XML file.

```
NiftyJmeDisplay niftyDisplay = new NiftyJmeDisplay(assetManager, ir
Nifty nifty = niftyDisplay.getNifty();
nifty.addXml("Interface/Screens/OptionsScreen.xml");
nifty.addXml("Interface/Screens/StartScreen.xml");
nifty.gotoScreen("startScreen");
StartScreenControl screenControl = (StartScreenControl) nifty.getSc
OptionsScreenControl optionsControl = (OptionsScreenControl) nifty.
stateManager.attach(screenControl);
stateManager.attach(optionsControl);
guiViewPort.addProcessor(niftyDisplay);
```

# Register additional explicit screen controllers

In addition to the `nifty.addXml()` methods to attach many nifty XML files, there exists a `nifty.registerScreenController()` method to explicitly attach more screen controllers.

The following code sample shows how you can explicitly attach several screen controllers before adding the XML file to nifty, which would otherwise cause nifty to implicitly instantiate the screen controller class.

```
NiftyJmeDisplay niftyDisplay = new NiftyJmeDisplay(assetManager, ir
Nifty nifty = niftyDisplay.getNifty();

nifty.registerScreenController(new OptionsScreenController(randomCo
nifty.addXml("Interface/Screens/OptionsScreen.xml"));
```

# Design Your Own Styles

By default, your Nifty XML screens use the built-in styles:

```
<useStyles filename="nifty-default-styles.xml" />
```

But you can switch to a set of custom styles in your game project's asset directory like this:

```
<useStyles filename="Interface/Styles/myCustomStyles.xml" />
```

Inside `myCustomStyles.xml` you define styles like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<nifty-styles>
 <useStyles filename="Interface/Styles/Font/myCustomFontStyle.xml" />
 <useStyles filename="Interface/Styles/Button/myCustomButtonStyle.xml" />
 <useStyles filename="Interface/Styles/Label/myCustomLabelStyle.xml" />
 ...
</nifty-styles>
```

Learn more about how to create styles by looking at the [Nifty GUI source code](#) for “nifty-style-black”. Copy it as a template and change it to create your own style.

Learn more from the NiftyGUI page:

- <http://sourceforge.net/apps/mediawiki/nifty-gui/index.php?title=Effects>

[gui](#), [documentation](#), [nifty](#), [hud](#), [click](#), [state](#), [states](#), [sound](#), [effect](#)

</div>

## **title: Laying out the GUI in XML**

# Laying out the GUI in XML

1. [Nifty GUI Concepts](#)
2. [Nifty GUI Best Practices](#)
3. [Nifty GUI XML Layout or Nifty GUI Java Layout](#)
4. [Nifty GUI Overlay or Nifty GUI Projection](#)
5. [Interact with the GUI from Java](#)

You can “draw” the GUI to the screen by writing XML code (alternatively you can also use Java).

## Plan Your GUI Layout



In this tutorial, you want to create two game screens: An out-of-game StartScreen that the players see before the game starts; and an in-game [HUD](#) that displays info during the game. Before writing code, you plan the GUI layout, either on paper or in a graphic application.

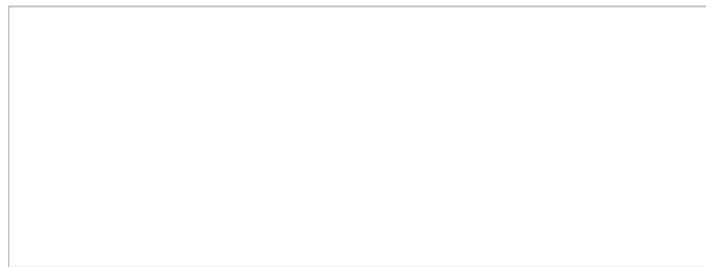
The StartScreen contains:

- The background layer has a centered layout and contains an image.
- The top layer has a vertical layout, containing 3 panels:
  - The top panel contains a label with the game title,
  - The middle panel contains a text field with the game description.
  - The bottom panel has a horizontal layout and contains two more panels:
    - The left panel contains a Start button.
    - The right panel contains a Quit button.

The HUD contains:

- The background layer has a centered layout, and contains the partially transparent HUD image.
- The top layer has a horizontal layout, containing 2 panels:
  - The left panel as transparent spacer.
  - The right panel has a vertical layout containing 2 panels, a label and an image.

# Implement Your GUI Layout



Create an empty `screen.xml` file in the `assets/Interface/` directory of your project. (Rightclick on Interface → New → Other... → GUI → Empty NiftyGui file) Afterwards create the directory `assets/Interface/Fonts` and add a new font e.g. Arial. (**Rightclick on Interface → New → Other... → Other → Folder and GUI → Font**)

One XML file can contain several, or even all screens. As a reminder: Nifty displays one screen at a time; a screen contains several layers on top of one another; each layer contains panels that are embedded into another; the panels contain the actual content (text, images, or controls).

## Make Screens

The following minimal XML file contains a start screen and a HUD screen. (Neither has been defined yet.)

```
<?xml version="1.0" encoding="UTF-8"?>
<nifty xmlns="http://nifty-gui.sourceforge.net/nifty-1.3.xsd" xmlns:gui="http://nifty-gui.sourceforge.net/nifty-1.3.xsd">
 <screen id="start">
 <!-- ... -->
 </screen>
 <screen id="hud">
 <!-- ... -->
 </screen>
</nifty>
```



Every Nifty GUI must have a start screen. The others (in this example, the HUD screen) are optional.

**Note:** In the following examples, the XML schema header is abbreviated to just `<nifty>`.

## Make Layers

The following minimal XML file shows how we added layers to the start screen and HUD screen. Delete all from the file and add following code:

```
<nifty>
 <screen id="start">
 <layer id="background" backgroundColor="#000f">
 <!-- ... -->
 </layer>
 <layer id="foreground" backgroundColor="#0000" childLayout="ver
 <!-- ... -->
 </layer>
 </screen>
 <screen id="hud">
 <layer id="background" backgroundColor="#000f">
 <!-- ... -->
 </layer>
 <layer id="foreground" backgroundColor="#0000" childLayout="hor
 <!-- ... -->
 </layer>
 </screen>
</nifty>
```

In a layer, you can now add panels and arrange them. Panels are containers that mark the areas where you want to display text, images, or controls (buttons etc) later.

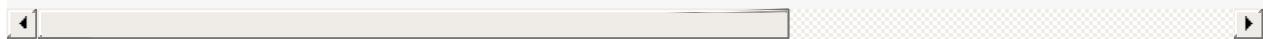
## Make Panels

A panel is the inner-most container (that will contain the actual content: text, images, or controls). You place panels inside layers. The following panels go into in the `start` screen's `foreground` layer:

```

<panel id="panel_top" height="25%" width="75%" align="center"
 backgroundColor="#f008">
</panel>
<panel id="panel_mid" height="50%" width="75%" align="center"
 backgroundColor="#0f08">
</panel>
<panel id="panel_bottom" height="25%" width="75%" align="center"
 backgroundColor="#00f8">
<panel id="panel_bottom_left" height="50%" width="50%" vali
 backgroundColor="#44f8">
</panel>
<panel id="panel_bottom_right" height="50%" width="50%" val
 backgroundColor="#88f8">
</panel>
</panel>

```

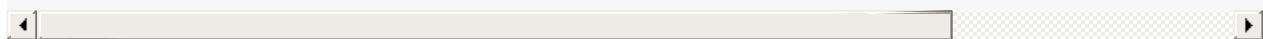


The following panels go into in the `hud` screen's `foreground` layer:

```

<panel id="panel_left" width="80%" height="100%" childLayout=
backgroundColor="#0f08">
 <!-- spacer -->
</panel>
<panel id="panel_right" width="20%" height="100%" childLayout
backgroundColor="#00f8" >
 <panel id="panel_top_right1" width="100%" height="15%" chil
 backgroundColor="#00f8">
 </panel>
 <panel id="panel_top_right2" width="100%" height="15%" chil
 backgroundColor="#44f8">
 </panel>
 <panel id="panel_bot_right" width="100%" height="70%" vali
 backgroundColor="#88f8">
 </panel>
</panel>

```



The result should look as follows:



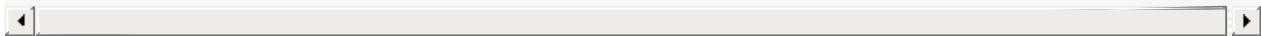
# Adding Content to Panels

See also [Layout Introduction](#) on the Nifty GUI site.

## Add Images

The [start-background.png](#) image is a fullscreen background picture. Add it to `Interface`. In the `start` screen, add the following image element:

```
<layer id="background" childLayout="center">
 <image filename="Interface/start-background.png"></image>
</layer>
```



The [hud-frame.png](#) image is a transparent frame that we use as HUD decoration. Add it to `Interface`. In the `hud` screen, add the following image element:

```
<layer id="background" childLayout="center">
 <image filename="Interface/hud-frame.png"></image>
</layer>
```



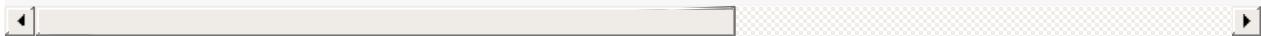
In order to make the `hud-frame.png` independent of the screen resolution you are using, you could use the `imageMode` attribute on the image element [Resizable Images \(ImageMode=resize\) explained](#)

```
<layer id="background" childLayout="center">
 <image filename="Interface/hud-frame.png" imageMode="resize">
</layer>
```



The [face1.png](#) image is an image that you want to use as a status icon. Add it to `Interface`. In the `hud` screen's `foreground` layer, add the following image element:

```
<panel id="panel_top_right2" width="100%" height="15%" childLayout="center">
 <image filename="Interface/face1.png" valign="center" alt="Status icon">
 </image>
</panel>
```

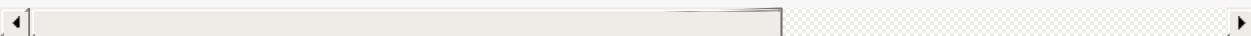


This image is scaled to use 50% of the height and 30% of the width of its container.

## Add Static Text

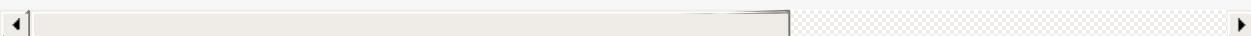
The game title is a typical example of static text. In the `start` screen, add the following text element:

```
<panel id="panel_top" height="25%" width="75%" align="center">
 <text text="My Cool Game" font="Interface/Fonts/Default.fnt" />
</panel>
```



For longer pieces of static text, such as an introduction, you can use `wrap="true"`. Add the following text element to the `start` screen :

```
<panel id="panel_mid" height="50%" width="75%" align="center">
 <text text="Here goes some text describing the game and the
 the text is quite long and needs to wrap at the end of line"
 font="Interface/Fonts/Default.fnt" width="100%" height="100%" wrap="true" />
</panel>
```



The font used is jME3's default font "Interface/Fonts/Default.fnt" which is included in the jMonkeyEngine.JAR. You can add your own fonts to your own `assets/Interface/Fonts` directory. Adjust the path to your font-name.

## Add Controls

Before you can use any control, you must load a Control Definition first. Add the following two lines *before* your screen definitions:

```
<useStyles filename="nifty-default-styles.xml" />
<useControls filename="nifty-default-controls.xml" />
```

Note that the `useStyles` tag must be the first child of the `nifty` tag, otherwise you will see an error in design view.

## Label Control

Use label controls for text that you want to edit dynamically from Java. One example for this is the score display. In the `hud` screen's `foreground` layer, add the following text element:

```
<panel id="panel_top_right" height="100%" width="15%" childLayout="center">
 <control name="label" color="#000" text="123" width="100%" height="100%"/>
</panel>
```

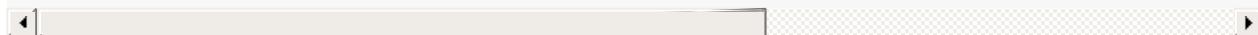


Note that the width and height do not scale the bitmap font, but indirectly make certain it is centered. If you want a different size for the font, you need to provide an extra bitmap font (they come with fixed sizes and don't scale well).

## Button Control

Our GUI plan asks for two buttons on the start screen. You add the Start and Quit buttons to the bottom panel of the `start` screen using the `<control>` element:

```
<panel id="panel_bottom_left" height="50%" width="50%" valign="bottom">
 <control name="button" label="Start" id="StartButton" align="center" style="background-color: #000; color: #fff; border: none; font-size: 16px; padding: 5px; margin-bottom: 10px; border-radius: 5px; font-family: sans-serif; font-weight: bold; text-decoration: none; transition: background-color 0.3s ease; text-align: center; white-space: nowrap; width: 100%; height: 100%"/>
</panel>
<panel id="panel_bottom_right" height="50%" width="50%" valign="bottom">
 <control name="button" label="Quit" id="QuitButton" align="center" style="background-color: #000; color: #fff; border: none; font-size: 16px; padding: 5px; border-radius: 5px; font-family: sans-serif; font-weight: bold; text-decoration: none; transition: background-color 0.3s ease; text-align: center; white-space: nowrap; width: 100%; height: 100%"/>
</panel>
```



Note that these controls don't do anything yet – we'll get to that soon.

Now remove all **backgroundColor="““** tags from your code. They were only needed to show the layout.

Your screen.xml should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<nifty xmlns="http://nifty-gui.sourceforge.net/nifty-1.3.xsd" xmlns:java="http://java.sun.com/xml/ns/javaee">
 <useStyles filename="nifty-default-styles.xml" />
 <useControls filename="nifty-default-controls.xml" />
 <screen id="start">
 <layer id="background" childLayout="center">
 <image filename="Interface/start-background.png"></image>
 </layer>
 <layer id="foreground" childLayout="vertical">
 <panel id="panel_top" height="25%" width="75%" align="center">
 <control name="button" label="Start" id="StartButton" style="background-color: #000; color: #fff; border: none; font-size: 16px; padding: 5px; margin-bottom: 10px; border-radius: 5px; font-family: sans-serif; font-weight: bold; text-decoration: none; transition: background-color 0.3s ease; text-align: center; white-space: nowrap; width: 100%; height: 100%"/>
 </panel>
 <panel id="panel_bottom" height="75%" width="75%" align="center">
 <control name="button" label="Quit" id="QuitButton" style="background-color: #000; color: #fff; border: none; font-size: 16px; padding: 5px; border-radius: 5px; font-family: sans-serif; font-weight: bold; text-decoration: none; transition: background-color 0.3s ease; text-align: center; white-space: nowrap; width: 100%; height: 100%"/>
 </panel>
 </layer>
 </screen>
</nifty>
```

```
<text text="My Cool Game" font="Interface/Fonts/Default.fnt"
</panel>
<panel id="panel_mid" height="50%" width="75%" align="center"
 <text text="Here goes some text describing the game and the
</panel>
<panel id="panel_bottom" height="25%" width="75%" align="center"
 <panel id="panel_bottom_left" height="50%" width="50%" valign="top"
 <control name="button" label="Start" id="StartButton" align="center"
 </panel>
 <panel id="panel_bottom_right" height="50%" width="50%" valign="top"
 <control name="button" label="Quit" id="QuitButton" align="center"
 </panel>
</panel>
</layer>
</screen>
<screen id="hud">
 <layer id="background" childLayout="center">
 <image filename="Interface/hud-frame.png"></image>
 </layer>
 <layer id="foreground" childLayout="horizontal">
 <panel id="panel_left" width="80%" height="100%" childLayout="vertical">
 <panel id="panel_right" width="20%" height="100%" childLayout="vertical">
 <panel id="panel_top_right1" width="100%" height="15%" childLayout="center">
 <control name="label" color="#000" text="123" width="100%" height="100%"/>
 </panel>
 <panel id="panel_top_right2" width="100%" height="15%" childLayout="center">
 <image filename="Interface/face1.png" valign="center" alt="A small face icon."/>
 </panel>
 <panel id="panel_bot_right" width="100%" height="70%" valign="bottom">
 </panel>
 </panel>
 </layer>
 </screen>
</nifty>
```

## Other Controls

Nifty additionally offers many customizable controls such as check boxes, text fields, menus, chats, tabs, ... See also [Elements](#) on the Nifty [GUI](#) site.

## Intermediate Result

When you preview this code in the jMonkeyEngine SDK, our tutorial demo should looks as follows: A start screen with two buttons, and a game screen with a simple HUD frame and a blue cube (which stands for any jME3 game content).



Compare this result with the layout draft above.

## Next Steps

Integrate the [GUI](#) into the game. Typically, you will overlay the [GUI](#).

- [Nifty GUI Overlay](#) (recommended)
- [Nifty GUI Projection](#) (optional)

[gui](#), [documentation](#), [nifty](#), [hud](#)

</div>

## title: Working Blender and OgreXML Versions

# Working Blender and OgreXML Versions

Here you can find working combinations of Blender and the OgreXML exporter, with any tips or bugs associated with each.

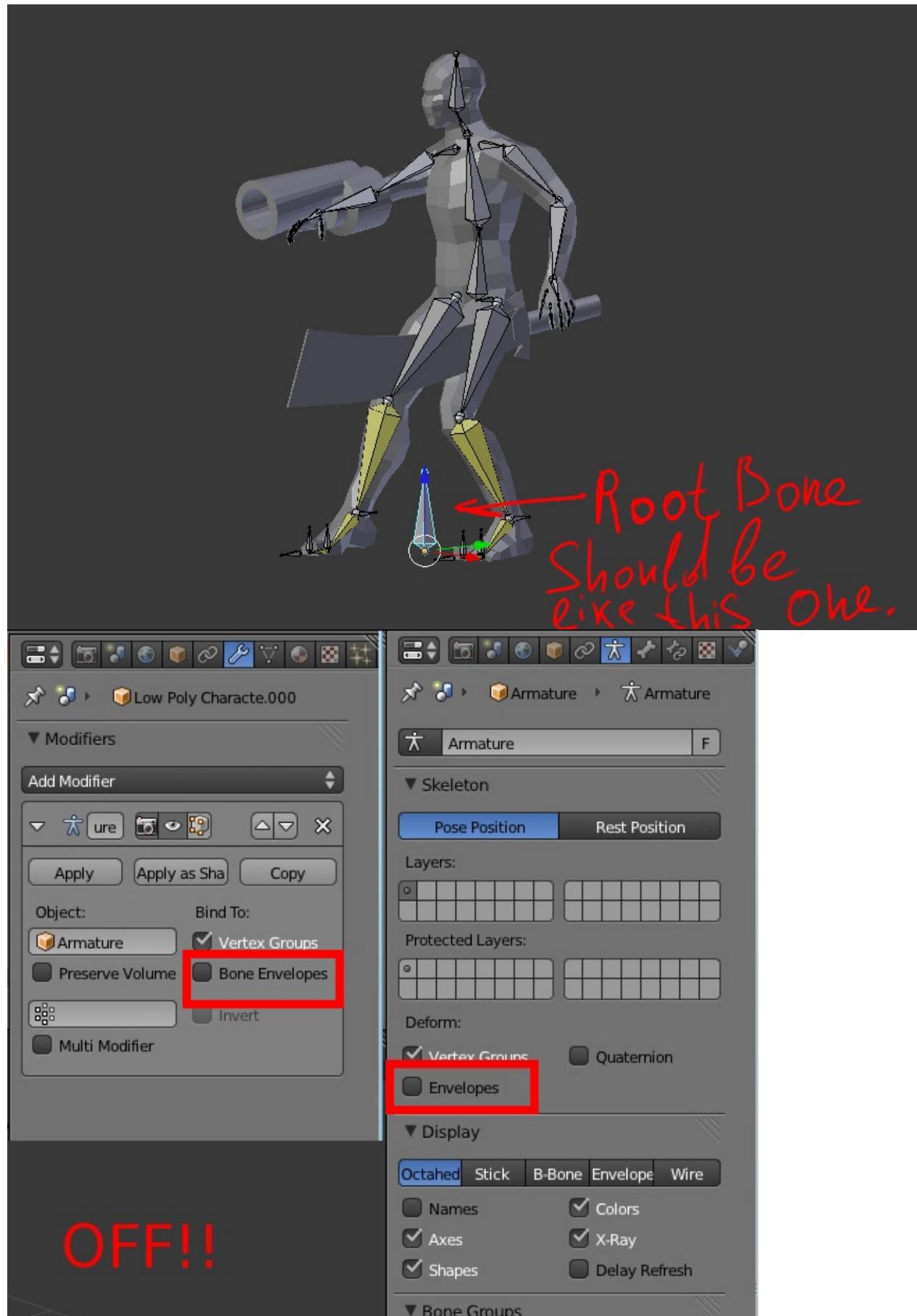
Blender Version	OgreXML Exporter Version	Notes
2.6.3	0.5.8	Root bone, no transforms on object, no envelopes
2.6.2	0.5.5	Root bone, no transforms on object, no envelopes
2.6.1	?	
2.6.0	?	

## Tips

Tips for exporting animations through OgreXML correctly:

- apply all transformations
- armature should have 0,0,0 transformation (loc,rot,scale)
- model object should have 0,0,0 transformation (loc,rot,scale)
- root bone should have 0,0,0 transformation (loc,rot,scale)
- no envelopes

Test Character - [http://dl.dropbox.com/u/26887202/123/jme\\_blender/characterOgre26.zip](http://dl.dropbox.com/u/26887202/123/jme_blender/characterOgre26.zip)



&lt;/div&gt;

# Troubleshooting

**Q:** *My animation is stretched.*

**A:** Use the exporting tips provided above

</div>

## **title: Open Game Finder**

# **Open Game Finder**

The Open Game Finder (OGF) by Mark Schrijver can be plugged into any Java game. OGF enables you to find other people playing the same multiplayer game, and join them.

- Homepage: <http://code.google.com/p/open-game-finder/>
- Documentation: <http://code.google.com/p/open-game-finder/w/list>

Both on the client and the server side of OGF is written purely in Java. OGF has a pluggable architecture and comes with a full set of plugins to get the job done. You can add your own plugins, or replace existing plugins to make them more in line with your game. OGF uses NiftyGUI as the main GUI plugin.

## **Installation**

1. Go to <http://code.google.com/p/open-game-finder/downloads/list>
2. Download Client-1.0-bin.zip and Server-1.0-bin.zip
3. Unzip the two files to, for example, your jMonkeyProjects directory.

## **Setting up the Database**

The OGF server uses an embedded Apache Derby database. You have to install the database, this means creating the data files and adding the tables. You can do this straight from the command line by running a script file.

- On Windows, use installServer.bat to create a new database from scratch. On Mac OS or Linux, run `java -jar lib/Server-0.1.jar install` in the Terminal.
- On Windows, use updateServer.bat to update the difference between the current state of the database and the way it should be. On Mac OS and Linux, run `java -jar lib/Server-0.1.jar update` in the Terminal.

**This new feature is currently untested.**

## **Running the server**

Change into the OGF-Server directory and run the server:

- On Windows: Run startServer.bat
- On Linux and MacOS X: Run `java -jar lib/Server-1.0.jar` in the Terminal.

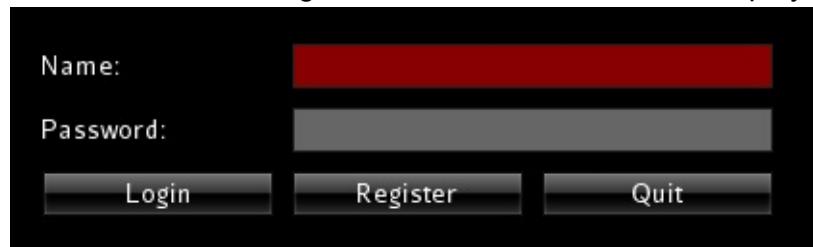
The server is now running and ready to accept connections.

**Note:** In the alpha release, the server runs on localhost. In the final release, you will be able to configure the host!

## Running the client

1. Change into the OGF-Client directory and run the client:
  - On Windows: Run startClient.bat
  - On Linux and MacOS X: Run `java -jar lib/Client-1.0.jar` in the Terminal.
2. If a Display Settings window appears, you can keep the defaults and click OK.

A client is now running, connects to the server, and displays a registration/login window.



**Note:** You can run several

clients on localhost for testing.

## Client: 1. Registration

If clients use OGF for the first time, they need to register. On the main screen of the client:

1. Click Register
2. Choose a user name and password (repeat the password).
3. Select an Avatar image.
4. Click register to complete the registration.

The client registers the account and opens the chat window directly.

## Client: 2. Login

If returning clients are already registered to an OGF server, they can log in. On the main screen of the client:

1. Enter a user name and password that you previously registered.
2. Click Login

The client logs you in and opens the chat window.

## Client: 3. Chat

The chat window shows a list of all users logged in to the server. Logged-in users can send public messages, and can receive public messages from others.

# Connecting to a Game

Q: I want to gather players using the OGF client to connect to the game server. How do I start my multiplayer game?

A: The following sample code demos the typical use case: [OGFClientStartup.java](#)  
In a JME3 Application's init method:

1. Create a com.ractoc.opengamefinder.client.GUIContainer object.
2. Create a game instance using the GUIContainer (via a ClientFactory).
3. Check the com.ractoc.pffj.api.BasePluginMessageResult for success or failure.

After this, continue writing your JME3 init method.

# Configuration

- Q: How can I offer more avatars to choose from?

A: Save the image files in the path `jMonkeyProjects/OGF-Client-1.0-bin/OGF/resources/avatars/`

- Q: How do I configure servers addresses?

A: TBD

[network](#)

`</div>`

## title: Particle Emitter Settings

# Particle Emitter Settings

You cannot create a 3D model for delicate things like fire, smoke, or explosions. Particle Emitters are quite an efficient solution to create these kinds of effects: The emitter renders a series of flat orthogonal images and manipulates them in a way that creates the illusion of anything from a delicate smoke cloud to individual flames, etc. Creating an effect involves some trial and error to get the settings *just right*, and it's worth exploring the expressiveness of the options described below.

Use the [Scene Explorer](#) in the [SDK](#) to design and preview effects.



</div>

## Create an Emitter

1. Create one emitter for each effect:

```
ParticleEmitter explosion = new ParticleEmitter(
 "My explosion effect", Type.Triangle, 30);
```

2. Attach the emitter to the rootNode and position it in the scene:

```
rootNode.attachChild(explosion);
explosion.setLocalTranslation(bomb.getLocalTranslation());
```

3. Trigger the effect by calling

```
explosion.emitAllParticles()
```

4. End the effect by calling

```
explosion.killAllParticles()
```

Choose one of the following mesh shapes

- Type.Triangle
- Type.Point

## Configure Parameters

Not all of these parameters are required for all kinds of effects. If you don't specify one of them, a default value will be used.

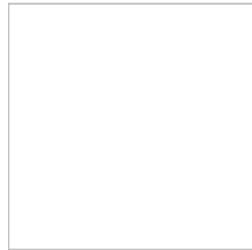
Parameter	Method	Default	
number	setNumParticles()		The of p the Spe con
emission rate	setParticlesPerSec()	20	De hov par per Set the effe Set con effe
size	setStartSize() , setEndSize()	0.2f, 2f	The sca Set for Set for
color	setStartColor() , setEndColor()	gray	Col opa par col Set col effe det Set col effe

direction/velocity	<pre>getParticleInfluencer(). setInitialVelocity(initialVelocity)</pre>	Vector3f(0,0,0)	A vector representing the initial velocity of the particle.
fanning out	<pre>getParticleInfluencer(). setVelocityVariation(variation)</pre>	0.2f	How much the velocity varies from the initial velocity.
direction (pick one)	<pre>setFacingVelocity()</pre>	false	true if the particle should face the direction it was born in.
direction (pick one)	<pre>setFaceNormal()</pre>	Vector3f.NAN	Vector representing the normal of the surface the particle was born on.
			The angle between the particle's initial velocity and the surface normal.

			< 1 mo min ste ma bur ma swa ma min valu eve (e.g. to v to c effe indi
lifetime	<code>setLowLife()</code> , <code>setHighLife()</code>	3f, 7f	
spinning	<code>setRotateSpeed()</code>	0f	0 = dor (e.g. con > 0 spin det mis
rotation	<code>setRandomAngle()</code>	false	true spr ran em det fals stra the tex
gravity	<code>setGravity()</code>	Vector3f(0.0f,0.1f,0.0f)	Par dire (e.g. (0,0 flyin (e.g. gra
start area	<code>setShape(new EmitterSphereShape( Vector3f.ZERO, 2f));</code>	EmitterPointShape()	By em em poi inci sha sph poi can

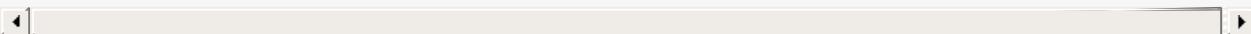
Build up your effect by specifying one parameter after the other. If you change several parameters at the same time, it's difficult to tell which of the values caused which outcome.

## Create an Effect Material



Use the common Particle.j3md Material Definition and a texture to specify the shape of the particles. The shape is defined by the texture you provide and can be anything – debris, flames, smoke, mosquitoes, leaves, butterflies... be creative.

```
Material flash_mat = new Material(
 assetManager, "Common/MatDefs/Misc/Particle.j3md");
flash_mat.setTexture("Texture",
 assetManager.loadTexture("Effects/Explosion/flash.png"));
flash.setMaterial(flash_mat);
flash.setImagesX(2); // columns
flash.setImagesY(2); // rows
flash.setSelectRandomImage(true);
```



The effect texture can be one image, or contain a sprite animation – a series of slightly different pictures in equally spaced rows and columns. If you choose the sprite animation:

- Specify the number of rows and columns using setImagesX(2) and setImagesY().
- Specify whether you want to play the sprite series in order (animation), or at random (explosion, flame), by setting setSelectRandomImage() true or false.

**Examples:** Have a look at the following default textures and you will see how you can create your own sprite textures after the same fashion.

## Default Particle Textures

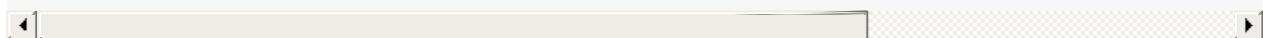
The Material is used together with grayscale texture: The black parts will be transparent and the white parts will be opaque (colored). The following effect textures are available by default from `test-data.jar`. You can also load your own textures from your assets directory.

Texture Path	Dimension	Preview
Effects/Explosion/Debris.png	3*3	
Effects/Explosion/flame.png	2*2	
Effects/Explosion/flash.png	2*2	
Effects/Explosion/roundspark.png	1*1	
Effects/Explosion/shockwave.png	1*1	
Effects/Explosion/smoketrail.png	1*3	
Effects/Explosion/spark.png	1*1	
Effects/Smoke/Smoke.png	1*15	

**Tip:** Use the `setStartColor()` / `setEndColor()` settings described above to colorize the white and gray parts of textures.

## Usage Example

```
ParticleEmitter fire = new ParticleEmitter("Emitter", Type.Tria
Material mat_red = new Material(assetManager, "Common/MatDefs/M
mat_red.setTexture("Texture", assetManager.loadTexture("Effects
fire.setMaterial(mat_red);
fire.setImagesX(2); fire.setImagesY(2); // 2x2 texture animatio
fire.setEndColor(new ColorRGBA(1f, 0f, 0f, 1f)); // red
fire.setStartColor(new ColorRGBA(1f, 1f, 0f, 0.5f)); // yellow
 fire.getParticleInfluencer().setInitialVelocity(new Vector3
fire.setStartSize(1.5f);
fire.setEndSize(0.1f);
fire.setGravity(0,0,0);
fire.setLowLife(0.5f);
fire.setHighLife(3f);
fire.getParticleInfluencer().setVelocityVariation(0.3f);
rootNode.attachChild(fire);
```



Browse the full source code of all [effect examples](#) here.

---

See also: [Effects Overview](#)

[documentation](#), [effect](#)

</div>

## **title: Physics Listeners**

# **Physics Listeners**

You can control physical objects (push them around) by applying physical forces to them.

Typically, you also want to respond to the resulting collisions, e.g. by subtracting health points or by playing a sound. To specify how the game responds to such physics events, you use Physics Listeners.

## **Sample Code**

- [TestCollisionListener.java](#)
- [TestAttachGhostObject.java](#)
- [TestGhostObject.java](#)

## **PhysicsGhostObjects**

Attach a `com.jme3.bullet.control.GhostControl` to any `Spatial` to turn it into a `PhysicsGhostObject`. Ghost objects automatically follow their spatial and detect collisions. The attached ghost itself is invisible and non-solid (!) and doesn't interfere with your game otherwise, it only passively reports collisions.

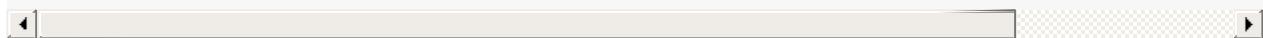
You can leave the `GhostControl` non-solid and invisible and attach it to an (invisible) `Node` in the scene to create something like a motion detector. But a `GhostControl` also works fine when added to `spatial`s that are solid (with `RigidBodyControl`) and visible (with `Geometry`). One use case for `GhostControls` is to check for collisions among `CharacterControls` when the characters are walking.

The shape of the ghost depends on the `CollisionShape` that you gave the `GhostControl`. This means that the `GhostControl`'s shape can be different from the `RigidBodyControl`'s shape. For example, the non-solid ghost shape can be bigger than the solid shape of the `Spatial` (so you can "feel" ahead).

```

GhostControl ghost = new GhostControl(
 new BoxCollisionShape(new Vector3f(1,1,1))); // a box-shaped ghost
Node node = new Node("a ghost-controlled thing");
node.addControl(ghost); // the ghost follows the node
// Optional: Add a Geometry, or other controls, to the node if you want
...
// attach everything to activate it
rootNode.attachChild(node);
getPhysicsSpace().add(ghost);

```



Ghost methods	Usage
getOverlappingObjects()	Returns the List of objects that are currently colliding (overlapping) with the ghost.
getOverlappingCount()	Returns the number of currently colliding objects.
getOverlapping(i)	Get PhysicsCollisionObject number i.

## Physics Tick Listener

The jBullet Physics implementation is stepped at a constant 60 physics ticks per second frame rate. Applying forces or checking for overlaps only has an effect right at a physics update cycle, which is not every frame. If you do physics interactions at arbitrary spots in the simpleUpdate() loop, calls will be dropped at irregular intervals, because they happen out of cycle.

### When (Not) to Use Tick Listener?

When you write game mechanics that apply forces, you must implement a tick listener (`com.jme3.bullet.PhysicsTickListener`) for it. The tick listener makes certain the forces are not dropped, but applied in time for the next physics tick.

Also, when you check for overlaps of two physical objects using a `GhostControl`, you cannot just go `ghost.getOverlappingObjects()` somewhere outside the update loop. You have to make certain 1 physics tick has passed before the overlapping objects list is filled with data. Again, the `PhysicsTickListener` does the timing for you.

When your game mechanics however just poll the current state (e.g. `getPhysicsLocation()`) of physical objects, or if you only use the `GhostControl` like a sphere trigger inside an update loop, then you don't need an extra `PhysicsTickListener`.

## How to Listen to Physics Ticks

Here's is the declaration of an exemplary Physics Control that listens to ticks. (The example shows a RigidBodyControl, but it can also be GhostControl.)

```
public class MyCustomControl
 extends RigidBodyControl implements PhysicsTickListener { ... }
```

When you implement the interface, you have to implement `physicsTick()` and `preTick()` methods.

- `prePhysicsTick(PhysicsSpace space, float tpf)` is called before each step, here you apply forces (change the state).
- `physicsTick(PhysicsSpace space, float tpf)` is called after each step, here you poll the results (get the current state).

The tpf value is time per frame in seconds. You can use it as a factor to time actions so they run equally on slow and fast machines.

```
@override
public void prePhysicsTick(PhysicsSpace space, float tpf){
 // apply state changes ...
}

@Override
public void physicsTick(PhysicsSpace space, float tpf){
 // poll game state ...
}
```

## Physics Collision Listener

### When (Not) to Use Collision Listener

If you do not implement the Collision Listener interface (`com.jme3.bullet.collision.PhysicsCollisionListener`), a collisions will just mean that physical forces between solid objects are applied automatically. If you just want “Balls rolling, bricks falling” you do not need a listener.

If however you want to respond to a collision event (com.jme3.bullet.collision.PhysicsCollisionEvent) with a custom action, then you need to implement the PhysicsCollisionListener interface. Typical actions triggered by collisions include:

- Increasing a counter (e.g. score points)
- Decreasing a counter (e.g. health points)
- Triggering an effect (e.g. explosion)
- Playing a sound (e.g. explosion, ouch)
- ... and countless more, depending on your game

## How to Listen to Collisions

You need to add the PhysicsCollisionListener to the physics space before collisions will be listened for. Here's an example of a Physics Control that uses a collision listener. (The example shows a RigidBodyControl, but it can also be GhostControl.)

```
public class MyCustomControl extends RigidBodyControl
 implements PhysicsCollisionListener {
 public MyCustomControl() {
 bulletAppState.getPhysicsSpace().addCollisionListener(this)
 ...
 }
}
```

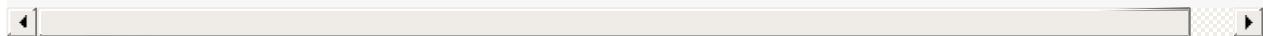
To respond to the PhysicsCollisionEvent you now have to override the `collision()` method in MyCustomControl. This gives you access to the `event` object. Mostly you will be interested in the identity of any two nodes that collided: `event.getNodeA()` and `event.getNodeB()`.

After you identify the colliding nodes, specify the action to trigger when this pair collides. Note that you cannot know which one will be Node A or Node B, you have to deal with either variant.

```

public void collision(PhysicsCollisionEvent event) {
 if (event.getNodeA().getName().equals("player")) {
 final Node node = event.getNodeA();
 /** ... do something with the node ... */
 } else if (event.getNodeB().getName().equals("player")) {
 final Node node = event.getNodeB();
 /** ... do something with the node ... */
 }
}

```



Note that after the `collision()` method ends, the `PhysicsCollisionEvent` is cleared. You must get all objects and values you need within the `collision()` method.

</div>

## Reading Details From a PhysicsCollisionEvent

The `PhysicsCollisionEvent` `event` gives you access to detailed information about the collision. You already know the event objects can identify which nodes collided, but it even knows how hard they collided:

Method	Purpose
<code>getObjectA()</code> <code>getObjectB()</code>	The two participants in the collision. You cannot know in advance whether some node will be recorded as A or B, you always have to consider both cases.
<code>getAppliedImpulse()</code>	A float value representing the collision impulse
<code>getAppliedImpulseLateral1()</code>	A float value representing the lateral collision impulse
<code>getAppliedImpulseLateral2()</code>	A float value representing the lateral collision impulse
<code>getCombinedFriction()</code>	A float value representing the collision friction
<code>getCombinedRestitution()</code>	A float value representing the collision restitution (bounciness)

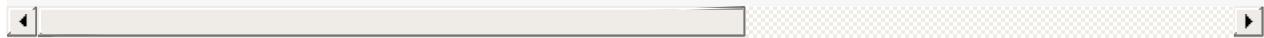
Note that after the collision method has been called the object is not valid anymore so you should copy any data you want to keep into local variables.

## Collision Groups

You can improve performance by restricting the number of tests that collision detection has to perform. If you have a case where you are only interested in collisions between certain objects but not others, you can assign sets of physical objects to different collision groups.

For example, for a click-to-select, you only care if the selection ray collides with a few selectable objects such as dropped weapons or powerups (one group), but not with non-selectables such as floors or walls (different group).

```
myNode.getControl(RigidBodyControl.class).setCollisionGroup(Physics
myNode.getControl(RigidBodyControl.class).setCollideWithGroups(Phys
```



[documentation](#), [physics](#), [collision](#), [forces](#), [interaction](#)

</div>

## **title: Physics: Gravity, Collisions, Forces**

# **Physics: Gravity, Collisions, Forces**

A physics simulation is used in games and applications where objects are exposed to physical forces: Think of games like pool billiard and car racing simulators. Massive objects are pulled by gravity, forces cause objects to gain momentum, friction slows them down, solid objects collide and bounce off one another, etc. Action and Adventure games also make use of physics to implement solid obstacles, falling, and jumping.

The jMonkeyEngine3 has built-in support for [jBullet Physics](#) (based on [Bullet Physics](#)) via the `com.jme3.bullet` package. This article focuses mostly on the RigidBodyControl, but also introduces you to others.

If you are looking for info on how to respond to physics events such as collisions, read about [Physics Listeners](#).

## **Technical Overview**

jME3 has a complete, slightly adapted but fully wrapped Bullet [API](#) that uses normal jME math objects (Vector3f, Quaternion etc) as input/output data. All normal bullet objects like RigidBodies, Constraints (called “Joints” in jME3) and the various collision shapes are available, all mesh formats can be converted from jME to bullet.

The PhysicsSpace object is the central object in bullet and all objects have to be added to it so they are physics-enabled. You can create multiple physics spaces as well to have multiple independent physics simulations or to run simulations in the background that you step at a different pace. You can also create a Bullet PhysicsSpace in jME3 with a `com.jme3.bullet.BulletAppState` which runs a PhysicsSpace along the update loop, which is the easiest way to instantiate a physics space. It can be run in a mode where it runs in parallel to rendering, yet syncs to the update loop so you can apply physics changes safely during the update() calls of Controls and SimpleApplication.

The base bullet objects are also available as simple to use controls that can be attached to spatial to directly control these by physics forces and influences. The RigidBodyControl for example includes a simple constructor that automatically creates a hull collision shape or a

mesh collision shape based on the given input mass and the mesh of the spatial it is attached to. This makes enabling physics on a Geometry as simple as “`spatial.addControl(new RigidBodyControl(1));`”

Due to some differences in how bullet and jME handle the scene and other objects relations there is some things to remember about the controls implementation:

- The collision shape is not automatically updated when the spatial mesh changes
  - You can update it by reattaching the control or by using the CollisionShapeFactory yourself.
- In bullet the scale parameter is on the collision shape (which equals the mesh in jME3) and not on the RigidBody so you cannot scale a collision shape without scaling any other RigidBody with reference of it
  - Note that you should share collision shapes in general and that j3o files loaded from file do that as well when instantiated twice so this is something to consider.
- **Physics objects remain in the physics space when their spatlials are detached from the scene graph!**
  - Use PhysicsSpace.remove(physicsObject) or simply `physicsControl.setEnabled(false);` to remove them from the PhysicsSpace
- If you apply forces to the physics object in an `update()` call they might not get applied because internally bullet still runs at 60fps while your app might run at 120.
  - You can use the PhysicsTickListener interface and register with the physics space and use the `preTick()` method to be sure that you actually apply the force in the right moment.
  - Reading values from the physics objects in the update loop should always yield correct values but they might not change over several frames due to the same reason.
- Reading or writing from the physics objects during the render phase is not recommended as this is when the physics space is stepped and would cause data corruption. This is why the debug display does not work properly in a threaded BulletAppState
- Bullet always uses world coordinates, there is no such concept as nodes so the object will be moved into a world location with no regard to its parent spatial.
  - You can configure this behavior using the `setApplyPhysicsLocal()` method on physics controls but remember the physics space still runs in world coordinates so you can visually detach things that will actually still collide in the physics space.
  - To use the local applying to simulate e.g. the internal physics system of a train passing by, simply create another BulletAppState and add all models with physics controls in local mode to a node. When you move the node the physics will happen all the same but the objects will move along with the node.

When you use this physics simulation, values correspond to the following units:

- 1 length unit (1.0f) equals 1 meter,
- 1 weight unit (1.0f) equals 1 kilogram,
- most torque and rotation values are expressed in radians.

Bullet physics runs internally at 60fps by default. This rate is not dependent on the actual framerate and it does not lock the framerate at 60fps. Instead, when the actual fps is higher than the physics framerate the system will display interpolated positions for the physics objects. When the framerate is lower than the physics framerate, the physics space will be stepped multiple times per frame to make up for the missing calculations.

Internally, the updating and syncing of the actual physics objects in the BulletAppState happens in the following way:

1. collision callbacks (`BulletAppState.update()`)
2. user update (`simpleUpdate` in main loop, `update()` in Controls and AppStates)
3. physics to scenegraph syncing and applying (`updateLogicalState()`)
4. stepping physics (before or in parallel to `Application.render()`)

## Sample Code

Full code samples are here:

- [TestBrickWall.java](#)
- [TestQ3.java](#)
- [TestSimplePhysics.java](#)
- [TestWalkingChar.java](#)

## Physics Application

A short overview of how to write a jME application with Physics capabilities:

Do the following once per application to gain access to the `physicsSpace` object:

1. Make your application extend `com.jme3.app.SimpleApplication`.
2. Create a BulletAppState field:

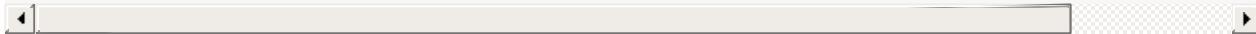
```
private BulletAppState bulletAppState;
```

3. Initialize your bulletAppState and attach it to the state manager:

```
public void simpleInitApp() {
 bulletAppState = new BulletAppState();
 stateManager.attach(bulletAppState);
```

In your application, you can always access the `BulletAppState` via the `ApplicationStateManager`:

```
BulletAppState bas = app.getStateManager().getState(BulletAppState.
```



For each Spatial that you want to be physical:

1. Create a `CollisionShape`.
2. Create the `PhysicsControl` from the `CollisionShape` and a mass value.
3. Add the `PhysicsControl` to its `Spatial`.
4. Add the `PhysicsControl` to the `PhysicsSpace`.
5. Attach the `Spatial` to the `rootNode` (as usual).
6. (Optional) Implement the `PhysicsCollisionListener` interface to respond to `PhysicsCollisionEvent` `s`.

Let's look at the details:

```
</div>
```

## Create a CollisionShape

A `CollisionShape` is a simplified shape for which physics are easier to calculate than for the true shape of the model. This simplification approach speeds up the simulation greatly.

Before you can create a `Physics Control`, you must create a `CollisionShape` from the `com.jme3.bullet.collision.shapes` package. (Read the tip under “`PhysicsControls` Code Samples” how to use default `CollisionShapes` for Boxes and Spheres.)

Non-Mesh CollisionShape	Usage	Examples
BoxCollisionShape()	Box-shaped behaviour, does not roll.	Oblong or cubic objects like bricks, crates, furniture.
SphereCollisionShape()	Spherical behaviour, can roll.	Compact objects like apples, soccer balls, cannon balls, compact spaceships.
CylinderCollisionShape()	Tube-shaped and disc-shaped behaviour, can roll on one side.	Oblong objects like pillars. Disc-shaped objects like wheels, plates.
CompoundCollisionShape()	A CompoundCollisionShape allows custom combinations of shapes. Use the <code>addChildShape()</code> method on the compound object to add other shapes to it and position them relative to one another.	A car with wheels (1 box + 4 cylinders), etc.
CapsuleCollisionShape()	A built-in compound shape of a vertical cylinder with one sphere at the top and one sphere at the bottom. Typically used with <a href="#">CharacterControls</a> : A cylinder-shaped body does not get stuck at corners and vertical obstacles; the rounded top and bottom do not get stuck on stair steps and ground obstacles.	Persons, animals.
SimplexCollisionShape()	A physical point, line, triangle, or rectangle Shape, defined by one to four points.	Guardrails
PlaneCollisionShape()	A 2D plane. Very fast.	Flat solid floor or wall.

All non-mesh CollisionShapes can be used for dynamic, kinematic, as well as static SpatialS.  
(Code samples see below)

Mesh CollisionShapes	Usage	Examples
MeshCollisionShape	A mesh-accurate shape for static or kinematic SpatialS. Can have complex shapes with openings and appendages. <b>Limitations:</b> Collisions between two mesh-accurate shapes cannot be detected, only non-mesh shapes can collide with this shape. This Shape does not work with dynamic SpatialS.	A whole static game level model.
HullCollisionShape	A less accurate shape for dynamic SpatialS that cannot easily be represented by a CompoundShape. <b>Limitations:</b> The shape is convex (behaves as if you gift-wrapped the object), i.e. openings, appendages, etc, are not individually represented.	A dynamic 3D model.
GImpactCollisionShape	A mesh-accurate shape for dynamic SpatialS. It uses <a href="http://gimpact.sourceforge.net/">http://gimpact.sourceforge.net/</a> . <b>Limitations:</b> CPU intensive, use sparingly! We recommend using HullCollisionShape (or CompoundShape) instead to improve performance. Collisions between two mesh-accurate shapes cannot be detected, only non-mesh shapes can collide with this shape.	Complex dynamic objects (like spiders) in Virtual Reality or scientific simulations.
HeightfieldCollisionShape	A mesh-accurate shape optimized for static terrains. This shape is much faster than other mesh-accurate shapes. <b>Limitations:</b> Requires heightmap data. Collisions between two mesh-accurate shapes cannot be detected, only non-mesh shapes can collide with this shape.	Static terrains.

On a CollisionShape, you can apply a few properties

CollisionShape Method	Property	Examples
setScale(new Vector3f(2f,2f,2f))	You can change the scale of collisionshapes (whether it be, Simple or Mesh). You cannot change the scale of a CompoundCollisionShape however. A sphere collision shape, will change its radius based on the X component of the vector passed in. You must scale a collision shape before attaching it to the physicsSpace, or you must readd it to the physicsSpace each time the scale changes.	Scale a player in the Y axis by 2: new Vector3f(1f, 2f, 1f)

The mesh-accurate shapes can use a CollisionShapeFactory as constructor (code samples see below).

Pick the simplest and most applicable shape for the mesh for what you want to do: If you give a box a sphere collision shape, it will roll; if you give a ball a box collision shape, it will sit on a slope. If the shape is too big, the object will seem to float; if the shape is too small it will seem to sink into the ground. During development and debugging, you can make collision shapes visible by adding the following line after the bulletAppState initialization:

```
bulletAppState.getPhysicsSpace().enableDebug(assetManager);
```

</div>

## CollisionShape Code Samples

- One way of using a constructor and a Geometry's mesh for static Spatial:

```
MeshCollisionShape level_shape =
 new MeshCollisionShape(level_geo.getMesh());
```

- One way of using a constructor and a Geometry's mesh for dynamic Spatial:

```
HullCollisionShape shape =
 new HullCollisionShape(katamari_geo.getMesh());
```

- Creating a dynamic compound shape for a whole Node and subnodes:

```
CompoundCollisionShape myComplexShape =
 CollisionShapeFactory.createMeshShape((Node) myComplexGeo-
```

- Creating a dynamic HullCollisionShape shape (or CompoundCollisionShape with HullCollisionShapes as children) for a Geometry:

```
CollisionShape shape =
 CollisionShapeFactory.createDynamicMeshShape(spaceCraft);
```

- An angular, non-mesh-accurate compound shape:

```
CompoundCollisionShape boxShape =
 CollisionShapeFactory.createBoxShape((Node) crate_geo);
```

- A round, non-mesh-accurate compound shape:

```
SphereCollisionShape sphereShape =
 new SphereCollisionShape(1.0f);
```

## Create PhysicsControl

BulletPhysics are available in jME3 through PhysicsControls classes from the com.jme3.bullet.control package. jME3's PhysicsControl classes directly extend BulletPhysics objects and are the recommended way to use physics in a jME3 application. PhysicsControls are flexible and can be added to any Spatial to make it act according to physical properties.

Standard PhysicsControls	Usage	Examples
RigidBodyControl	The most commonly used PhysicsControl. You can use it for dynamic objects (solid objects that freely affected by collisions, forces, or gravity), for static objects (solid but not affected by any forces), or kinematic objects (remote-controlled solid objects).	Impacting projectiles, moving obstacles like crates, rolling and bouncing balls, elevators, flying aircraft or space ships. Solid immobile floors, walls, static obstacles.
GhostControl	Use for collision and intersection detection between physical objects. A GhostControl itself is <i>non-solid</i> and invisible. GhostControl moves with the Spatial it is attached to. Use GhostControls to <a href="#">implement custom game interactions</a> by adding it to a visible Geometry.	A monster's "aggro radius", CharacterControl collisions, motion detectors, photo-electric alarm sensors, poisonous or radioactive perimeters, life-draining ghosts, etc.

Special PhysicsControls	Usage	Examples
VehicleControl PhysicsVehicleWheel	Special Control used for <a href="#">"terrestrial vehicles with suspension and wheels.</a>	Cars, tanks, hover crafts, ships, motorcycles...
CharacterControl	Special Control used for <a href="#">Walking Characters.</a>	Upright walking persons, animals, robots...
RagDollControl	Special Control used for <a href="#">collapsing, flailing, or falling characters</a>	Falling persons, animals, robots, "Rag dolls"

Click the links for details on the special PhysicsControls. This article is about RigidBodyControl.

## Physics Control Code Samples

The most commonly used physics control is RigidBodyControl. The RigidBodyControl constructor takes up to two parameters: a collision shape and a mass (a float in kilograms). The mass parameter also determines whether the object is dynamic (movable) or static (fixed). For a static object such as a floor or wall, specify zero mass.

```
RigidBodyControl myThing_phys =
 new RigidBodyControl(myThing_shape , 123.0f); // dynamic
```

```
RigidBodyControl myDungeon_phys =
 new RigidBodyControl(myDungeon_shape , 0.0f); // static
```

If you give your floor a non-zero mass, it will fall out of the scene!

The following creates a box Geometry with the correct default BoxCollisionShape:

```
Box b = new Box(1,1,1);
Geometry box_geo = new Geometry("Box", b);
box_geo.addControl(new RigidBodyControl(1.0f)); // explicit non-z
```

The following creates a MeshCollisionShape for a whole loaded (static) scene:

```
...
gameLevel.addControl(new RigidBodyControl(0.0f)); // explicit zero
```

Spheres and Boxes automatically fall back on the correct default CollisionShape if you do not specify a CollisionShape in the RigidBodyControl constructor. Complex static objects can fall back on MeshCollisionShapes, unless it is a Node, in which case it will become a CompoundCollisionShape containing a MeshCollisionShape

## Add PhysicsControl to Spatial

For each physical Spatial in the scene:

1. Add a PhysicsControl to a Spatial.

```
myThing_geo.addControl(myThing_phys);
```

2. Remember to also attach the Spatial to the rootNode, as always!

## Add PhysicsControl to PhysicsSpace

The PhysicsSpace is an object in BulletAppState that is like a rootNode for Physics Controls.

- Just like you add the Geometry to the rootNode, you add its PhysicsControl to the PhysicsSpace.

```
bulletAppState.getPhysicsSpace().add(myThing_phys);
rootNode.attachChild(myThing_geo);
```

- When you remove a Geometry from the scene and detach it from the rootNode, also remove the PhysicsControl from the PhysicsSpace:

```
bulletAppState.getPhysicsSpace().remove(myThing_phys);
myThing_geo.removeFromParent();
```

You can either add the *PhysicsControl* to the PhysicsSpace, or add the PhysicsControl to the Geometry and then add the *Geometry* to the PhysicsSpace. jME3 understands both and the outcome is the same.

## Changing the Scale of a PhysicsControl

To change the scale of a PhysicsControl you must change the scale of the collisionshape which belongs to it.

MeshCollisionShapes can have a scale correctly set, but it only works when being constructed on a geometry (not a node). CompoundCollisionShapes cannot be scaled at this time(the type obtained when creating a CollisionShape from a Node i.e using imported models).

When you import a model from blender, it often comes as a Node (even if it only contains 1 mesh), which is by de-facto automatically converted to a CompoundCollisionShape. So when you try to scale this it won't work! Below illustrates an example, of how to scale an imported model:

```
// Doesn't scale
// This modified version contains Node -> Geometry (name = "MonkeyHead")
Spatial model = assetManager.loadModel("Models/MonkeyHead.j3o"); model
// Won't work as this is now a CompoundCollisionShape containing a
model.getControl(RigidBodyControl.class).getCollisionShape().setScaling(1, 1, 1);
bulletAppState.getPhysicsSpace().add(model);

// Works fine
Spatial model = assetManager.loadModel("Models/MonkeyHead.j3o"); //
 // IMPORTANT : You must navigate to the Geometry for this to work
Geometry geom = ((Geometry) ((Node) model).getChild("MonkeyHeadGeometry"));
geom.addControl(new RigidBodyControl(0));
// Works great (scaling of a MeshCollisionShape)
geom.getControl(RigidBodyControl.class).getCollisionShape().setScaling(1, 1, 1);
bulletAppState.getPhysicsSpace().add(geom);
```



With the corresponding output below: [External Link](#) [External Link](#)

</div>

## PhysicsSpace Code Samples

The PhysicsSpace also manages global physics settings. Typically, you can leave the defaults, and you don't need to change the following settings, but it's good to know what they are for:

<b>bulletAppState.getPhysicsSpace() Method</b>	<b>Usage</b>
setGravity(new Vector3f(0, -9.81f, 0));	Specifies the global gravity.
setAccuracy(1f/60f);	Specifies physics accuracy. The higher the accuracy, the slower the game. Decrease value if objects are passing through one another, or bounce oddly. (e.g. Change value from 1f/60f to something like 1f/80f.)
setMaxSubSteps(4);	Compensates low FPS: Specifies the maximum amount of extra steps that will be used to step the physics when the game fps is below the physics fps. This maintains determinism in physics in slow (low-fps) games. For example a maximum number of 2 can compensate for framerates as low as 30 fps (physics has a default accuracy of 60 fps). Note that setting this value too high can make the physics drive down its own fps in case its overloaded.
setWorldMax(new Vector3f(10000f, 10000f, 10000f)); setWorldMin(new Vector3f(-10000f, -10000f, -10000f));	Specifies the size of the physics space as two opposite corners (only applies to AXIS_SWEEP broadphase).

## Specify Physical Properties

After you have registered, attached, and added everything, you can adjust physical properties or apply forces.

On a RigidBodyControl, you can set the following physical properties.

<b>RigidBodyControl Method</b>	<b>Property</b>	<b>Examples</b>
setGravity(new Vector3f(0f,-9.81f,0f))	You can change the gravity of individual physics objects after they were added to the PhysicsSpace. Gravity is a vector pointing from this Spatial towards the source of gravity. The longer the vector, the stronger is gravity.  If gravity is the same absolute direction for all objects (e.g. on a planet surface), set this vector globally on the PhysicsSpace object and not individually.	For planet earth: new Vector3f(0f, -9.81f, 0f)

	If the center of gravity is relative (e.g. towards a black hole) then setGravity() on each Spatial to constantly adjust the gravity vectors at each tick of their update() loops.	
setMass(1f)	Sets the mass in kilogram. Dynamic objects have masses > 0.0f. Heavy dynamic objects need more force to be moved and light ones move with small amounts of force. Static immobile objects (walls, floors, including buildings and terrains) must have a mass of zero!	Person: 60f, ball: 1.0f Floor: 0.0f (!)
setFriction(1f)	Friction. Slippery objects have low friction. The ground has high friction.	Ice, slides: 0.0f Soil, concrete, rock: 1.0f
setRestitution(0.0f)	Bounciness. By default objects are not bouncy (0.0f). For a bouncy rubber object set this > 0.0f. Both the object and the surface must have non-zero restitution for bouncing to occur. This setting has an impact on performance, so use it sparingly.	Brick: 0.0f Rubber ball: 1.0f
setCcdMotionThreshold()	The amount of motion in 1 physics tick to trigger the continuous motion detection in moving objects that push one another. Rarely used, but necessary if your moving objects get stuck or roll through one another.	around 0.5 to 1 * object diameter

On a RigidBodyControl, you can apply the following physical forces:

<b>RigidBodyControl Method</b>	<b>Motion</b>
setPhysicsLocation()	Positions the objects. Do not use setLocalTranslation() for physical objects. Important: Make certain not to make CollisionShapes overlap when positioning them.
setPhysicsRotation()	Rotates the object. Do not use setLocalRotate() for physical objects.
setKinematic(true)	By default, RigidBodyControls are dynamic (kinematic=false) and are affected by forces. If you set kinematic=true, the object is no longer affected by forces, but it still affects others. A kinematic is solid, and must have a mass. (See detailed explanation below.)

## Kinematic vs Dynamic vs Static

All physical objects...

- must not overlap.
- can detect collisions and report several values about the impact.
- can respond to collisions dynamically, or statically, or kinematically.

Property	Static	Kinematic	Dynamic
<b>Examples</b>	Immobile obstacles: Floors, walls, buildings, ...	Remote-controlled solid objects: Airships, meteorites, elevators, doors; networked or remote- controlled NPCs; invisible “airhooks” for hinges and joints.	Interactive objects: Rolling balls, movable crates, falling pillars, zero-g space ship...
<b>Does it have a mass?</b>	no, 0.0f	yes <sup>1)</sup> , >0.0f	yes, >0.0f
<b>How does it move?</b>	never	setLocalTranslation();	setLinearVelocity(); applyForce(); setWalkDirection(); for CharacterControl
<b>How to place in scene?</b>	setPhysicsLocation(); setPhysicsRotation()	setLocalTranslation(); setLocalRotation();	setPhysicsLocation(); setPhysicsRotation()
<b>Can it move and push others?</b>	no	yes	yes
<b>Is it affected by forces? (Falls when it mid-air? Can be pushed by others?)</b>	no	no	yes
<b>How to activate this behaviour?</b>	setMass(0f); setKinematic(false);	setMass(1f); setKinematic(true);	setMass(1f); setKinematic(false);

## When Do I Use Kinematic Objects?

- Kinematics are solid and characters can “stand” on them.
- When they collide, Kinematics push dynamic objects, but a dynamic object never pushes a Kinematic.
- You can hang kinematics up “in mid-air” and attach other PhysicsControls to them using [hinges and joints](#). Picture them as “air hooks” for flying aircraft carriers, floating islands

in the clouds, suspension bridges, swings, chains...

- You can use Kinematics to create mobile remote-controlled physical objects, such as moving elevator platforms, flying blimps/airships. You have full control how Kinematics move, they never “fall” or “topple over”.

The position of a kinematic RigidBodyControl is updated automatically depending on its spatial's translation. You move Spatial with a kinematic RigidBodyControl programmatically, that means you write translation and rotation code in the update loop. You describe the motion of kinematic objects either by using methods such as `setLocalTranslation()` or `move()`, or by using a [MotionPath](#).

</div>

## Forces: Moving Dynamic Objects

Use the following methods to move dynamic physical objects.

PhysicsControl Method	Motion
<code>setLinearVelocity(new Vector3f(0f,0f,1f))</code>	Set the linear speed of this object.
<code>setAngularVelocity(new Vector3f(0f,0f,1f))</code>	Set the rotational speed of the object; the x, y and z component are the speed of rotation around that axis.
<code>applyCentralForce(...)</code>	Move (push) the object once with a certain moment, expressed as a Vector3f.
<code>applyForce(...)</code>	Move (push) the object once with a certain moment, expressed as a Vector3f. Optionally, you can specify where on the object the pushing force hits.
<code>applyTorque(...)</code>	Rotate (twist) the object once around its axes, expressed as a Vector3f.
<code>applyImpulse(...)</code>	An idealised change of momentum. This is the kind of push that you would use on a pool billiard ball.
<code>applyTorqueImpulse(...)</code>	An idealised change of momentum. This is the kind of push that you would use on a pool billiard ball.
<code>clearForces()</code>	Cancels out all forces (force, torque) etc and stops the motion.

It is technically possible to position PhysicsControls using `setLocalTranslation()`, e.g. to place them in their start position in the scene. However you must be very careful not to cause an “impossible state” where one physical object overlaps with another! Within the game, you typically use the setters shown here exclusively.

PhysicsControls also support the following advanced features:

<b>PhysicsControl Method</b>	<b>Property</b>
setCollisionShape(collisionShape)	Changes the collision shape after creation.
setCollideWithGroups() setCollisionGroup() addCollideWithGroup(COLLISION_GROUP_01) removeCollideWithGroup(COLLISION_GROUP_01)	Collision Groups are integer bit masks – enums are available in the CollisionObject. All physics objects are by default in COLLISION_GROUP_01. Two objects collide when the collideWithGroups set of one contains the Collision Group of the other. Use this to improve performance by grouping objects that will never collide in different groups (the engine saves times because it does not need to check on them).
setDamping(float, float)	The first value is the linear threshold and the second the angular. This simulates dampening of forces, for example for underwater scenes.
setAngularFactor(1f)	Set the amount of rotation that will be applied. A value of zero will cancel all rotational force outcome. (?)
setSleepingThreshold(float, float)	Sets the sleeping thresholds which define when the object gets deactivated to save resources. The first value is the linear threshold and the second the angular. Low values keep the object active when it barely moves (slow precise performance), high values put the object to sleep immediately (imprecise fast performance). (?)
setCcdMotionThreshold(0f)	Sets the amount of motion that has to happen in one physics tick to trigger the continuous motion detection in moving objects that push one another. This avoids the problem of fast objects moving through other objects. Set to zero to disable

	(default).
setCcdSweptSphereRadius(.5f)	Bullet does not use the full collision shape for continuous collision detection, instead it uses a “swept sphere” shape to approximate a motion, which can be imprecise and cause strange behaviors such as objects passing through one another or getting stuck. Only relevant for fast moving dynamic bodies.

You can `setApplyPhysicsLocal(true)` for an object to make it move relatively to its local physics space. You would do that if you need a physics space that moves with a node (e.g. a spaceship with artificial gravity surrounded by zero-g space). By default, it's set to false, and all movement is relative to the world.

</div>

## Best Practices

- **Multiple Objects Too Slow?** Do not overuse PhysicsControls. Although PhysicsControls are put to “sleep” when they are not moving, creating a world solely out of dynamic physics objects will quickly bring you to the limits of your computer's capabilities.  
**Solution:** Improve performance by replacing some physical SpatialS with non-physical SpatialS. Use the non-physical ones for non-solid things for which you do not need to detect collisions – foliage, plants, effects, ghosts, all remote or unreachable objects.
- **Complex Shape Too Slow?** Breaking the level into manageable pieces helps the engine improve performance: The less CPU-intensive `broadphase` filters out parts of the scene that are out of reach. It only calculates the collisions for objects that are actually close to the action.  
**Solution:** A huge static city or terrain model should never be loaded as one huge mesh. Divide the scene into multiple physics objects, with each its own CollisionShape. Choose the most simple CollisionShape possible; use mesh-accurate shapes only for the few cases where precision is more important than speed. For example, you can use the very fast `PlaneCollisionShape` for flat streets, floors and the outside edge of the scene, if you keep these pieces separate.
- **Eject?** If you have physical nodes jittering wildly and being ejected “for no apparent reason”, it means you have created an impossible state – solid objects overlapping. This can happen when you position solid spatialS too close to other solid spatialS, e.g.

when moving them with `setLocalTranslation()`.

**Solution:** Use the debug mode to make CollisionShapes visible and verify that CollisionShapes do not overlap.

```
bulletAppState.getPhysicsSpace().enableDebug(assetManager);
```

- **Buggy?** If you get weird behaviour, such as physical nodes passing through one another, or getting stuck for no reason.  
**Solution:** Look at the physics space accessors and change the accuracy and other parameters.
- **Need more interactivity?** You can actively *control* a physical game by triggering forces. You may also want to be able *respond* to collisions, e.g. by subtracting health, awarding points, or by playing a sound.  
**Solution:** To specify how the game responds to collisions, you use [Physics Listeners](#).

[physics](#), [documentation](#), [control](#)

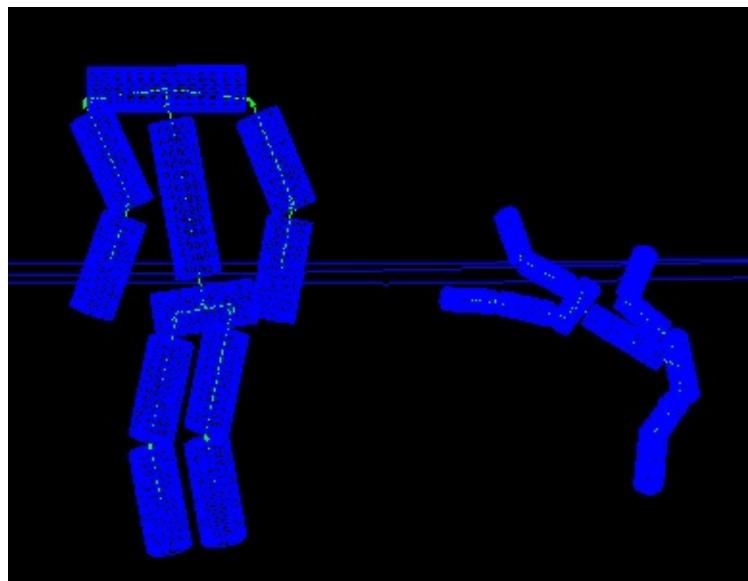
</div>

1) Inertia is calculated for kinematic objects, and you need mass to do that.

## title: Ragdoll Physics

# Ragdoll Physics

The jMonkeyEngine3 has built-in support for [jBullet physics](#) via the `com.jme3.bullet` package. Physics are not only responsible for handing collisions, but they also make [hinges](#) and [joints](#) possible. One special example of physical joints are ragdoll physics, shown here.



## Sample Code

- [TestRagDoll.java](#) (Tip: Click to pull the ragdoll up)
- [TestBoneRagdoll.java](#) – This ragdoll replaces a rigged model of a character in the moment it is “shot” to simulate a collapsing person. (Also note DoF of the limbs.)

## Preparing the Physics Game

1. Create a SimpleApplication with a [BulletAppState](#)
  - This gives us a PhysicsSpace for PhysicControls
2. Add a physical floor (A box collision shape with mass zero)

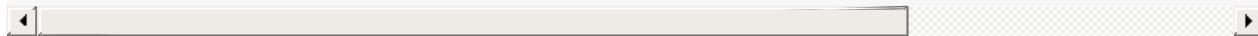
## Creating the Ragdoll

A ragdoll is a simple “person” (dummy) that you build out of cylinder collision shapes. The ragdoll has 11 limbs: 1 for shoulders, 1 for the body, 1 for hips; plus 2 arms and 2 legs that are made up of two limbs each. In your game, you will likely replace the cylinders with your own (better looking) limb models. In this example here we just use simple cylinders.

## Limbs

Since you're just creating the ragdoll for this example, all the limbs have the same shape, and you can write a simple helper method to create them. The function returns a PhysicsControl with CollisionShape with the width, height, location, and rotation (vertical or horizontal) that you specify. You choose a CapsuleCollisionShape (a cylinder with rounded top and bottom) so the limbs collide smoothly against one another.

```
private Node createLimb(float width, float height, Vector3f location,
 int axis = rotate ? PhysicsSpace.AXIS_X : PhysicsSpace.AXIS_Y);
 CapsuleCollisionShape shape = new CapsuleCollisionShape(width, height);
 Node node = new Node("Limb");
 RigidBodyControl rigidBodyControl = new RigidBodyControl(shape, mass);
 node.setLocalTranslation(location);
 node.addControl(rigidBodyControl);
 return node;
}
```



You write a custom helper method to initialize the limbs. Look at the screenshot above for orientation.

- All cylinders have the same diameter, 0.2f.
- You make the body and shoulders longer than the other limbs, 1.0f instead of 0.5f.
- You determine the coordinates for positioning the limbs to form a person.
- The shoulders and hips are *vertical* cylinders, this is why we set the rotation to true.

```

Node shoulders = createLimb(0.2f, 1.0f, new Vector3f(0.00f, 1.5f,
Node uArmL = createLimb(0.2f, 0.5f, new Vector3f(-0.75f, 0.8f,
Node uArmR = createLimb(0.2f, 0.5f, new Vector3f(0.75f, 0.8f,
Node lArmL = createLimb(0.2f, 0.5f, new Vector3f(-0.75f, -0.2f,
Node lArmR = createLimb(0.2f, 0.5f, new Vector3f(0.75f, -0.2f,
Node body = createLimb(0.2f, 1.0f, new Vector3f(0.00f, 0.5f,
Node hips = createLimb(0.2f, 0.5f, new Vector3f(0.00f, -0.5f,
Node uLegL = createLimb(0.2f, 0.5f, new Vector3f(-0.25f, -1.2f,
Node uLegR = createLimb(0.2f, 0.5f, new Vector3f(0.25f, -1.2f,
Node lLegL = createLimb(0.2f, 0.5f, new Vector3f(-0.25f, -2.2f,
Node lLegR = createLimb(0.2f, 0.5f, new Vector3f(0.25f, -2.2f,

```

You now have the outline of a person. But if you ran the application now, the individual limbs would fall down independently of one another – the ragdoll is still lacking joints.

## Joints

As before, you write a small helper method. This time its purpose is to quickly join two limbs A and B at the connection point that we specify.

- Convert A's and B's connectionPoint vector from world coordinate space to local coordinate space.
- Use a ConeJoint, a special joint that approximates the degree of freedom that limbs typically have. The ConeJoint constructor requires the two nodes, and the two local pivot coordinates that we just determined.
- Set the joints limits to allow swinging, but not twisting.

```

private PhysicsJoint join(Node A, Node B, Vector3f connectionPoint) {
 Vector3f pivotA = A.worldToLocal(connectionPoint, new Vector3f());
 Vector3f pivotB = B.worldToLocal(connectionPoint, new Vector3f());
 ConeJoint joint = new ConeJoint(A.getControl(RigidBodyControl.class),
 B.getControl(RigidBodyControl.class),
 pivotA, pivotB);
 joint.setLimit(1f, 1f, 0);
 return joint;
}

```

Use the helper method to connect all limbs with joints where they belong, at one end of the limb.

```

join(body, shoulders, new Vector3f(0.00f, 1.4f, 0));
join(body, hips, new Vector3f(0.00f, -0.5f, 0));
join(uArmL, shoulders, new Vector3f(-0.75f, 1.4f, 0));
join(uArmR, shoulders, new Vector3f(0.75f, 1.4f, 0));
join(uArmL, lArmL, new Vector3f(-0.75f, 0.4f, 0));
join(uArmR, lArmR, new Vector3f(0.75f, 0.4f, 0));
join(uLegL, hips, new Vector3f(-0.25f, -0.5f, 0));
join(uLegR, hips, new Vector3f(0.25f, -0.5f, 0));
join(uLegL, lLegL, new Vector3f(-0.25f, -1.7f, 0));
join(uLegR, lLegR, new Vector3f(0.25f, -1.7f, 0));

```

Now the ragdoll is connected. If you ran the app now, the doll would collapse, but the limbs would stay together.

## Attaching Everything to the Scene

We create one (non-physical) Node named `ragDoll`, and attach all other nodes to it.

```

ragDoll.attachChild(shoulders);
ragDoll.attachChild(body);
ragDoll.attachChild(hips);
ragDoll.attachChild(uArmL);
ragDoll.attachChild(uArmR);
ragDoll.attachChild(lArmL);
ragDoll.attachChild(lArmR);
ragDoll.attachChild(uLegL);
ragDoll.attachChild(uLegR);
ragDoll.attachChild(lLegL);
ragDoll.attachChild(lLegR);

```

To use the ragdoll in a scene, we attach its main node to the `rootNode`, and to the `PhysicsSpace`.

```

rootNode.attachChild(ragDoll);
bulletAppState.getPhysicsSpace().addAll(ragDoll);

```

## Applying Forces

To pull the doll up, you could add an input handler that triggers the following action:

```
Vector3f upforce = new Vector3f(0, 200, 0);
shoulders.applyContinuousForce(true, upforce);
```

We can use the action to pick the doll up and put it back on its feet, or what ever. Read more about [Forces](#) here.

## Detecting Collisions

Read the [Responding to a PhysicsCollisionEvent](#) chapter in the general physics documentation on how to detect collisions. You can detect collisions between limbs or between limbs and the floor, and trigger game events.

## Best Practices

If you experience weird behaviour in a ragdoll – such as exploding into pieces and then reassembling – check your collision shapes. Verify you did not position the limbs too close to one another when assembling the ragdoll. You typically see physical nodes being ejected when their collision shapes intersect, which puts physics in an impossible state.

[documentation](#), [physics](#), [character](#), [NPC](#), [forces](#), [collisions](#)

</div>

## **title: Recast Navigation for JME**

# **Recast Navigation for JME**

## **What is Recast Navigation**

Recast Navigation is C++ library for path-finding in 3D, continuous space. Recast has two big modules:

1. Recast
2. Detour

### **Recast**

Recast is state of the art navigation mesh construction tool set for games.

- It is automatic, which means that you can throw any level geometry at it and you will get robust mesh out
- It is fast which means swift turnaround times for level designers
- It is open source so it comes with full source and you can customize it to your heart's content.

### **Detour**

Recast is accompanied with Detour, path-finding and spatial reasoning toolkit. You can use any navigation mesh with Detour, but of course the data generated with Recast fits perfectly.

Detour offers simple static navigation mesh which is suitable for many simple cases, as well as tiled navigation mesh which allows you to plug in and out pieces of the mesh. The tiled mesh allows you to create systems where you stream new navigation data in and out as the player progresses the level, or you may regenerate tiles as the world changes.

## **jNavigation**

jNavigation is port Java library for [Recast navigation](#). jNavigation is the project in progress, and currently it enables building [Navigation meshes](#) and path-finding for one agent (bot).

# Example

In next code is described how the user should build navigation mesh, and query for it.

```
// Step 1. Initialize build config.
Config config = new Config();

Mesh mesh = ((Geometry) scene.getChild("terrain")).getMesh();

Vector3f minBounds = RecastBuilder.calculateMinBounds(mesh);
Vector3f maxBounds = RecastBuilder.calculateMaxBounds(mesh);

config.setMaxBounds(maxBounds);
config.setMinBounds(minBounds);
config.setCellSize(0.3f);
config.setCellHeight(0.2f);
config.setWalkableSlopeAngle(45);
config.setWalkableClimb(1);
config.setWalkableHeight(2);
config.setWalkableRadius(2);
config.setMinRegionArea(8);
config.setMergeRegionArea(20);
config.setBorderSize(20);
config.setMaxEdgeLength(12);
config.setMaxVerticesPerPoly(6);
config.setDetailSampleMaxError(1f);
config.setDetailSampleDistance(6);

RecastBuilder.calculateGridWidth(config);
RecastBuilder.calculatesGridHeight(config);

// Step 2. Rasterize input polygon soup.

//context is needed for logging that is not yet fully supported in
//It must NOT be null.
Context context = new Context();

// Allocate voxel heightfield where we rasterize our input data to.
Heightfield heightfield = new Heightfield();
if (!RecastBuilder.createHeightfield(context, heightfield, config))
```

```
 System.out.println("Could not create solid heightfield");
 return;
}

// Allocate array that can hold triangle area types.

// In Recast terminology, triangles are what indices in jME is. I l

// Find triangles which are walkable based on their slope and raster
char[] areas = RecastBuilder.markWalkableTriangles(context, config.
RecastBuilder.rasterizeTriangles(context, mesh, areas, heightfield,

// Step 3. Filter walkables surfaces.
// Once all geometry is rasterized, we do initial pass of filtering
// remove unwanted overhangs caused by the conservative rasterizati
// as well as filter spans where the character cannot possibly star
RecastBuilder.filterLowHangingWalkableObstacles(context, config.get
RecastBuilder.filterLedgeSpans(context, config, heightfield);
RecastBuilder.filterWalkableLowHeightSpans(context, config.getWalka

// Step 4. Partition walkable surface to simple regions.
// Compact the heightfield so that it is faster to handle from now
// This will result more cache coherent data as well as the neighbor
// between walkable cells will be calculated.
CompactHeightfield compactHeightfield = new CompactHeightfield();

if (!RecastBuilder.buildCompactHeightfield(context, config, heightf
 System.out.println("Could not build compact data");
 return;
}

if (!RecastBuilder.erodeWalkableArea(context, config.getWalkableRad
 System.out.println("Could not erode");
 return;
}

// Partition the heightfield so that we can use simple algorithm la
// There are 3 martitioning methods, each with some pros and cons:
```

```
// 1) Watershed partitioning
// - the classic Recast partitioning
// - creates the nicest tessellation
// - usually slowest
// - partitions the heightfield into nice regions without holes or overlaps
// - there are some corner cases where this method creates produces
// - holes may appear when a small obstacles is close to large ones
// - overlaps may occur if you have narrow spiral corridors (in which case
// * generally the best choice if you precompute the navmesh, use
// 2) Monotone partitioning
// - fastest
// - partitions the heightfield into regions without holes and overlaps
// - creates long thin polygons, which sometimes causes paths with many turns
// * use this if you want fast navmesh generation
String partitionType = "Sample partition watershed";

if (partitionType.equals("Sample partition watershed")) {
 if (!RecastBuilder.buildDistanceField(context, compactHeightfield))
 System.out.println("Could not build distance field");
 return;
}
if (!RecastBuilder.buildRegions(context, compactHeightfield, context.getGridSize()))
 System.out.println("Could not build watershed regions");
return;
}

if (partitionType.equals("Sample partition monotone")) {
 if (!RecastBuilder.buildRegionsMonotone(context, compactHeightfield))
 System.out.println("Could not build monotone regions");
 return;
}

// Step 5. Trace and simplify region contours.
// Create contours.
ContourSet contourSet = new ContourSet();

if (!RecastBuilder.buildContours(context, compactHeightfield, 2f, context.getGridSize()))
 System.out.println("Could not create contours");
```

```
 return;
 }

 // Step 6. Build polygons mesh from contours.
 // Build polygon navmesh from the contours.
 PolyMesh polyMesh = new PolyMesh();

 if (!RecastBuilder.buildPolyMesh(context, contourSet, config.getMaxVerticesPerPoly()))
 System.out.println("Could not triangulate contours");
 return;
}

// Step 7. Create detail mesh which allows to access approximate heightmap
PolyMeshDetail polyMeshDetail = new PolyMeshDetail();

if (!RecastBuilder.buildPolyMeshDetail(context, polyMesh, compactHeightMap))
 System.out.println("Could not build detail mesh.");
return;
}

// (Optional) Step 8. Create Detour data from Recast poly mesh.
// The GUI may allow more max points per polygon than Detour can handle.
// Only build the detour navmesh if we do not exceed the limit.
if (config.getMaxVertsPerPoly() > DetourBuilder.VERTS_PER_POLYGON())
 return;
}

NavMeshCreateParams createParams = new NavMeshCreateParams();
createParams.getData(polyMesh);
createParams.getData(polyMeshDetail);
//setting optional off-mesh connections (in my example there are none)
createParams.getData(config);
createParams.setBuildBvTree(true);

char[] navData = DetourBuilder.createNavMeshData(createParams);

if (navData == null) {
 System.out.println("Could not build Detour navmesh.");
 return;
}
```

```
NavMesh navMesh = new NavMesh();

if (!navMesh.isAllocationSuccessful()) {
 System.out.println("Could not create Detour navmesh");
 return;
}

Status status;
status = navMesh.init(navData, TileFlags.DT_TILE_FREE_DATA.value());
if (status.isFailed()) {
 System.out.println("Could not init Detour navmesh");
 return;
}

NavMeshQuery query = new NavMeshQuery();
status = query.init(navMesh, 2048);
if (status.isFailed()) {
 System.out.println("Could not init Detour navmesh query");
 return;
}
```

After this (if everything is successful) you can use methods in `query` that was created for path-finding purposes.

## How to get jNavigation

There is 2 ways to get jNavigation:

- as plugin form
- as developmental project

### Plugin

You can download “stable” version from [repository](#)

### Developmental project

Instructions for downloading and setting it up:

- Download C++ wrapper from [jNavigationNative repository](#)

- Build downloaded project with C++ compiler
- Download java library from [jNavigation repository](#)
- In Java project in class `com.jme3.ai.navigation.utils.RecastJNI.java` change URL to where your build of C++ project is.

```
static {
 // the URL that needs to be changed
 System.load("../jNavigationNative.dll");
}
```

If there is problem with building C++ project see [link](#).

## Questions & Suggestions

- For suggestion and/or question on jNavigation post on [forum](#)
- For question on Recast (C++ library) ask on [Google groups](#)

## Source

- [jNavigation plugin repository](#)
- [Developmental jNavigation repository](#)
- [Developmental jNavigationNative repository](#)

## Useful links

- [How to build the native recast bindings](#)
- [Study: Navigation Mesh Generation](#)
- [Documentation of C++ Recast library](#) It can be useful for tracing bugs.

</div>

## title: Saving and Loading Games (.j3o)

# Saving and Loading Games (.j3o)

Spatial (that is Nodes and Geometries) can contain audio and light nodes, particle emitters, controls, and user data (player score, health, inventory, etc). For your game distribution, you must convert all original models to a faster binary format. You save individual Spatial as well as scenes using `com.jme3.export.binary.BinaryExporter`.

The jMonkeyEngine's binary file format is called `.j3o`. You can convert, view and edit `.j3o` files and their materials in the jMonkeyEngine [SDK](#) and compose scenes (this does not include editing meshes). For the conversion, you can either use the BinaryExporters, or a context menu in the SDK.

The jMonkeyEngine's serialization system is the `com.jme3.export.Savable` interface. JME3's BinaryExporter can write standard Java objects, JME3 objects, and primitive data types that are included in a [spatial's user data](#). If you use custom game data classes, see below how to make them "Savable".

There is also a `com.jme3.export.xml.XMLExporter` and `com.jme3.export.xml.XMLImporter` that similarly converts jme3 spatial to an XML format. But you wouldn't use that to load models at runtime (quite slow).

</div>

## Sample Code

- [TestSaveGame.java](#)

## Saving a Node

The following example overrides `stop()` in SimpleApplication to save the rootNode to a file when the user quits the application. The saved rootNode is a normal `.j3o` binary file that you can open in the [SDK](#).

Note that when you save a model that has textures, the references to those textures are stored as absolute paths, so when loading the `j3o` file again, the textures have to be accessible at the exact location (relative to the assetmanager root, by default the `assets`

directory) they were loaded from. This is why the SDK manages the conversion on the project level.

```
/* This is called when the user quits the app. */
@Override
public void stop() {
 String userHome = System.getProperty("user.home");
 BinaryExporter exporter = BinaryExporter.getInstance();
 File file = new File(userHome+"/Models/"+ "MyModel.j3o");
 try {
 exporter.save(rootNode, file);
 } catch (IOException ex) {
 Logger.getLogger(Main.class.getName()).log(Level.SEVERE, "Err
 }
 super.stop(); // continue quitting the game
}
```

</div>

## Loading a Node

The following example overrides `simpleInitApp()` in `SimpleApplication` to load `Models/MyModel.j3o` when the game is initialized.

```
@Override
public void simpleInitApp() {
 String userHome = System.getProperty("user.home");
 assetManager.registerLocator(userHome, FileLocator.class);
 Node loadedNode = (Node)assetManager.loadModel("Models/MyModel.j3o");
 loadedNode.setName("loaded node");
 rootNode.attachChild(loadedNode);
}
```

Here you see why we save user data inside spatial – so it can be saved and loaded together with the .j3o file. If you have game data outside Spatial, you have to remember to save() and load(), and get() and set() it yourself.

</div>

# Custom Savable Class

JME's BinaryExporter can write standard Java objects (String, ArrayList, buffers, etc), JME objects (Savables, such as Material), and primitive data types (int, float, etc). If you are using any custom class together with a Spatial, then the custom class must implement the

`com.jme3.export.Savable` interface. There are two common cases where this is relevant:

- The Spatial is carrying any [Custom Controls](#).

Example: You used something like `myspatial.addControl(myControl);`

- The Spatial's user data can contain a custom Java object.

Example: You used something like `myspatial.setUserData("inventory", myInventory);`

If your custom classes (the user data or the Controls) do not implement Savable, then the BinaryImporter/BinaryExporter cannot save the Spatial!

So every time you create a custom Control or custom user data class, remember to implement Savable:

```

import com.jme3.export.InputCapsule;
import com.jme3.export.JmeExporter;
import com.jme3.export.JmeImporter;
import com.jme3.export.OutputCapsule;
import com.jme3.export.Savable;
import com.jme3.material.Material;
import java.io.IOException;

public class MyCustomClass implements Savable {
 private int someIntValue; // some custom user data
 private float someFloatValue; // some custom user data
 private Material someJmeObject; // some custom user data

 ...
 // your other code...
 ...

 public void write(JmeExporter ex) throws IOException {
 OutputCapsule capsule = ex.getCapsule(this);
 capsule.write(someIntValue, "someIntValue", 1);
 capsule.write(someFloatValue, "someFloatValue", 0f);
 capsule.write(someJmeObject, "someJmeObject", new Material());
 }

 public void read(JmeImporter im) throws IOException {
 InputCapsule capsule = im.getCapsule(this);
 someIntValue = capsule.readInt("someIntValue", 1);
 someFloatValue = capsule.readFloat("someFloatValue", 0f);
 someJmeObject = capsule.readSavable("someJmeObject", new Material());
 }
}

```

To make a custom class savable:

1. Implement `Savable` and add the `write()` and `read()` methods as shown in the example above.
2. Do the following for each non-temporary class field:
  - Add one line that `write()` s the data to the JmeExport output capsule.
    - Specify the variable to save, give it a String name (can be the same as the

variable name), and specify a default value.

- Add one line that `read...()` is the data to the `JmeImport` input capsule.
  - On the left side of the assignment, specify the class field that you are restoring
  - On the right side, use the appropriate `capsule.read...()` method for the data type. Specify the String name of the variable (must be the same as you used in the `write()` method ), and again specify a default value.

As with all serialization, remember that if you ever change data types in custom classes, the updated `read()` methods will no longer be able to read your old files. Also there has to be a constructor that takes no Parameters.

[convert](#), [j3o](#), [models](#), [load](#), [save](#), [documentation](#), [serialization](#), [import](#), [export](#), [spatial](#), [node](#), [mesh](#), [geometry](#), [scenegraph](#), [sdk](#)

</div>

## **title: Taking Screenshots**

# **Taking Screenshots**

The com.jme3.app.state.ScreenshotAppState enables your users to take screenshots of the running game.

You activate this feature as follows in your simpleInitApp() method:

```
ScreenshotAppState screenShotState = new ScreenshotAppState();
this.stateManager.attach(screenShotState);
```

The default screenshot key is KeyInput.KEY\_SYSRQ, also known as “System Request / Print Screen” key. On Mac keyboards, this key does not exist, so on Mac OS you take screenshots using Command+Shift+3 (fullscreen) or Command+Shift+4 (windowed: press space to select a window and then click).

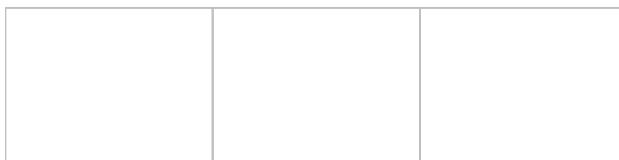
The screenshot is saved to the user directory.

## title: Shapes

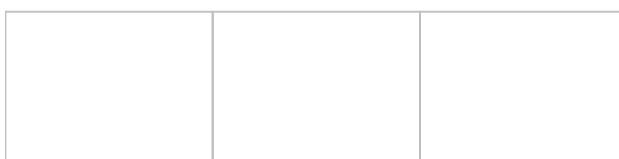
# Shapes

The simplest type of Meshes are the built-in JME Shapes. You can create Shapes without using the AssetManager.

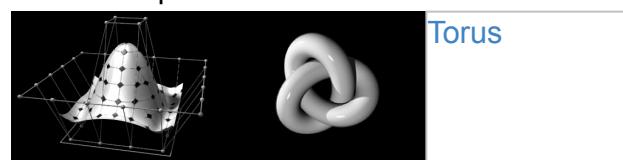
## 3D shapes



- `com.jme3.scene.shape.Box` – A cube or cuboid. Single-sided Quad faces (outside only).
- `com.jme3.scene.shape.StripBox` – A cube or cuboid. Solid filled faces (inside and outside).
- `com.jme3.scene.shape.Cylinder` – A disk or pillar.
- `com.jme3.scene.shape.Sphere` – A ball or ellipsoid.



- `com.jme3.scene.shape.Dome` – A semi-sphere, e.g. SkyDome.
  - For a cone, set the Dome's `radialSamples>4` and `planes=2`.
  - For a pyramid, set the Dome's `radialSamples=4` and `planes=2`.
- `com.jme3.scene.shape.Torus` – An single-holed torus or “donut”.
- `com.jme3.scene.shape.PQTorus` – A parameterized torus. A PQ-Torus looks like a



donut knotted into spirals.

- `com.jme3.scene.shape.Surface` – A curved surface (called **NURBS**) described by knots, weights and control points. Compare with `shape.Curve`.

## Non-3D shapes

- com.jme3.scene.shape.Quad – A flat 2D rectangle (single-sided, center is in bottom-left corner)
- com.jme3.scene.shape.Line – A straight 1D line defined by a start and end point.
- com.jme3.scene.shape.Curve – A curved 1D spline. Compare with shape.Surface.

## com.jme3.math versus com.jme3.shape?

Do not mix up these visible com.jme3.shapes with similarly named classes from the com.jme3.math package. Choose the right package when letting your IDE fill in the import statements!

- com.jme3.math.Line – is invisible, has a direction, goes through a point, infinite length.
- com.jme3.math.Ray – is invisible, has a direction and start point, but no end.
- com.jme3.math.Spline – is an invisible curve.
- etc

These maths objects are invisible and are used for collision testing (ray casting) or to describe motion paths. They cannot be wrapped into a Geometry.

## Usage

### Basic Usage

To add a shape to the scene:

1. Create the base mesh shape.
2. Wrap the mesh into a Geometry.
3. Assign a Material to the Geometry.
4. Attach the Geometry to the rootNode to make it visible.

Create one static shape as mesh and use it in several geometries, or clone() the geometries.  
</div>

### Complex Shapes

You can compose more complex custom Geometries out of simple Shapes. Think of the buildings in games like Angry Birds, or the building blocks in Second Life (“prims”) and in Tetris (“Tetrominos”).

1. Create a Node. By default it is located at the origin (0/0/0) – leave the Node there for

now.

2. Create your shapes and wrap each into a Geometry, as just described.
3. Attach each Geometry to the Node.
4. Arrange the Geometries around the Node (using `setLocalTranslation()`) so that the Node is in the center of the new constellation. The central Node is the pivot point for transformations (move/scale/rotate).
5. Move the pivot Node to its final location in the scene. Moving the pivot Node moves the attached constellation of Geometries with it.

The order is important: First arrange around origin, then transform. Otherwise, transformations are applied around the wrong center (pivot). Of course, you can attach your constellation to other pivot Nodes to create even more complex shapes (a chair, a furnished room, a house, a city, ...), but again, arrange them around the origin first before you transform them. Obviously, such composed Geometries are simpler than hand-sculpted meshes from a mesh editor.

## Code Examples

Create the Mesh shape:

```
Sphere mesh = new Sphere(32, 32, 10, false, true);
```

```
Dome mesh = new Dome(Vector3f.ZERO, 2, 4, 1f, false); // Pyramid
```

```
Dome mesh = new Dome(Vector3f.ZERO, 2, 32, 1f, false); // Cone
```

```
Dome mesh = new Dome(Vector3f.ZERO, 32, 32, 1f, false); // Small hemi
```

```
Dome mesh = new Dome(Vector3f.ZERO, 32, 32, 1000f, true); // SkyDome
```

```
PQTorus mesh = new PQTorus(5, 3, 2f, 1f, 32, 32); // Spiral torus
```

```
PQTorus mesh = new PQTorus(3, 8, 2f, 1f, 32, 32); // Flower torus
```

Use one of the above examples together with the following geometry in a scene:

```
Geometry geom = new Geometry("A shape", mesh); // wrap shape into geometry
Material mat = new Material(assetManager,
 "Common/MatDefs/Misc/ShowNormals.j3md"); // create material
geom.setMaterial(mat); // assign material to geometry
// if you want, transform (move, rotate, scale) the geometry.
rootNode.attachChild(geom); // attach geometry to scene graph
```

## See also

\* [Optimization](#) – The GeometryBatchFactory class combines several of your shapes with the same texture into one mesh with one texture.

[spatial](#), [node](#), [mesh](#), [geometry](#), [scenegraph](#)  
</div>

## title: How to add a Sky to your Scene

# How to add a Sky to your Scene

Here is an example for how you add a static horizon (a background landscape and a sky) to a scene. Having a discernable horizon with a suitable landscape (or space, or ocean, or whatever) in the background makes scenes look more realistic than just a single-colored “sky” background.

</div>

## Adding the Sky

Adding a sky is extremely easy using the `com.jme3.util.SkyFactory`.

```
rootNode.attachChild(SkyFactory.createSky(
 assetManager, "Textures/Sky/Bright/BrightSky.dds", false)
```

To add a sky you need to supply:

1. The `assetManager` object to use
2. A cube or sphere map texture of the sky
3. Set the boolean to true if you are using a sphere map texture. For a cube map, use `false`.

Tip: Cube map is the default. You would know if you had created a sphere map.

Internally, the `SkyFactory` calls the following methods:

1. `sky.setQueueBucket(Bucket.Sky);` makes certain the sky is rendered in the right order, behind everything else.
2. `sky.setCullHint(Spatial.CullHint.Never);` makes certain that the sky is never culled.
3. The `SkyFactory` uses the internal jME3 material definition `sky.j3md`. This Material definition works with sphere and cube maps.

# Creating the Textures

As the sky texture we use the sample BrightSky.dds file from jme3test-test-data.

How to create a sky textures?

- There are many tools out there that generate cube and sphere maps. Examples for landscape texture generators are Terragen or Bryce.
- The actual texture size does not matter, as long as you add the Sky Geometry to the Sky bucket: Everything in the sky bucket will always be infinitely far away behind everything else, and never intersect with your scene. Of course the higher the resolution, the better it will look. On the other hand, if the graphic is too big, it will slow the game down.
- A box or sphere map is the simplest solution. But you can use any Node as sky, even complex sets of geometries and quads with animated clouds, blinking stars, city skylines, etc.
- JME3 supports cube maps in PNG, JPG, or (compressed) DDS format.

Box or Sphere?

- If you have access to cube map textures, then use a SkyBox
  - [SkyBox examples](#)
- If you have access to sphere map textures – specially projected sky images that fit inside a sphere – then you use a SkySphere or SkyDome.
  - [SkySphere example](#)

</div>

## **title: Spatial**

# **Spatial**

This is an introduction to the concept of **Spatial**s, the elements of the 3D scene graph. The scene graph is a data structure that manages all objects in your 3D world. For example, the scene graph keeps track of the 3D models that you load and position. When you extend a Java class from `com.jme3.app.SimpleApplication`, you automatically inherit the scene graph and its `rootNode`.

The `rootNode` is the central element of the scene graph. Even if the scene graph is empty, it always contains at least the `rootNode`. We *attach* **Spatial**s to the `rootNode`. Attached **Spatial**s are always in a *parent-child relationship*. Every time you attach a **Spatial** to something, it is implicitly detached from its previous parent. A **Spatial** can have only one parent. A **Spatial** can have several children.

If you think you need to understand the scene graph concept better, please read [Scenegraph for dummies](#) first.

## **Node versus Geometry**

In your Java code, a **Spatial** is either an instance of `com.jme3.scene.Node` or a `com.jme3.scene.Geometry` instance. You use the two types of **Spatial**s for different purposes:



	<b>com.jme3.scene.Spatial</b>	
<b>Purpose:</b>	A Spatial is an abstract data structure that stores user data and transformations (= translation, rotation, scale) of elements of the 3D scene graph. Spatial can be saved and loaded using the <a href="#">Asset Manager</a> .	
	<b>com.jme3.scene.Geometry</b>	<b>com.jme3.scene.Node</b>
<b>Visibility:</b>	A Geometry represents a <b>visible</b> 3D object in the scene graph.	A Node is an <b>invisible “handle”</b> for a group of Spatial in the scene graph.
<b>Purpose:</b>	Use Geometries to represent an object's <b>look</b> : Every Geometry contains a polygon mesh and a material, specifying its shape, color, texture, and opacity/transparency. You attach Geometries to Nodes.	Use Nodes to <b>structure and group</b> Geometries and other Nodes. Every Node is attached to one parent node, and each node can have zero or more children (Nodes or Geometries) attached to itself. <b>When you transform (move, rotate, etc) a parent node, all its children are transformed (moved, rotated, etc).</b>
<b>Content:</b>	Transformations; custom user data; mesh and material;	Transformations; custom user data; no mesh, no material.
<b>Examples:</b>	Box, sphere, player, building, terrain, vehicle, missiles, NPCs, etc...	rootNode, guiNode, audioNode, a custom grouping node such as vehicleNode or shipNode with passengers attached, etc.

You never create a Spatial with `Spatial s = new Spatial();` ! A Spatial is an abstract concept, like a mammal (there is no actual creature called “mammal” walking around here). You create either a com.jme3.scene.Node or com.jme3.scene.Geometry instance. Some methods, however, require a `Spatial` type as argument: This is because they are able to accept both Nodes and Geometries as arguments. In this case, you simply *cast* a Node or Geometry to Spatial.

</div>

## Mesh

The polygon [Mesh](#) inside a Geometry can be one of three things:

- **Shapes:** The simplest type of Meshes are jME's default [Shapes](#) such as cubes and spheres. You can use several Shapes to build complex Geometries. Shapes are built-in and can be created without using the AssetManager.
- **3D Models:** [3D models and scenes](#) are also made up of meshes, but are more complex

than Shapes. You create Models and Scenes in external 3D Mesh Editors and export them as Ogre XML or Wavefront OBJ. Use the [Asset Manager](#) to load models into a your jME3 game.

- **Custom Meshes:** Advanced users can create [Custom Meshes](#) programmatically.

## What is a Clone?

Cloned spatial share the same mesh, while each cloned spatial can have its own local transformation (translation, rotation, and scale) in the scene. This means you only use `clone()` on spatial whose meshes never change. The most common use case for cloning is when you use several Spatial that are based on the same [Shapes](#) (e.g. trees, crates).

The second use case is: When you load a model using `loadModel()` from the AssetManager, you may automatically get a `clone()` ed object. In particular:

- If the model is not animated (it has no `AnimControl`), jME loads a clone. All clones share one mesh object in order to use less memory.
- If the model is animated (it has a `AnimControl`), then `loadModel()` duplicates the mesh for each loaded instance. (Uses more memory, but can animate.)

Usually there is no need to manually use any of the `clone()` methods on models. Using the [Asset Manager](#)'s `loadModel()` method will automatically do the right thing for your models.

“Box worlds” are not made up of statically cloned `Box()` shapes, this would still be too slow for large worlds. To learn how to make real fast box worlds, search the web for *voxelization* techniques.

</div>

## How to Add Fields and Methods to a Spatial

You can include custom user data –that is, custom Java objects and methods– in Nodes and Geometries. This is very useful for maintaining information about a game element, such as health, budget, ammunition, inventory, equipment, etc for players, or landmark locations for terrains, and much more.

You want to add custom accessor methods to a spatial? Do not extend `Node` or `Geometry`, use [Custom Controls](#) instead. You want to add custom fields to a spatial? Do not extend `Node` or `Geometry`, use the built-in `setUserData()` method instead. Where ever the Spatial is accessible, you can easily access the object's class fields (user data) and accessors (control methods) this way.

This first example adds an integer field named `health` to the Spatial `playerNode`, and initializes it to 100.

```
playerNode.setUserData("health", 100);
```

The second example adds a set of custom accessor methods to the player object. You create a [custom PlayerControl\(\) class](#) and you add this control to the Spatial:

```
playerNode.addControl(new PlayerControl());
```

In your `PlayerControl()` class, you define custom methods that set and get your user data in the `spatial` object. For example, the control could add accessors that set and get the player's health:

```
public int getHealth() {
 return (Integer)spatial.getUserData("health");
}
public void setHealth(int h) {
 spatial.setUserData("health", h);
}
```

Elsewhere in your code, you can access this data wherever you have access to the Spatial `playerNode`.

```
health = playerNode.getControl(PlayerControl.class).getHealth();
...
playerNode.getControl(PlayerControl.class).setHealth(99);
```

- You can add as many data objects (of String, Boolean, Integer, Float, Array types) to a Spatial as you want. Just make sure to label them with unique case-sensitive strings (`health`, `Inventory`, `equipment`, etc).
- The saved data can even be a custom Java object if you make the custom Java class [implement the Savable interface!](#)
- When you save a Spatial as a .j3o file, the custom data is saved, too, and all Savables are restored the next time you load the .j3o!

This is how you list all data keys that are already defined for one Spatial:

```
for(String key : spatial.getUserDataKeys()){
 System.out.println(spatial.getName()+'s keys: '+key);
}
```

## How to Access a Named Sub-Mesh

Often after you load a scene or model, you need to access a part of it as an individual Geometry in the scene graph. Maybe you want to swap a character's weapon, or you want to play a door-opening animation. First you need to know the unique name of the sub-mesh.

1. Open the model in a 3D mesh editor, or in the jMonkeyEngine SDK's Scene Composer.
2. Find out the existing names of sub-meshes in the model.
3. Assign unique names to sub-meshes in the model if necessary.

In the following example, the Node `house` is the loaded model. The sub-meshes in the Node are called its children. The String, here `door 12`, is the name of the mesh that you are searching.

```
Geometry submesh = (Geometry) houseScene.getChild("door 12");
```

## What is Culling?

There are two types of culling: Face culling, and view frustum culling.

**Face culling** means not drawing certain polygons of a mesh. Face culling behaviour is a property of the material.

Usage: The “inside” of a mesh (the so called backface) is typically never visible to the player, and as an optimization, the `Back` mode skips calculating all backfaces by default. Activating the `off` or `Front` modes can be useful when you are debugging [custom meshes](#) and try to identify accidental inside-out faces.

You can switch the `com.jme3.material.RenderState.FaceCullMode` to either:

- `FaceCullMode.Back` (default) – Only the frontsides of a mesh are drawn. Backface culling is the default behaviour.
- `FaceCullMode.Front` – Only the backsides of a mesh are drawn. A mesh with frontface culling will most likely be invisible. Used for debugging “inside-out” custom meshes.
- `FaceCullMode.FrontAndBack` – Use this to make a mesh temporarily invisible.
- `FaceCullMode.off` – Every side of the mesh is drawn. Looks normal, but slows down

large scenes.

Example:

```
material.getAdditionalRenderState().setFaceCullMode(FaceCullMode.Fr
```

**View frustum culling** refers to not drawing (and not even calculating) certain whole models in the scene. At any given moment, half of the scene is behind the player and out of sight anyway. View frustum culling is an optimization to not calculate scene elements that are not visible – elements that are “outside the view frustum”.

The decision what is visible and what not, is done automatically by the engine (`cullHint.Dynamic`). Optionally, you can manually control whether the engine culls individual spatial (and children) from the scene graph:

- `CullHint.Dynamic` – Default, faster because it doesn't waste time with objects that are out of view.
- `CullHint.Never` – Calculate and draw everything always (even if it does not end up on the user's screen because it's out of sight). Slower, but can be used while debugging custom meshes.
- `CullHint.Always` – The whole spatial is culled and is not visible. A fast way to hide a Spatial temporarily. Culling a Spatial is faster than detaching it, but it uses more memory.
- `CullHint.Inherit` – Inherit culling behaviour from parent node.

Example:

```
spatial.setCullHint(CullHint.Never); // always drawn
```

</div>

## See also

- [Optimization](#) – The GeometryBatchFactory class batches several Geometries into meshes with each their own texture.
- [Traverse SceneGraph](#) – Find any Node or Geometry in the scenegraph.

[spatial](#), [node](#), [mesh](#), [geometry](#), [scenegraph](#)

</div>

## title: Optimizing Your Game Using Statistics

# Optimizing Your Game Using Statistics

When you create a SimpleApplication, you see the default StatsView and FpsView in the left corner. The StatsView displays object statistics that are used during development, for example for debugging and optimization. Below the StatsView is the FpsView that displays the frames that jMonkeyEngine can render per second.

The main use case of these statistics is to find out why the application may be running slow and where to start fixing the performance.

The StatsView + FpsView look like this example:

```
FrameBuffers (M) = 2
FrameBuffers (F) = 2
FrameBuffers (S) = 2
Textures (M) = 7
Textures (F) = 3
Textures (S) = 3
Shaders (M) = 6
Shaders (F) = 3
Shaders (S) = 4
Objects = 24
Uniforms = 31
Triangles = 582
Vertices = 1148
Frames per Second: 30
```

## On/Off

You switch the StatsView on or off in the simpleInitApp() method by setting a boolean:

```
setDisplayFps(false); // to hide the FPS
setDisplayStatView(false); // to hide the statistics
```

# Terminology

Types of items counted:

- FrameBuffers: Total number of rendering surfaces used for off-screen rendering and render-to-texture functionality.
- Textures: Total number of distinct textures used in the scene.
- Shaders: Total number of shaders used for effects (shading, blur, lighting, glow, etc).
- Objects: Total number of objects in the OpenGL pipeline. That is, objects attached to the rootNode and guiNode, etc.
- Uniforms: Total number of shader uniforms. Uniforms are predefined variables used as parameters in shader calculations, containing data such as matrices, vectors, time, and colors.
- Triangles: Total number of triangles (faces) of the meshes of all objects.
- Vertices: Total number of vertices (corner points) of the meshes of all the objects.

Types of statistics:

- **(M) = Memory** – Number of items currently in OpenGL memory.
- **(F) = Frame** – Number of items used by current frame (visible).
- **(S) = Switches** – Number of items that were state switched in the last frame.

The StatsView does not include any Physics statistics.

## How to Interpret the FPS when Optimizing

The FPS (frames per second) shows you how fast jME runs the update loop. If the FPS values goes below 30, the game slows down and runs jerky, which makes the game either frustrating or impossible to play. You need either decrease the number of operations in the update loop, or decrease memory usage (object count).

If your application grows more and more sluggish, deactivate or decrease one feature set at a time: Deactivate drop shadows, physics, anti-aliasing... Try fewer light sources, fewer NPCs, fewer samples in spheres (i.e. less smooth spheres). Temporarily replace the scene (or parts of it) with a simpler test scene to check whether the scene has too many triangles, etc.

Keep an eye on FPS and the StatsView and find out which element has the biggest impact on performance. This is where you start optimizing.

## How to Interpret The Statistics

To interpret the numbers correctly, consider that the 14 lines of text themselves already count as 14 objects with 914 vertices. You need to subtract these values from the totals for smaller performance experiments.

What do you want to avoid?

1. FrameBuffers: If you don't use any post-processing effects (`FilterPostProcessor`), this count should be zero. The more effects you use, the more FrameBuffers are in use. If this value is high while others are normal, and your game is sluggish, you can speed up the application by using fewer post-processing effects.
2. The Object Count (Batch Count) is a very important value that indicates how many geometries were rendered in the last frame. In general, if you keep the object count around 100-200, your game will be fast and responsive. If the count is permanently higher, hand-code rules that detach remote objects, or optimize a complex multi-material scene using: `GeometryBatchFactory.optimize(complexNode, true);` or a [Texture Atlas](#).
3. Triangle Counts. If your game is sluggish and triangle (polygon) count is high, then you are rendering too many, too detailed meshes. Tell your graphic designers to create models with lower polygon counts, or use a [Level of Detail](#) (LOD) optimization. The limit is around 100'000 vertices for a scene, considering that the slowest currently used graphic cards cannot handle anything beyond that.
4. Are any counts constantly increasing right before the game slows down or runs out of memory? Check whether you are accidentally adding objects in a loop, instead of only once.
5. Verify that the numbers are plausible. If you think you generated a test scene out of "a few boxes", but the StatsView shows ten thousands of triangles, then you probably have extra objects out of sight somewhere (due to barely visible materials, overlapping with other objects, scaled too big or too small to see, etc).
  - Example: In a test scene made up of boxes, you'd expect a vertex:triangle:object ratio of 8:12:1.
  - Terrains, models and spheres have higher counts, depending on their size and [Level of Detail](#) (LOD). A high-poly model looks pretty in Blender, but you must find a lower-poly, low-LOD compromise if you want several large objects in one scene!
6. If S (objects being switched) are high compared to F (objects used), then you use or generate too many different Materials (etc). You are unnecessarily forcing jME to re-load and re-bind objects (= Switches), which is bad for performance. Also if you have many transparent materials in your scene, this results in more switches, and you should use fewer transparent materials.
7. If the M values (objects in memory) are high compared to F (objects used), that means a lot more GL objects are kept in memory than are actually used. This can happen in large scenes with many materials. Consider breaking the scene up and detaching

objects while they are out of sight, so the built-in culling can optimize the scene.

What goal are you trying to achieve in general?

- The values for (M) and (F) should be within the same order of magnitude. This means your code only loads objects that it actually needs, and that the hardware can actually handle.
- It's okay if Switches (S) are lower than Used in Current Frame (F).
- The FPS should be 30 or more on the slowest hardware that you target.
- 10'000-50'000 vertices is a typical average value for a scene.

</ol>

---

See also:

- [What's a good triangle count?](#) Forum discussion
- [Level of Detail](#)

</div>

## title: Steer Behaviors

# Steer Behaviors

Steer behaviors allows you to control the locomotion of “characters”, this can reduce drastically the time needed to develop a game since for almost every game we need to set how these “characters” will be moving around the scene.

**Steer behaviors in action:** [youtube\\_yyzntsgv00](#)

Be sure that you have checked the demos: They can be downloaded here:  
[jmesteer.bdevel.org](http://jmesteer.bdevel.org)

## First steps

The steer behaviors AI is integrated with MonkeyBrains so before start coding be sure that you have checked the [monkey brains documentation](#)

Be sure to create a reference to MonkeyBrains from your project.

Finally, do not forget to import the `com.jme3.ai.agents.behaviors.npc.steering` package.

## Overview

### Available behaviours:

- Move
- Seek
- Arrive
- Flee
- Pursuit
- Leader follow
- Evade
- Cohesion
- Alignment
- Obstacle Avoidance
- Unaligned obstacle avoidance

- Hide
- Slow
- Queuing
- Containment
- Path follow
- Wall approach
- Wander Area
- Simple Wander
- Relative wander
- Sphere wander
- Box explore
- Separation

All the behaviours extend from the `AbstractSteeringBehavior` class.

## Adding a steer behavior

Create instances from the steer behavior classes, They are located in the `com.jme3.ai.agents.behaviors.npc.steering` package.

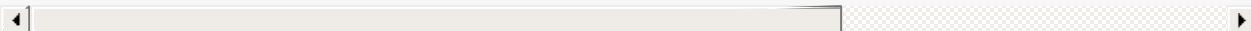
If we want to add more than one steer behavior, we need to create a container:

If you add more than one steer behavior to a `SimpleMainBehavior` it will cause problems in the rotation of the agents.

Container	Purpose
CompoundSteeringBehavior	Contains and merges several <code>AbstractSteeringBehavior</code> instances
BalancedCompoundSteeringBehavior	Each force generated inside this container is reduced in relation with a proportion factor: “Partial Force” / “Total container force”

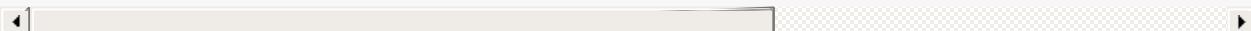
Once we know which container fits better for our agent, We create a new instance and add all the behaviors that we need:

```
SimpleMainBehaviour mainBehavior = new SimpleMainBehavior(myAgent);
CompoundSteeringBehavior steer = new CompoundSteeringBehavior();
//BalancedCompoundSteeringBehavior steer = new BalancedCompoundSteeringBehavior();
steer.addSteerBehavior(steerBehavior1);
steer.addSteerBehavior(steerBehavior2);
mainBehaviour.addBehavior(steer);
myAgent.setMainBehavior(mainBehavior);
```



Note that you can have nested containers, like is shown in the following example:

```
SimpleMainBehaviour mainBehavior = new SimpleMainBehavior(myAgent);
CompoundSteeringBehavior steer = new CompoundSteeringBehavior();
BalancedCompoundSteeringBehavior nestedSteer = new BalancedCompoundSteeringBehavior();
nestedSteer.addSteerBehavior(steerBehavior1);
steer.addSteerBehavior(nestedSteer);
steer.addSteerBehavior(steerBehavior2);
mainBehavior.addBehavior(steer);
myAgent.setMainBehavior(mainBehavior);
```



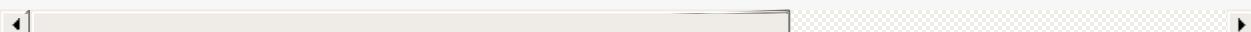
</div>

## Prioritizing behaviors

You can assign priority layers: The steering controller first checks the higher layer to see if all the behaviors returns a value higher than `minLengthToInvalidSteer`, if so it uses that layer. Otherwise, it moves on to the second layer, and so on.

To assign priority layers add behaviors with the following function:

```
addSteerBehavior (AbstractSteeringBehavior behavior, int priority)
```



To optimize the process speed add the behaviors with the lowest priority first.

The layer and the min length to consider the behavior invalid are 0 by default.

</div>

## Setting up forces

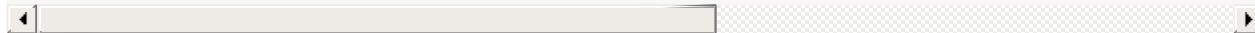
If a behavior extends from the `AbstractStrengthSteeringBehavior` class, you can manage how the produced forces will work.

Use `setupStrengthControl(float scalar)` to increase/decrease the steer force produced by a behavior or `setupStrengthControl(Plane plane)`. If you want to work with 2D behaviors.

Example:

```
Plane horizontalPlane = new Plane(new Vector3f(0,1,0), 0);

steerBehavior1.setupStrengthControl(0.5f); //Force reduced a 50%
steerBehavior2.setupStrengthControl(horizontalPlane); //Force controlled by plane
steerContainer.setupStrengthControl(horizontalPlane, 2f); //Controlled by container
```



## Implementing your own steer behavior

To benefit from all the features, you have to create a new class that extends from `AbstractStrengthSteeringBehavior`.

The responsible for the agent's acceleration is the vector returned in the `calculateRawSteering()` method:

```
@Override
protected Vector3f calculateRawSteering() {
 Vector3f steerForce = Vector3f.ZERO;

 //calculations

 return steerForce;
}
```

In addition, you can change a brake factor which will reduce the resultant velocity for the agent:

```

@Override
protected Vector3f calculateRawSteering(){
 this.setBrakingFactor(0.5f); //The agent's velocity will be
 return Vector3f.ZERO;
}

```

The braking force must be a float contained in the [0,1] interval

0 means the maximum braking force and 1 No braking force

</div>

## Strict arguments

To ensure that the behavior will work as you had planned it to work It's recommended to create your own [IllegalArgumentException](#) class. To do this, create your own container class extending from `com.jme3.ai.agents.behaviors.npc.steering.SteeringExceptions` ; Each exception inside the container class extends from `SteeringBehaviorException` . Furthermore, It will help users to recognize better which is the origin of any problem.

Example:

```

public class CustomSteeringExceptions extends SteeringException

 public static class CustomRuntimeException extends SteeringException {
 public CustomRuntimeException(String msg) { super(msg); }
 // ... other exceptions ...
 }
}

```

```

public SteerBehaviorConstructor(Agent agent, int value, Spatial spatial)
 super(agent, spatial);
 if(value > 5) throw new CustomSteeringExceptions.customRuntimeException();
 this.value = value;
}

```

## Useful links

java steer behaviors project: [jmesteer.bdevel.org](http://jmesteer.bdevel.org)

</div>

## title: JME3 Canvas in a Swing GUI

# JME3 Canvas in a Swing GUI

3D games are typically played full-screen, or in a window that takes over the mouse and all inputs. However it is also possible to embed a jME 3 canvas in a standard Swing application.

This can be useful when you create some sort of interactive 3D viewer with a user interface that is more complex than just a HUD: For instance an interactive scientific demo, a level editor, or a game character designer.

- Advantages:
  - You can use Swing components (frame, panels, menus, controls) next to your jME3 game.
  - The NetBeans GUI builder is compatible with the jMonkeyEngine; you can use it to lay out the Swing GUI frame, and then add() the jME canvas into it. Install the GUI builder via Tools → Plugins → Available Plugins.
- Disadvantages:
  - You cannot use SimpleApplication's default mouse capturing for camera navigation, but have to come up with a custom solution.

Here is the full [TestCanvas.java](#) code sample.

## Extending SimpleApplication

You start out just the same as for any jME3 game: The base application, here `SwingCanvasTest`, extends `com.jme3.app.SimpleApplication`. As usual, you use `simpleInitApp()` to initialize the scene, and `simpleUpdate()` as event loop.

The camera's default behaviour in SimpleApplication is to capture the mouse, which doesn't make sense in a Swing window. You have to deactivate and replace this behaviour by `flyCam.setDragToRotate(true);` when you initialize the application:

```

public void simpleInitApp() {
 // activate windowed input behaviour
 flyCam.setDragToRotate(true);
 // Set up inputs and load your scene as usual
 ...
}

```

In short: The first thing that is different is the `main()` method. We don't call `start()` on the `SwingCanvasTest` object as usual. Instead we create a `Runnable()` that creates and opens a standard Swing `jFrame`. In the runnable, we also create our `SwingCanvasTest` game with special settings, create a `Canvas` for it, and add that to the `jFrame`. Then we call `startCanvas()`.

## Main() and Runnable()

The Swing isn't thread-safe and doesn't allow us to keep the jME3 canvas up-to-date. This is why we create a runnable for the jME canvas and queue it in the AWT event thread, so it can be invoked "later" in the loop, when Swing is ready with updating its own stuff.

In the `SwingCanvasTest`'s `main()` method, create a queued `Runnable()`. It will contain the jME canvas and the Swing frame.

```

public static void main(String[] args) {
 java.awt.EventQueue.invokeLater(new Runnable() {
 public void run() {
 // ... see below ...
 }
 });
}

```

Note that you have to use `app.enqueue()` when modifying objects in the scene from the AWT Event Queue like you have to use `java.awt.EventQueue.invokeLater()` from other threads (e.g. the update loop) when changing swing elements. This can get hairy quickly if you don't have a proper threading model planned so you might want to use NiftyGUI as it is embedded in the update loop thread and is also cross-platform compatible (e.g. android etc.).

</div>

## Creating the Canvas

Here in the `run()` method, we start the jME application, create its canvas, create a Swing frame, and add everything together.

Specify the `com.jme3.system.AppSettings` so jME knows the size of the Swing panel that we put it into. The application will not ask the user for display settings, you have to specify them in advance.

```
AppSettings settings = new AppSettings(true);
settings.setWidth(640);
settings.setHeight(480);
```

We create our canvas application `SwingCanvasTest`, and give it the settings. We manually create a canvas for this game and configure the `com.jme3.system.JmeCanvasContext`. The method `setSystemListener()` makes sure that the listener receives events relating to context creation, update, and destroy.

```
SwingCanvasTest canvasApplication = new SwingCanvasTest();
canvasApplication.setSettings(settings);
canvasApplication.createCanvas(); // create canvas!
JmeCanvasContext ctx = (JmeCanvasContext) canvasApplication.getCont
ctx.setSystemListener(canvasApplication);
Dimension dim = new Dimension(640, 480);
ctx.getCanvas().setPreferredSize(dim);
```

Note that we have not called `start()` on the application, as we would usually do in the `main()` method. We will call `startCanvas()` later instead.

## Creating the Swing Frame

Inside the `run()` method, you create the Swing window as you would usually do. Create an empty `jFrame` and `add()` components to it, or create a custom `jFrame` object in another class file (for example, by using the NetBeans GUI builder) and create an instance of it here. Which ever you do, let's call the `jFrame` `window`.

```
JFrame window = new JFrame("Swing Application");
window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

We create a standard JPanel inside the JFrame. Give it any Layout you wish – here we use a simple Flow Layout. Where the code sample says “Some Swing Component”, this is where you add your buttons and controls.

The important step is to add() the canvas component into the panel, like all the other Swing components.

```
JPanel panel = new JPanel(new FlowLayout()); // a panel
// add all your Swing components ...
panel.add(new JButton("Some Swing Component"));
...
// add the JME canvas
panel.add(ctx.getCanvas());
```

OK, the jFrame and the panel are ready. We add the panel into the jFrame, and pack everything together. Set the window's visibility to true make it appear.

```
window.add(panel);
window.pack();
window.setVisible(true);
```

Remember that we haven't called start() on the jME application yet? For the canvas, there is a special `startCanvas()` method that you must call now:

```
canvasApplication.startCanvas();
```

Clean, build, and run!

## Navigation

Remember, to navigate in the scene, click and drag (!) the mouse, or press the WASD keys. Depending on your game you may even want to define custom inputs to handle navigation in this untypical environment.

[documentation](#), [gui](#)  
</div>

## title: Terrain Collision

### Terrain Collision

This tutorial expands the HelloTerrain tutorial and makes the terrain solid. You combine what you learned in [Hello Terrain](#) and [Hello Collision](#) and add a CollisionShape to the terrain. The terrain's CollisionShape lets the first-person player (who is also a CollisionShape) collide with the terrain, i.e. walk on it and stand on it.

### Sample Code

```
package jme3test.helloworld;

import com.jme3.app.SimpleApplication;
import com.jme3.bullet.BulletAppState;
import com.jme3.bullet.collision.shapes.CapsuleCollisionShape;
import com.jme3.bullet.collision.shapes.CollisionShape;
import com.jme3.bullet.control.CharacterControl;
import com.jme3.bullet.control.RigidBodyControl;
import com.jme3.bullet.util.CollisionShapeFactory;
import com.jme3.input.KeyInput;
import com.jme3.input.controls.ActionListener;
import com.jme3.input.controls.KeyTrigger;
import com.jme3.material.Material;
import com.jme3.math.Vector3f;
import com.jme3.renderer.Camera;
import com.jme3.scene.Node;
import com.jme3.terrain.geomipmap.TerrainLodControl;
import com.jme3.terrain.heightmap.AbstractHeightMap;
import com.jme3.terrain.geomipmap.TerrainQuad;
import com.jme3.terrain.heightmap.ImageBasedHeightMap;
import com.jme3.texture.Texture;
import com.jme3.texture.Texture.WrapMode;
import java.util.ArrayList;
import java.util.List;
```

```
import jme3tools.converters.ImageToAwt;

/**
 * This demo shows a terrain with collision detection,
 * that you can walk around in with a first-person perspective.
 * This code combines HelloCollision and HelloTerrain.
 */
public class HelloTerrainCollision extends SimpleApplication
 implements ActionListener {

 private BulletAppState bulletAppState;
 private RigidBodyControl landscape;
 private CharacterControl player;
 private Vector3f walkDirection = new Vector3f();
 private boolean left = false, right = false, up = false, down = false;
 private TerrainQuad terrain;
 private Material mat_terrain;

 public static void main(String[] args) {
 HelloTerrainCollision app = new HelloTerrainCollision();
 app.start();
 }

 @Override
 public void simpleInitApp() {
 /** Set up Physics */
 bulletAppState = new BulletAppState();
 stateManager.attach(bulletAppState);
 //bulletAppState.getPhysicsSpace().enableDebug(assetManager);

 flyCam.setMoveSpeed(100);
 setUpKeys();

 /** 1. Create terrain material and load four textures into it.
 mat_terrain = new Material(assetManager,
 "Common/MatDefs/Terrain/Terrain.j3md");

 /** 1.1) Add ALPHA map (for red-blue-green coded splat textures)
 mat_terrain.setTexture("Alpha", assetManager.loadTexture(
 "Textures/Terrain/splat/alphamap.png"));
```

```

/** 1.2) Add GRASS texture into the red layer (Tex1). */
Texture grass = assetManager.loadTexture(
 "Textures/Terrain/splat/grass.jpg");
grass.setWrap(WrapMode.Repeat);
mat_terrain.setTexture("Tex1", grass);
mat_terrain.setFloat("Tex1Scale", 64f);

/** 1.3) Add DIRT texture into the green layer (Tex2) */
Texture dirt = assetManager.loadTexture(
 "Textures/Terrain/splat/dirt.jpg");
dirt.setWrap(WrapMode.Repeat);
mat_terrain.setTexture("Tex2", dirt);
mat_terrain.setFloat("Tex2Scale", 32f);

/** 1.4) Add ROAD texture into the blue layer (Tex3) */
Texture rock = assetManager.loadTexture(
 "Textures/Terrain/splat/road.jpg");
rock.setWrap(WrapMode.Repeat);
mat_terrain.setTexture("Tex3", rock);
mat_terrain.setFloat("Tex3Scale", 128f);

/** 2. Create the height map */
AbstractHeightMap heightmap = null;
Texture heightMapImage = assetManager.loadTexture(
 "Textures/Terrain/splat/mountains512.png");
heightmap = new ImageBasedHeightMap(heightMapImage.getImage());
heightmap.load();

/** 3. We have prepared material and heightmap.
 * Now we create the actual terrain:
 * 3.1) Create a TerrainQuad and name it "my terrain".
 * 3.2) A good value for terrain tiles is 64x64 -- so we supply
 * 3.3) We prepared a heightmap of size 512x512 -- so we supply
 * 3.4) As LOD step scale we supply Vector3f(1,1,1).
 * 3.5) We supply the prepared heightmap itself.
 */
terrain = new TerrainQuad("my terrain", 65, 513, heightmap.getT

/** 4. We give the terrain its material, position & scale it, a

```

```

 terrain.setMaterial(mat_terrain);
 terrain.setLocalTranslation(0, -100, 0);
 terrain.setLocalScale(2f, 1f, 2f);
 rootNode.attachChild(terrain);

 /** 5. The LOD (level of detail) depends on where the camera is:
 List<Camera> cameras = new ArrayList<Camera>();
 cameras.add(getCamera());
 TerrainLodControl control = new TerrainLodControl(terrain, cameras);
 terrain.addControl(control);

 /** 6. Add physics:
 // We set up collision detection for the scene by creating a static
 // RigidBodyControl with mass zero.*/
 terrain.addControl(new RigidBodyControl(0));

 // We set up collision detection for the player by creating
 // a capsule collision shape and a CharacterControl.
 // The CharacterControl offers extra settings for
 // size, stepheight, jumping, falling, and gravity.
 // We also put the player in its starting position.
 CapsuleCollisionShape capsuleShape = new CapsuleCollisionShape(0.5f, 1.5f);
 player = new CharacterControl(capsuleShape, 0.05f);
 player.setJumpSpeed(20);
 player.setFallSpeed(30);
 player.setGravity(30);
 player.setPhysicsLocation(new Vector3f(-10, 10, 10));

 // We attach the scene and the player to the rootnode and the physics
 // space to make them appear in the game world.
 bulletAppState.getPhysicsSpace().add(terrain);
 bulletAppState.getPhysicsSpace().add(player);

 }

 /** We over-write some navigational key mappings here, so we can
 * add physics-controlled walking and jumping: */
 private void setUpKeys() {
 inputManager.addMapping("Left", new KeyTrigger(KeyInput.KEY_A));
 inputManager.addMapping("Right", new KeyTrigger(KeyInput.KEY_D));
 inputManager.addMapping("Up", new KeyTrigger(KeyInput.KEY_W));
 }
}

```

```

 inputManager.addMapping("Down", new KeyTrigger(KeyInput.KEY_S))
 inputManager.addMapping("Jump", new KeyTrigger(KeyInput.KEY_SPACE))
 inputManager.addListener(this, "Left");
 inputManager.addListener(this, "Right");
 inputManager.addListener(this, "Up");
 inputManager.addListener(this, "Down");
 inputManager.addListener(this, "Jump");
 }

 /**
 * These are our custom actions triggered by key presses.
 * We do not walk yet, we just keep track of the direction the user
 */
 public void onAction(String binding, boolean value, float tpf) {
 if (binding.equals("Left")) {
 if (value) { left = true; } else { left = false; }
 } else if (binding.equals("Right")) {
 if (value) { right = true; } else { right = false; }
 } else if (binding.equals("Up")) {
 if (value) { up = true; } else { up = false; }
 } else if (binding.equals("Down")) {
 if (value) { down = true; } else { down = false; }
 } else if (binding.equals("Jump")) {
 player.jump();
 }
 }

 /**
 * This is the main event loop--walking happens here.
 * We check in which direction the player is walking by interpreting
 * the camera direction forward (camDir) and to the side (camLeft)
 * The setWalkDirection() command is what lets a physics-controlling
 * We also make sure here that the camera moves with player.
 */
 @Override
 public void simpleUpdate(float tpf) {
 Vector3f camDir = cam.getDirection().clone().multLocal(0.6f);
 Vector3f camLeft = cam.getLeft().clone().multLocal(0.4f);
 walkDirection.set(0, 0, 0);
 if (left) { walkDirection.addLocal(camLeft); }
 if (right) { walkDirection.addLocal(camLeft.negate()); }
 if (up) { walkDirection.addLocal(camDir); }
 }
}

```

```

 if (down) { walkDirection.addLocal(camDir.negate()); }
 player.setWalkDirection(walkDirection);
 cam.setLocation(player.getPhysicsLocation());
 }
}

```

To try this code, create a New Project → JME3 → BasicGame using the default settings. Paste the sample code over the pregenerated Main.java class. Change the package to “mygame” if necessary. Open the Project Properties, Libraries, and add the `jme3-test-data` library to make certain you have all the files.

Compile and run the code. You should see a terrain. You can use the WASD keys and the mouse to run up and down the hills.

## Understanding the Code

### The Terrain Code

Read [Hello Terrain](#) for details of the following parts that we reuse:

1. The `AbstractHeightMap` is an efficient way to describe the shape of the terrain.
2. The `Terrain.j3md`-based Material and its texture layers let you colorize rocky mountain, grassy valleys, and a paved path criss-crossing over the landscape.
3. The TerrainQuad is the finished `terrain` Spatial that you attach to the rootNode.

### The Collision Detection Code

Read [Hello Collision](#) for details of the following parts that we reuse:

1. The `BulletAppState` lines activate physics.
2. The `ActionListener` (`onAction()`) lets you reconfigure the input handling for the first-person player, so it takes collision detection into account.
3. The custom `setUpKeys()` method loads your reconfigured input handlers. They now don't just walk blindly, but calculate the `walkDirection` vector that we need for collision detection.
4. `simpleUpdate()` uses the `walkDirection` vector and makes the character walk, while taking obstacles and solid walls/floor into account.

```
player.setWalkDirection(walkDirection);
```

5. The RigidBodyControl `landscape` is the CollisionShape of the terrain.
6. The physical first-person player is a CapsuleCollisionShape with a CharacterControl.

## Combining the Two

Here are the changed parts to combine the two:

1. You create a static (zero-mass) RigidBodyControl.
2. Add the control to the `terrain` to make it physical.

```
/** 6. Add physics: */
 terrain.addControl(new RigidBodyControl(0));
```

You attach the `terrain` and the first-person `player` to the `rootNode`, and to the physics space, to make them appear in the game world.

```
bulletAppState.getPhysicsSpace().add(terrain);
bulletAppState.getPhysicsSpace().add(player);
```

## Conclusion

You see that you can combine snippets of sample code (such as HelloTerrain and HelloCollision), and create a new application from it that combines two features into something new.

You should spawn high up in the area and fall down to the map, giving you a few seconds to survey the area. Then walk around and see how you like the lay of the land.

---

See also:

- [Hello Terrain](#),
- [Terrain](#)

[terrain](#), [collision](#)

</div>

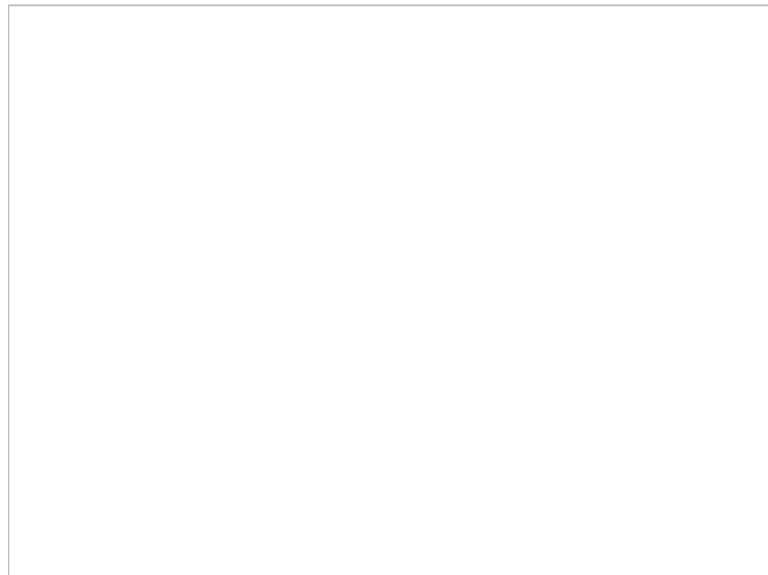
## **title: TerraMonkey - The jMonkeyEngine Terrain System**

# **TerraMonkey - The jMonkeyEngine Terrain System**

The goal of TerraMonkey is to provide a base implementation that will be usable for 80% of people's goals, while providing tools and a good foundation for the other 20% to build off of. Check out the videos in the following announcements:

- [New features](#)
- [More textures and Tools](#)

## **Overview**



TerraMonkey is a GeoMipMapping quad tree of terrain tiles that supports real time editing and texture splatting. That's a mouth full! Lets look at each part:

- **GeoMipMapping:** a method of changing the level of detail (LOD) of geometry tiles based on how far away they are from the camera. Between the edges of two tiles, it will seam those edges together so you don't get gaps or holes. For an in-depth read on how it works, here is a pdf [http://www.flipcode.com/archives/article\\_geomipmaps.pdf](http://www.flipcode.com/archives/article_geomipmaps.pdf).
- **Quad Tree:** The entire terrain structure is made up of TerrainPatches (these hold the

actual meshes) as leaves in a quad tree (TerrainQuad). TerrainQuads are subdivided by 4 until they reach minimum size, then a TerrainPatch is created, and that is where the actual geometry mesh lives. This allows for fast culling of the terrain that you can't see.

- **Splatting:** The ability to paint multiple textures onto your terrain. What differs here from JME2 is that this is all done in a shader, no more render passes. So it performs much faster.
- **Real-time editing:** TerraMonkey terrains are editable in jMonkeyEngine SDK, and you are able to modify them in real time, for example by raising and lowering the terrain.

## Current Features:

- Support for 16 splat textures. You use a custom combination of Diffuse, Normal, Specular, and Glow Maps.
- GeoMipMapping: LodControl optimizes the level of detail
- Terrain can be randomized or generated from a heightmap
- jMonkeyEngine SDK terrain editor
- Streaming [terrain grid](#) (ie. “infinite” terrain)

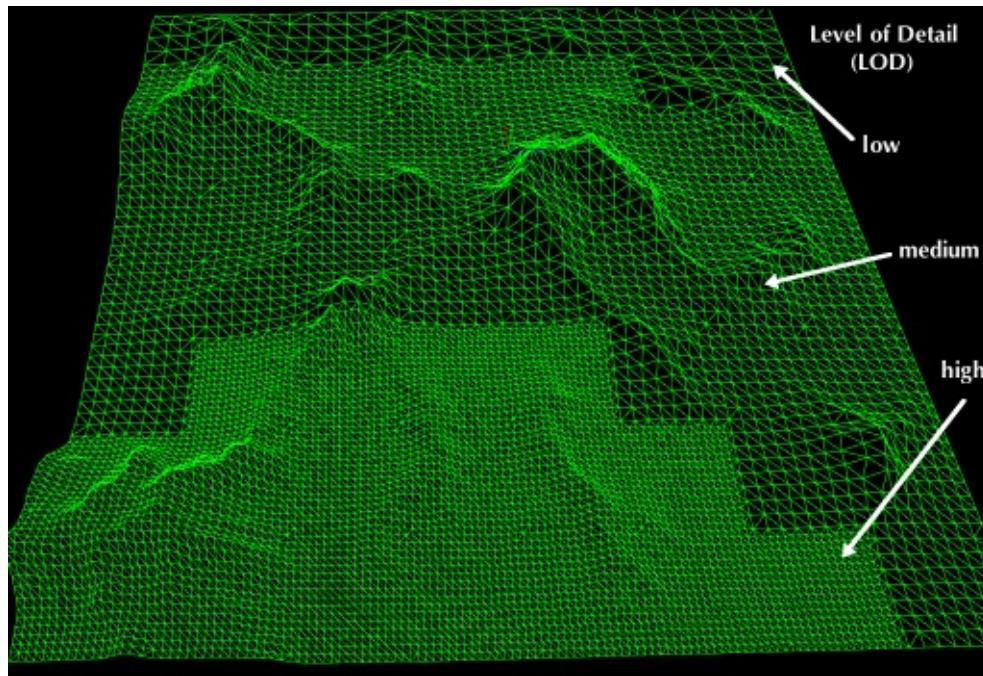
## Planned Features:

- Hydraulic erosion and procedural texture generation
- Holes: caves, cliffs

## Sample Code

- [TerrainTest.java](#)
- [TerrainTestAdvanced.java](#)
- [TerrainTestCollision.java](#)
- [TerrainTestModifyHeight.java](#)
- [TerrainTestReadWrite.java](#)

## Geo Mip Mapping



You have seen GeoMipMapping implemented in games before. This is where the farther away terrain has fewer polygons, and as you move closer, more polygons fill in. The whole terrain is divided into a grid of patches, and each one has its own level of detail (LOD). The GeoMipMapping algorithm looks at each patch, and its neighbours, to determine how to render the geometry. It will seam the edges between two patches with different LOD.

GeoMipMapping often leads to “popping” where you see the terrain switch from one LOD to another. TerraMonkey has been designed so you can swap out different LOD calculation algorithms based on what will look best for your game. You can do this with the LodCalculator interface.

GeoMipMapping in TerraMonkey has been split into several parts: the terrain quad tree, and the LODGeomap. The geomap deals with the actual LOD and seaming algorithm. So if you want a different data structure for your terrain system, you can re-use this piece of code. The quad tree (TerrainQuad and TerrainPatch) provide a means to organize the LODGeomaps, notify them of their neighbour's LOD change, and to update the geometry when the LOD does change. To change the LOD it does this by changing the index buffer of the triangle strip, so the whole geometry doesn't have to be re-loaded onto the video card. If you are eager, you can read up more detail how GeoMipMapping works here:

[www.flipcode.com/archives/article\\_geomipmaps.pdf](http://www.flipcode.com/archives/article_geomipmaps.pdf)

## Terrain Quad Tree

TerraMonkey is a quad tree. Each node is a TerrainQuad, and each leaf is a TerrainPatch. A TerrainQuad has either 4 child TerrainQuads, or 4 child TerrainPatches. The TerrainPatch holds the actual mesh geometry. This structure is almost exactly the same as JME2's

TerrainPage system. Except now each leaf has a reference to its neighbours, so it doesn't ever have to traverse the tree to get them.

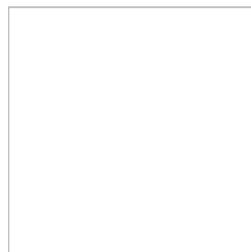
## Texture Splatting

When you “slap” a texture on a mesh, the whole mesh looks the same. For big meshes (such as terrains) that is undesirable because it looks very boring (your whole landscape would be all rock, or all grass, or all sand). Texture Splatting is a technique that lets you “paint” several textures into one combined texture. Each of the splat textures has an opacity value so you can define where it is visible in the final overall texture.

The default material for TerraMonkey is TerrainLighting.j3md. This material combines several texture maps to produce the final custom texture. Remember, Diffuse Maps are the main textures that define the look; optionally, each Diffuse Map can be enhanced with a Normal Map; Alpha Maps describe the opacity of each Diffuse Map used (one color –red, green, blue, or alpha– stands for one Diffuse Map's opacity); Glow and Specular Maps define optional effects.

We recommend to [create and edit Splat Textures for terrains visually in the jMonkeyEngine SDK](#), and not do it manually. If you are simply curious about how the SDK's terrain texture plugin works, or if you indeed want to generate materials manually, then read on for the implementation details.

Here are the names of TerrainLighting.j3md's material properties:



- 1-3 Alpha Maps
  - `AlphaMap`
  - `AlphaMap_1`
  - `AlphaMap_2`
- 12 Diffuse and/or Normal Maps (either in 6 pairs, or 12 stand-alone Diffuse Maps)
  - `DiffuseMap` , `DiffuseMap_0_scale` , `NormalMap`
  - `DiffuseMap_1` , `DiffuseMap_1_scale` , `NormalMap_1`
  - `DiffuseMap_2` , `DiffuseMap_2_scale` , `NormalMap_2`



- DiffuseMap\_3 , DiffuseMap\_3\_scale , NormalMap\_3
  - DiffuseMap\_4 , DiffuseMap\_4\_scale , NormalMap\_4
  - ...
  - DiffuseMap\_11 , DiffuseMap\_11\_scale , NormalMap\_11
- “Light” maps
    - GlowMap
    - SpecularMap

**Note:** DiffuseMap\_0\_scale is a float value (e.g. 1.0f); you must specify one scale per Diffuse Map.

OpenGL supports a maximum of 16 *samplers* in any given shader. This means you can only use a subset of material properties at the same time if you use the terrain's default lighting shader (TerrainLighting.j3md)!

Adhere to the following constraints:

- 
- 1-12 Diffuse Maps. One Diffuse Map is the minimum!
  - 1-3 Alpha Maps. For each 4 Diffuse Maps, you need 1 more Alpha Map!
  - 0-6 Normal Maps. Diffuse Maps & Normal Maps always come in pairs!
  - 0 or 1 Glow Map
  - 0 or 1 Specular Map.
  - **The sum of all textures used must be 16, or less.**

Here are some common examples what this means:

- 
- 3 Alpha + 11 Diffuse + 1 Normal.
  - 3 Alpha + 11 Diffuse + 1 Glow.
  - 3 Alpha + 11 Diffuse + 1 Specular.
  - 3 Alpha + 10 Diffuse + 3 Normal.
  - 3 Alpha + 10 Diffuse + 1 Normal + 1 Glow + 1 Specular.
  - 2 Alpha + 8 Diffuse + 6 Normal.
  - 2 Alpha + 6 Diffuse + 6 Normal + 1 Glow + 1 Specular.
  - 1 Alpha + 3 Diffuse + 3 Normal + 1 Glow + 1 Specular (rest unused)

You can hand-paint Alpha, Diffuse, Glow, and Specular maps in a drawing program, like Photoshop. Define each splat texture in the Alpha Map in either Red, Green, Blue, or Alpha (=RGBA). The JmeTests project bundled in the [SDK](#) includes some image files that show you how this works. The example images show a terrain heightmap next to its Alpha Map (which has been prepared for 3 Diffuse Maps), and one exemplary Diffuse/Normal Map pair.

</div>

## Code Sample: Terrain.j3md

This example shows the simpler material definition `Terrain.j3md`, which only supports 1 Alpha Map, 3 Diffuse Maps, 3 Normal Maps, and does not support Phong illumination. It makes the example shorter – `TerrainLighting.j3md` works accordingly (The list of material properties see above. Links to extended sample code see above.)

First, we load our textures and the heightmap texture for the terrain

```
// Create material from Terrain Material Definition
matRock = new Material(assetManager, "Common/MatDefs/Terrain/Terrai
// Load alpha map (for splat textures)
matRock.setTexture("Alpha", assetManager.loadTexture("Textures/Terr
// load heightmap image (for the terrain heightmap)
Texture heightMapImage = assetManager.loadTexture("Textures/Terrair
// load grass texture
Texture grass = assetManager.loadTexture("Textures/Terrain/splat/gr
grass.setWrap(WrapMode.Repeat);
matRock.setTexture("Tex1", grass);
matRock.setFloat("Tex1Scale", 64f);
// load dirt texture
Texture dirt = assetManager.loadTexture("Textures/Terrain/splat/dir
dirt.setWrap(WrapMode.Repeat);
matRock.setTexture("Tex2", dirt);
matRock.setFloat("Tex2Scale", 32f);
// load rock texture
Texture rock = assetManager.loadTexture("Textures/Terrain/splat/roa
rock.setWrap(WrapMode.Repeat);
matRock.setTexture("Tex3", rock);
matRock.setFloat("Tex3Scale", 128f);
```

We create the heightmap from the `heightMapImage`.

```
AbstractHeightMap heightmap = null;
heightmap = new ImageBasedHeightMap(heightMapImage.getImage(), 1f);
heightmap.load();
```

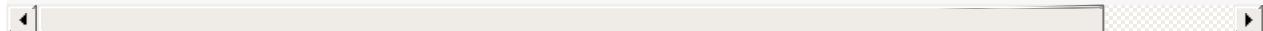


Next we create the actual terrain.

- The terrain tiles are 65×65.
- The total size of the terrain is 513×513, but it can easily be up to 1025×1025.
- It uses the heightmap to generate the height values.

</ul>

```
terrain = new TerrainQuad("terrain", 65, 513, heightmap.getHeightMap());
terrain.setMaterial(matRock);
terrain.setLocalScale(2f, 1f, 2f); // scale to make it less steep
List<Camera> cameras = new ArrayList<Camera>();
cameras.add(getCamera());
TerrainLodControl control = new TerrainLodControl(terrain, cameras);
terrain.addControl(control);
rootNode.attachChild(terrain);
```



PS: As an alternative to an image-based height map, you can also generate a Hill heightmap:

```
heightmap = new HillHeightMap(1025, 1000, 50, 100, (byte) 3);
```

</div>

## title: Optimization: Texture Atlas

# Optimization: Texture Atlas

The `jme3tools.optimize.TextureAtlas` allows combining multiple textures into one texture atlas. Loading one geometry with one material is much more efficient than handling several geometries and materials. Optimally, you already export your textures as texture atlas from e.g. Blender.

`jme3tools.optimize.GeometryBatchFactory`, in contrast, only works if the geometries have only one material with textures.

1. Create a `TextureAtlas`.
2. Add textures to texture atlas, each texture goes onto one map (e.g. `DiffuseMap`).  
The image data is stored in a byte array for each named texture map.
3. Later, you retrieve each map by name as a `Texture`, and use it in materials.

## Sample Code

- [TestTextureAtlas.java](#)

## API

TextureAtlas method	Usage
addGeometry(g)	Add this geometry's DiffuseMap (or ColorMap), NormalMap, and SpecularMap to the atlas (if they exist). The DiffuseMap will automatically be the master map.
addTexture(t, mapname)	Add a texture to the named master map.
addTexture(t1, mapName, t2)	Add a texture t1 to the named secondary map, and make the location of texture t1 correspond to texture t2 on the master map. t2 can be Texture object or the String name of the texture.
applyCoords(g)	Applies the texture coordinates to the geometries mesh. Short for the default, applyCoords(geom, 0, geom.getMesh()) .
applyCoords(g, offset, mesh)	Applies the texture coordinates at the given texture coord buffer offset to the mesh, if the DiffuseMap or ColorMap of the input geometry g exist in the atlas. The mesh can be g.getMesh() . Target buffer offset is between 0 and buffer.size().
getAtlasTile(texture)	Get the TextureAtlasTile for the given Texture. The TextureAtlasTile objects contains info about this texture. such as size and location in the atlas.
getAtlasTexture(mapName)	Creates a new atlas texture from the added textures for the given map name.

TextureAtlasTile method	Usage
getHeight(), getWidth()	Gets the size of the texture.
getX(), getY()	Gets the x and y coordinate inside the texture atlas where this texture can be found
transformTextureCoords(inBuf, offset, outBuf)	Transforms the texture coordinates in a buffer from their original 0-1 values to the new values fitting to the location of the tile on the atlas texture.
getLocation(l)	Get the transformed texture coordinate for a given input location.

## Primary and Secondary Maps

The helper methods that work with Geometry objects automatically consider the `DiffuseMap` (for Lighting.j3md-based Materials) or `ColorMap` (for Unshaded.j3md-based Materials) the **master map**, and additionally treat the `NormalMap` and `SpecularMap` as secondary maps, if they exist in the Geometry.

- The first map name that you supply becomes the **master map**. The master map defines locations on the atlas. Typically, you name the master map `DiffuseMap` (for `Lighting.j3md`) or `ColorMap` (for `Unshaded.j3md`).  
In general, if you want to use the texture with a certain shader, you should supply the map names that are specified in this shader's `.j3md` for clarity.
- Secondary textures (other map names, for example `SpecularMap` and `NormalMap` for `Lighting.j3md`) have to reference an existing texture on the master map, so the Atlas knows where to position the added texture on the secondary map. This is necessary because the maps share texture coordinates and thus need to be placed at the same location on both maps. If you do not share texture coordinates in your maps, use separate `TextureAtlas` classes.

You reference textures by their **asset key name**, this is what their “id” is. For each texture, the atlas stores its location. A texture with an existing key name is never added more than once to the atlas. You can access the information for each texture or geometry texture via helper methods.

- Textures are not scaled automatically, and your atlas needs to be created large enough to hold all textures. All methods that allow adding textures return false if the texture could not be added due to the atlas being full.
- Secondary textures (normal maps, specular maps etc.) have to be the same size as the main (e.g. `DiffuseMap`) texture.
- The `TextureAtlas` lets you change the texture coordinates of a mesh or geometry to point at the new locations of its texture inside the atlas (if the texture exists inside the atlas). ??

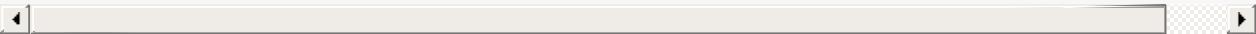
Note that models that use texture coordinates outside the 0-1 range (that is, they use repeating/wrapping textures) do not work correctly as their new coordinates leak into other parts of the atlas and thus display other textures instead of repeating the texture!

</div>

## Usage examples: Combine Geometries

Use `makeAtlasBatch` to turn several geometries that are loaded from a `j3o` file into one geometry:

```
// scene contains many geometries ared attached to one node:
Node scene = assetManager.loadModel("Scenes/MyScene.j3o");
// geom is one geometry containing all of these geometries together
Geometry geom = TextureAtlas.makeAtlasBatch(scene);
rootNode.attachChild(geom);
```



Create a texture atlas and change the texture coordinates of one geometry:

```
Node scene = assetManager.loadModel("Scenes/MyScene.j3o");

// Either auto-create texture Atlas from an existing node that has
TextureAtlas atlas = TextureAtlas.createAtlas(scene);

//... or create Atlas manually by adding textures (e.g. myDiffuse)
TextureAtlas atlas = new TextureAtlas(1024,1024);
atlas.addTexture(myDiffuseTexture, "DiffuseMap");

// ... or create Atlas manually by adding textured geometries (e.g.
TextureAtlas atlas = new TextureAtlas(1024,1024);
atlas.addGeometry(MyGeometry);

// Create material, load texture from Atlas, apply texture to mate
Material mat = new Material(mgr, "Common/MatDefs/Light/Lighting.j3m");
mat.setTexture("DiffuseMap", atlas.getAtlasTexture("DiffuseMap"));

// Get one sub-geometry on which you want to use the Atlas, apply
// of this geometry to the atlas, and replace the material.
Geometry geom = scene.getChild("MyGeometry");
atlas.applyCoords(geom);
geom.setMaterial(mat);
```



</div>

## **title: Traverse the SceneGraph**

# **Traverse the SceneGraph**

You can run a search across the whole scene graph and search for individual Spatial (`Nodes` and `Geometry`s) by custom criteria, such as the Spatial's name, or the Spatial's class, or the Spatial's user data, or Spatial's Controls. You do this when you want modify the found nodes (move them, call a method, etc) but you don't have a local variable for them.

## **Example Use Cases**

### **Example 1:**

1. You have created a procedural scene with lots of dynamically generated elements.
2. You want to find individual Spatial under ever-changing conditions and modify their state.

### **Example 2:**

1. You created a mostly static scene in the jMonkeyEngine SDK and exported it as .j3o file.  
The scene also contains interactive objects, for example a particle emitter, spatial with user data, or spatial with custom controls.
2. You load the .j3o scene using the assetManager.
3. You want to interact with one of the loaded interactive scene elements in your Java code.  
For example, you want to call `emitAllParticles()` on the particle emitter. Or you want to find all NPC's Geometries with a custom CivilianControl, and call the CivilianControl method that makes them start acting their role.

In this case, you can use a `SceneGraphVisitorAdapter` to identify and access the Spatial in question.

## **Code Sample**

For each search, you create a `com.jme3.scene.SceneGraphVisitorAdapter` that defines your search criteria and what you want to do with the found Spatial. Then you call the `depthFirstTraversal(visitor)` or `breadthFirstTraversal(visitor)` method of the Spatial (e.g. the `rootNode`, or better, a subnode) to start the search.

```
SceneGraphVisitor visitor = new SceneGraphVisitor() {

 @Override
 public void visit(Spatial spat) {
 // search criterion can be control class:
 MyControl control = spatial.getControl(MyControl.class);
 if (control != null) {
 // you have access to any method, e.g. name.
 System.out.println("Instance of " + control.getClass().getName()
 + " found for " + spatial.getName());
 }
 }

};

// Now scan the tree either depth first...
rootNode.depthFirstTraversal(visitor);
// ... or scan it breadth first.
rootNode.breadthFirstTraversal(visitor);
```

Which of the two methods is faster depends on how you designed the scenegraph, and what tree element you are looking for. If you are searching for one single Geometry that is a “leaf” of the tree, and then stop searching, depth-first may be faster. If you search for a high-level grouping Node, breadth-first may be faster.

The choice of depth- vs breadth-first also influences the order in which found elements are returned (children first or parents first). If you want to modify user data that is inherited from the parent node (e.g. transformations), the order of application is important, because the side-effects add up.

You can use the `SceneGraphVisitorAdapter` class to scan separately for Geometry and Nodes.

## See Also

- The Scene Graph
- Spatial

[spatial](#), [node](#), [mesh](#), [geometry](#), [scenegraph](#)

</div>

## title: Main Update Loop

# Main Update Loop

Extending your application from `com.jme3.app.SimpleApplication` provides you with an update loop. This is where you implement your game logic (game mechanics).

Some usage examples: Here you remote-control NPCs (computer controlled characters), generate game events, and respond to user input.

To let you see the main update loop in context, understand that the `SimpleApplication` does the following:

- **Initialization** – Execute `simpleInitApp()` method once.
- **Main Update Loop**
  1. Input listeners respond to mouse clicks and keyboard presses – [Input handling](#)
  2. Update game state:
    - i. Update overall game state – Execute [Application States](#)
    - ii. User code update – Execute `simpleUpdate()` method
    - iii. Logical update of entities – Execute [Custom Controls](#)
  3. Render audio and video
    - i. [Application States](#) rendering.
    - ii. Scene rendering.
    - iii. User code rendering – Execute `simpleRender()` method.
  4. Repeat loop.
- **Quit** – If user requests `exit()`, execute `cleanup()` and `destroy()`.

The jME window closes and the loop ends.

## Usage

In a trivial `SimpleApplication` (such as a [Hello World tutorial](#)), all code is either in the `simpleInitApp()` (initialization) or `simpleUpdate()` (behaviour) method – or in a helper method/class that is called from one of these two. This trivial approach will make your main class very long, hard to read, and hard to maintain. You don't need to load the whole scene at once, and you don't need to run all conditionals tests all the time.

It's a best practice to modularize your game mechanics and spread out initialization and update loop code over several Java objects:

- Move modular code blocks from the `simpleInitApp()` method into [AppStates](#). Attach AppStates to initialize custom subsets of “one dungeon”, and detach it when the player exits this “dungeon”.

Examples: Weather/sky audio and visuals, physics collision shapes, sub-rootnodes of individual dungeons including dungeon NPCs, etc.

- Move modular code blocks from the `simpleUpdate()` method into the update loops of [Custom Controls](#) to control individual entity behavior (NPCs), and into the update method of [AppStates](#) to control world events.

Examples: Weather behaviour, light behaviour, physics behaviour, individual NPC behaviour, trap behaviour, etc.

[basegame](#), [control](#), [input](#), [init](#), [keyinput](#), [loop](#), [states](#), [state](#)

</div>

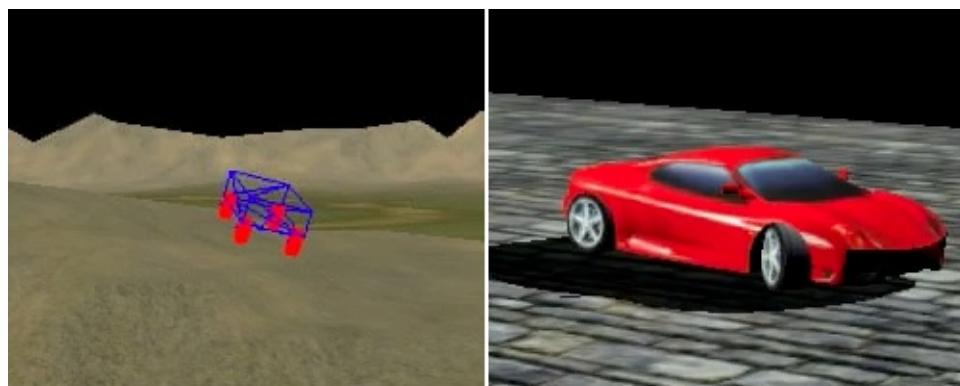
## title: Controlling a Physical Vehicle

# Controlling a Physical Vehicle

For physical vehicles, jME's uses the jBullet ray-cast vehicle. In this vehicle implementation, the physical chassis 'floats' along on four non-physical vertical rays.

Internally, each wheel casts a ray down, and using the ray's intersection point, jBullet calculates the suspension length, and the suspension force. The suspension force is applied to the chassis, keeping it from hitting the ground. The friction force is calculated for each wheel where the ray intersects with the ground. Friction is applied as a sideways and forwards force. [1\)](#)

This article shows how you use this vehicle implementation in a jME3 application.



## Sample Code

Full code samples are here:

- [TestPhysicsCar.java](#)
- [TestFancyCar.java](#)

## Overview of this Physics Application

The goal is to create a physical vehicle with wheels that can be steered and that interacts (collides with) with the floor and obstacles.

1. Create a SimpleApplication with a [BulletAppState](#)

- This gives us a PhysicsSpace for PhysicsNodes
2. Create a VehicleControl + CompoundCollisionShape for the physical vehicle behaviour
    - i. Set physical properties of the vehicle, such as suspension.
  3. Create a VehicleNode for the car model
    - i. Create a box plus 4 cylinders as wheels (using `vehicle.addWheel()` ).
    - ii. Add the VehicleControl behaviour to the VehicleNode geometry.
  4. Create a RigidBodyControl and CollisionShape for the floor
  5. Map key triggers and add input listeners
    - Navigational commands Left, Right, Forward, Brake.
  6. Define the steering actions to be triggered by the key events.
    - `vehicle.steer()`
    - `vehicle.accelerate()`
    - `vehicle.brake()`

## Creating the Vehicle Chassis

The vehicle that we create here in the [TestPhysicsCar.java](#) example is just a “box on wheels”, a basic vehicle shape that you can replace with a fancy car model, as demonstrated in [TestFancyCar.java](#).

Every physical object must have a collision shape, that we prepare first. For the vehicle, we choose a compound collision shape that is made up of a box-shaped body of the right size for the vehicle. We will add the wheels later.

```
CompoundCollisionShape compoundShape = new CompoundCollisionShape()
BoxCollisionShape box = new BoxCollisionShape(new Vector3f(1.2f, 0.
```



**Best Practice:** We attach the BoxCollisionShape (the vehicle body) to the CompoundCollisionShape at a Vector of (0,1,0): This shifts the effective center of mass of the BoxCollisionShape downwards to 0,-1,0 and makes a moving vehicle more stable!

```
compoundShape.addChildShape(box, new Vector3f(0, 1, 0));
```

Any kind of geometry can make up the visible part of the vehicle, here we use a wireframe box. We create a node that we use to group the geometry.

```
Node vehicleNode=new Node("vehicleNode");
vehicle = new VehicleControl(compoundShape, 400);
vehicleNode.addControl(vehicle);
```

We initialize the Vehicle Control with the compound shape, and set its mass to a heavy value, 400f. The Vehicle Control represents the car's physical behaviour.

```
vehicle = new VehicleControl(compoundShape, 400);
```

Finally we add the behaviour (VehicleControl) to the visible Geometry (node).

```
vehicleNode.addControl(vehicle);
```

We configure the physical properties of the vehicle's suspension: Compresion, Damping, Stiffness, and MaxSuspensionForce. Picking workable values for the wheel suspension can be tricky – for background info have a look at these [Suspension Settings Tips](#). For now, let's work with the following values:

```
float stiffness = 60.0f;//200=f1 car
float compValue = .3f; // (should be lower than damp)
float dampValue = .4f;
vehicle.setSuspensionCompression(compValue * 2.0f * FastMath.sqrt(stiffness));
vehicle.setSuspensionDamping(dampValue * 2.0f * FastMath.sqrt(stiffness));
vehicle.setSuspensionStiffness(stiffness);
vehicle.setMaxSuspensionForce(10000.0f);
```

We now have a node `vehicleNode` with a visible “car” geometry, which acts like a vehicle. One thing that's missing are wheels.

## Adding the Wheels

We create four wheel Geometries and add them to the vehicle. Our wheel geometries are simple, non-physical discs (flat Cylinders), they are just visual decorations. Note that the physical wheel behaviour (the com.jme3.bullet.objects.VehicleWheel objects) is created internally by the `vehicle.addwheel()` method.

The `addwheel()` method sets following properties:

- Vector3f connectionPoint – Coordinate where the suspension connects to the chassis (internally, this is where the Ray is casted downwards).
- Vector3f direction – Wheel direction is typically a (0,-1,0) vector.
- Vector3f axle – Axle direction is typically a (-1,0,0) vector.
- float suspensionRestLength – Suspension rest length in world units
- float wheelRadius – Wheel radius in world units
- boolean isFrontWheel – Whether this wheel is one of the steering wheels.  
Front wheels are the ones that rotate visibly when the vehicle turns.

We initialize a few variables that we will reuse when we add the four wheels. yOff, etc, are the particular wheel offsets for our small vehicle model.

```
Vector3f wheelDirection = new Vector3f(0, -1, 0);
Vector3f wheelAxe = new Vector3f(-1, 0, 0);
float radius = 0.5f;
float restLength = 0.3f;
float yOff = 0.5f;
float xOff = 1f;
float zOff = 2f;
```

We create a Cylinder mesh shape that we use to create the four visible wheel geometries.

```
Cylinder wheelMesh = new Cylinder(16, 16, radius, radius * 0.6f, tr
```

For each wheel, we create a Node and a Geometry. We attach the Cylinder Geometry to the Node. We rotate the wheel by 90° around the Y axis. We set a material to make it visible. Finally we add the wheel (plus its properties) to the vehicle.

```
Node node1 = new Node("wheel 1 node");
Geometry wheels1 = new Geometry("wheel 1", wheelMesh);
node1.attachChild(wheels1);
wheels1.rotate(0, FastMath.HALF_PI, 0);
wheels1.setMaterial(mat);

vehicle.addWheel(node1, new Vector3f(-xOff, yOff, zOff),
 wheelDirection, wheelAxe, restLength, radius, true);
```

The three next wheels are created in the same fashion, only the offsets are different. Remember to set the Boolean parameter correctly to indicate whether it's a front wheel.

```

...
vehicle.addWheel(node2, new Vector3f(xOff, yOff, zOff),
 wheelDirection, wheelAxle, restLength, radius, true);
...
vehicle.addWheel(node3, new Vector3f(-xOff, yOff, -zOff),
 wheelDirection, wheelAxle, restLength, radius, false);
...
vehicle.addWheel(node4, new Vector3f(xOff, yOff, -zOff),
 wheelDirection, wheelAxle, restLength, radius, false);

```

Attach the wheel Nodes to the vehicle Node to group them, so they move together.

```

vehicleNode.attachChild(node1);
vehicleNode.attachChild(node2);
vehicleNode.attachChild(node3);
vehicleNode.attachChild(node4);

```

As always, attach the vehicle Node to the rootNode to make it visible, and add the Vehicle Control to the PhysicsSpace to make the car physical.

```

rootNode.attachChild(vehicleNode);
getPhysicsSpace().add(vehicle);

```

Not shown here is that we also created a Material `mat`.

## Steering the Vehicle

Not shown here is the standard way how we map the input keys to actions (see full code sample). Also refer to [Input Handling](#).

In the ActionListener, we implement the actions that control the vehicle's direction and speed. For the four directions (accelerate=up, brake=down, left, right), we specify how we want the vehicle to move.

- The braking action is pretty straightforward:  
`vehicle.brake(brakeForce)`
- For left and right turns, we add a constant to `steeringValue` when the key is pressed, and subtract it when the key is released.  
`vehicle.steer(steeringValue);`

- For acceleration we add a constant to `accelerationValue` when the key is pressed, and subtract it when the key is released.  
`vehicle.accelerate(accelerationValue);`
- Because we can and it's fun, we also add a turbo booster that makes the vehicle jump when you press the assigned key (spacebar).  
`vehicle.applyImpulse(jumpForce, Vector3f.ZERO);`

```
public void onAction(String binding, boolean value, float tpf) {
 if (binding.equals("Lefts")) {
 if (value) { steeringValue += .5f; } else { steeringValue += - .5f; }
 vehicle.steer(steeringValue);
 } else if (binding.equals("Rights")) {
 if (value) { steeringValue += -.5f; } else { steeringValue += .5f; }
 vehicle.steer(steeringValue);
 } else if (binding.equals("Ups")) {
 if (value) {
 accelerationValue += accelerationForce;
 } else {
 accelerationValue -= accelerationForce;
 }
 vehicle.accelerate(accelerationValue);
 } else if (binding.equals("Downs")) {
 if (value) { vehicle.brake(brakeForce); } else { vehicle.brake(-brakeForce); }
 } else if (binding.equals("Space")) {
 if (value) {
 vehicle.applyImpulse(jumpForce, Vector3f.ZERO);
 }
 } else if (binding.equals("Reset")) {
 if (value) {
 System.out.println("Reset");
 vehicle.setPhysicsLocation(Vector3f.ZERO);
 vehicle.setPhysicsRotation(new Matrix3f());
 vehicle.setLinearVelocity(Vector3f.ZERO);
 vehicle.setAngularVelocity(Vector3f.ZERO);
 vehicle.resetSuspension();
 } else {
 }
 }
}
```

For your reference, this is how we initialized the constants for this example:

```
private final float accelerationForce = 1000.0f;
private final float brakeForce = 100.0f;
private float steeringValue = 0;
private float accelerationValue = 0;
private Vector3f jumpForce = new Vector3f(0, 3000, 0);
```

Remember, the standard input listener code that maps the actions to keys can be found in the code samples.

## Detecting Collisions

Read the [Responding to a PhysicsCollisionEvent](#) chapter in the general physics documentation on how to detect collisions. You would do this if you want to react to collisions with custom events, such as adding points or subtracting health.

## Best Practices

This example shows a very simple but functional vehicle. For a game you would implement steering behaviour and acceleration with values that are typical for the type of vehicle that you want to simulate. Instead of a box, you load a chassis model. You can consider using an [AnalogListener](#) to respond to key events in a more sophisticated way.

For a more advanced example, look at [TestFancyCar.java](#).

[documentation](#), [physics](#), [vehicle](#), [collision](#)

</div>

1) <https://docs.google.com/Doc?docid=0AXVUZ5xw6XpKZGNuZG56a3FfMzU0Z2NyZnF4Zmo&hl=en>

## title: Video

# Video

Relevant forum topics:

<http://hub.jmonkeyengine.org/t/ive-made-a-movieappstate-so-you-dont-have-to/31673>

<http://hub.jmonkeyengine.org/t/news-about-playing-videos-on-jme/30084/5>

[YUNPM - a Java Media Player](#) ([java-gaming.org](http://java-gaming.org))

[GstLWJGL](#) ([java-gaming.org](http://java-gaming.org))

[GL Media Player](#) ([jogamp.org](http://jogamp.org))

[Xuggler](#)

This API is deprecated.

jMonkeyEngine supports Jheora Ogg video ( `com.jme3.video` ).

Full code sample here: [TestVideoPlayer.java](#)

You create either a file inputstream to load the video from your hard drive:

```
FileInputStream fis = new FileInputStream("E:\\my_bunny.ogg");
```

Or you stream the video live from a web location:

```
InputStream fis = new URL("http://server/my_video.ogg").openStream()
```

Setting the queued frames to a value higher than 5 ( `videoQueue = new VQueue(5);` ) will make web streamed playback smoother at the cost of memory. Here is an example of video streaming in context:

```

private void createVideo(){
 try {
 InputStream fis = new URL("http://mirrorblender.top-ix.
 "bigbuckbunny_movies/big_buck_bunny_480p_stereo.ogg").c
 videoQueue = new VQueue(5); // streaming
 decoder = new AVThread(fis, videoQueue);
 videoThread = new Thread(decoder, "Jheora Video Decoder");
 videoThread.setDaemon(true);
 videoThread.start();
 masterClock = decoder.getMasterClock();
 } catch (IOException ex) {
 ex.printStackTrace();
 }
}

```

Use the `simpleUpdate()` method to play the audio:

```

@Override
public void simpleUpdate(float tpf){
 if (source == null){
 if (decoder.getAudioStream() != null){
 source = new AudioNode(decoder.getAudioStream(), n
 source.setPositional(false);
 source.setReverbEnabled(false);
 audioRenderer.playSource(source);
 }else{
 // uncomment this statement to be able to play vide
 // without audio.
 return;
 }
 }

 if (waitTime > 0){
 waitTime -= tpf;
 if (waitTime > 0)
 return;
 else{
 waitTime = 0;
 drawFrame(frameToDraw);
 }
 }
}

```

```
 frameToDraw = null;
 }
} else{
 VFrame frame;
 try {
 frame = videoQueue.take();
 } catch (InterruptedException ex){
 stop();
 return;
 }
 if (frame.getTime() < lastFrameTime){
 videoQueue.returnFrame(frame);
 return;
 }

 if (frame.getTime() == -2){
 // end of stream
 System.out.println("End of stream");
 stop();
 return;
 }

 long AV_SYNC_THRESHOLD = 1 * Clock.MILLISECONDS_TO_NANOS;

 long delay = frame.getTime() - lastFrameTime;
 long diff = frame.getTime() - masterClock.getTime();
 long syncThresh = delay > AV_SYNC_THRESHOLD ? delay : A

 // if difference is more than 1 second, synchronize.
 if (Math.abs(diff) < Clock.SECONDS_TO_NANOS){
 if(diff <= -syncThresh) {
 delay = 0;
 } else if(diff >= syncThresh) {
 delay = 2 * delay;
 }
 }

 System.out.println("M: "+decoder.getSystemClock().getTime()
 ", V: "+decoder.getVideoClock().getTime()
 ", A: "+decoder.getAudioClock().getTime())
}
```

```
 if (delay > 0){
 waitNanos(delay);
 drawFrame(frame);
 }else{
 videoQueue.returnFrame(frame);
 lastFrameTime = frame.getTime();
 }
 }
}
```

#### Helper Methods:

```
private void drawFrame(VFrame frame){
 Image image = frame.getImage();
 frame.setImage(image);
 picture.setTexture(assetManager, frame, false);

 // note: this forces renderer to upload frame immediately,
 // since it is going to be returned to the video queue pool
 // it could be used again.
 renderer.setTexture(0, frame);
 videoQueue.returnFrame(frame);
 lastFrameTime = frame.getTime();
}
```

```
private void waitNanos(long time){
 long millis = (long) (time / Clock.MILLIS_TO_NANOS);
 int nanos = (int) (time - (millis * Clock.MILLIS_TO_NANOS));

 try {
 Thread.sleep(millis, nanos);
 }catch (InterruptedException ex){
 stop();
 return;
 }
}
```

&lt;/div&gt;

## title: Walking Character

# Walking Character

In the [Hello Collision](#) tutorial and the [TestQ3.java](#) code sample you have seen how to create collidable landscapes and walk around in a first-person perspective. The first-person camera is enclosed by a collision shape and is steered by the BetterCharacterControl.

Other games however require a third-person perspective of the character: In these cases you use a CharacterControl on a Spatial. This example also shows how to set up custom navigation controls, so you can press WASD to make the third-person character walk; and how to implement dragging the mouse to rotate.

Some details on this page still need to be updated from old CharacterControl [API](#) to BetterCharacterControl [API](#).

</div>

## Sample Code

Several related code samples can be found here:

- [TestPhysicsCharacter.java](#) (third-person view)
- [TestWalkingChar.java](#) (third-person view)
  - Uses also [BombControl.java](#)
- [TestQ3.java](#) (first-person view)
- [HelloCollision.java](#) (first-person view)

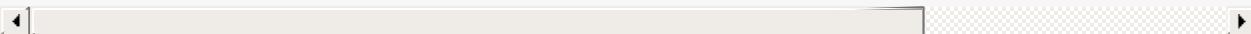
The code in this tutorial is a combination of these samples.

## BetterCharacterControl

Motivation: When you load a character model, give it a RigidBodyControl, and use forces to push it around, you do not get the desired behaviour: RigidBodyControl'ed objects (such as cubes and spheres) roll or tip over when pushed by physical forces. This is not the behaviour that you expect of a walking character. JME3's BulletPhysics integration offers a BetterCharacterControl with a `setWalkDirection()` method. You use it to create simple characters that treat floors and walls as solid, and always stays locked upright while moving.

This code sample creates a simple upright Character:

```
// Load any model
Node myCharacter = (Node) assetManager.loadModel("Models/myCharacter");
rootNode.attachChild(myCharacter);
// Create a appropriate physical shape for it
CapsuleCollisionShape capsuleShape = new CapsuleCollisionShape(1.5f);
CharacterControl myCharacter_phys = new CharacterControl(capsuleShape);
// Attach physical properties to model and PhysicsSpace
myCharacter.addControl(myCharacter_phys);
bulletAppState.getPhysicsSpace().add(myCharacter_phys);
```



To use the BetterCharacterControl, you may load your character like this:

```
// Load any model
Spatial playerSpatial = assetManager.loadModel("Models/Oto/Oto.mesh");
player = (Node)playerSpatial; // You can set the model directly to
Node playerNode = new Node(); // You can create a new node to wrap
playerNode.attachChild(player); // add it to the wrapper
player.move(0,3.5f,0); // adjust position to ensure collisions occur
player.setLocalScale(0.5f); // optionally adjust scale of model
// setup animation:
control = player.getControl(AnimControl.class);
control.addListener(this);
channel = control.createChannel();
channel.setAnim("stand");
playerControl = new BetterCharacterControl(1.5f, 6f, 1f); // construc
playerNode.addControl(playerControl); // attach to wrapper
// set basic physical properties:
playerControl.setJumpForce(new Vector3f(0,5f,0));
playerControl.setGravity(new Vector3f(0,1f,0));
playerControl.warp(new Vector3f(0,10,10)); // warp character into l
// add to physics state
bulletAppState.getPhysicsSpace().add(playerControl);
bulletAppState.getPhysicsSpace().addAll(playerNode);
rootNode.attachChild(playerNode); // add wrapper to root
```



The BulletPhysics CharacterControl only collides with “real” PhysicsControls (RigidBody). It

does not detect collisions with other CharacterControls! If you need additional collision checks, add GhostControls to your characters and create a custom [collision listener](#) to respond. (The JME3 team may implement a better CharacterControl one day.)

A CharacterControl is a special kinematic object with restricted movement.

CharacterControls have a fixed “upward” axis, this means they do not topple over when walking over an obstacle (as opposed to RigidBodyControls), which simulates a being's ability to balance upright. A CharacterControl can jump and fall along its upward axis, and it can scale steps of a certain height/steeepness.

CharacterControl Method	Property
setUpAxis(1)	Fixed upward axis. Values: 0 = X axis , 1 = Y axis , 2 = Z axis. Default: 1, because for characters and vehicles, up is typically along the Y axis.
setJumpSpeed(10f)	Jump speed (movement along upward-axis)
setFallSpeed(20f)	Fall speed (movement opposite to upward-axis)
setMaxSlope(1.5f)	How steep the slopes and steps are that the character can climb without considering them an obstacle. Higher obstacles need to be jumped. Vertical height in world units.
setGravity(1f)	The intensity of gravity for this CharacterControl'ed entity. Tip: To change the direction of gravity for a character, modify the up axis.
setWalkDirection(new Vector3f(0f,0f,0.1f))	(CharacterControl only) Make a physical character walk continuously while checking for floors and walls as solid obstacles. This should probably be called “setPositionIncrementPerSimulatorStep”. This argument is neither a direction nor a velocity, but the amount to increment the position each physics tick: vector length = accuracy*speed in m/s. Use <code>setWalkDirection(Vector3f.ZERO)</code> to stop a directional motion.

For best practices on how to use `setWalkDirection()`, see the [Navigation Inputs](#) example below.

</div>

## Walking Character Demo

### Code Skeleton

```
public class WalkingCharacterDemo extends SimpleApplication
 implements ActionListener, AnimEventListener {

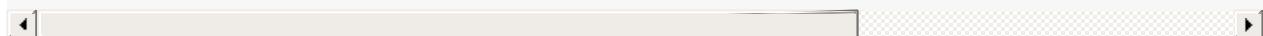
 public static void main(String[] args) {
 WalkingCharacterDemo app = new WalkingCharacterDemo();
 app.start();
 }

 public void simpleInitApp() { }

 public void simpleUpdate(float tpf) { }

 public void onAction(String name, boolean isPressed, float tpf) { }

 public void onAnimCycleDone(AnimControl control, AnimChannel char
 public void onAnimChange(AnimControl control, AnimChannel channel
```



## Overview

To create a walking character:

1. (Unless you already have it) Activate physics in the scene by adding a [BulletAppState](#).
2. Init the scene by loading the game level model (terrain or floor/buildings), and giving the scene a [MeshCollisionShape](#).
3. Create the animated character:
  - i. Load an animated character model.
  - ii. Add a [CharacterControl](#) to the model.
4. Set up animation channel and controllers.
5. Add a [ChaseCam](#) or [CameraNode](#).
6. Handle navigational inputs.

## Activate Physics

```
private BulletAppState bulletAppState;
...
public void simpleInitApp() {
 bulletAppState = new BulletAppState();
 //bulletAppState.setThreadingType(BulletAppState.ThreadingType.
 stateManager.attach(bulletAppState);
 ...
}
[<] [>] [x]
```

## Initialize the Scene

In the `simpleInitApp()` method you initialize the scene and give it a `MeshCollisionShape`. The sample in the jme3 sources uses a custom helper class that simply creates a flat floor and drops some cubes and spheres on it:

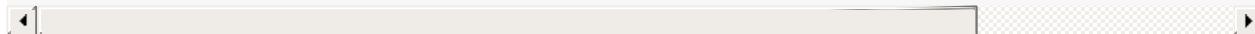
```
public void simpleInitApp() {
 ...
 PhysicsTestHelper.createPhysicsTestWorld(rootNode,
 assetManager, bulletAppState.getPhysicsSpace());
 ...
```

In a real game, you would load a scene model here instead of a test world. You can load a model from a local or remote zip file, and scale and position it:

```

private Node gameLevel;
...
public void simpleInitApp() {
 ...
//assetManager.registerLocator("quake3level.zip", ZipLocator.class);
assetManager.registerLocator(
"http://jmonkeyengine.googlecode.com/files/quake3level.zip",
HttpZipLocator.class);
MaterialList matList = (MaterialList) assetManager.loadAsset("Scen
OgreMeshKey key = new OgreMeshKey("main.meshxml", matList);
gameLevel = (Node) assetManager.loadAsset(key);
gameLevel.setLocalTranslation(-20, -16, 20);
gameLevel.setLocalScale(0.10f);
gameLevel.addControl(new RigidBodyControl(0));
rootNode.attachChild(gameLevel);
bulletAppState.getPhysicsSpace().addAll(gameLevel);
...

```



Also, add a light source to be able to see the scene.

```

AmbientLight light = new AmbientLight();
light.setColor(ColorRGBA.White.mult(2));
rootNode.addLight(light);

```

## Create the Animated Character

You create an animated model, such as Oto.mesh.xml.

1. Place the “Oto” model into the `assets/Models/Oto/` directory of your project.
2. Create the CollisionShape and adjust the capsule radius and height to fit your character model.
3. Create the CharacterControl and adjust the stepheight (here 0.05f) to the height that the character can climb up without jumping.
4. Load the visible model. Make sure its start position does not overlap with scene objects.
5. Add the CharacterControl to the model and register it to the physicsSpace.
6. Attach the visible model to the rootNode.

```
private CharacterControl character;
private Node model;
...
public void simpleInitApp() {
 ...
 CapsuleCollisionShape capsule = new CapsuleCollisionShape(3f, 4f);
 character = new CharacterControl(capsule, 0.05f);
 character.setJumpSpeed(20f);
 model = (Node) assetManager.loadModel("Models/Oto/Oto.mesh.xml");
 model.addControl(character);
 bulletAppState.getPhysicsSpace().add(character);
 rootNode.attachChild(model);
 ...
}
```

**Did you know?** A CapsuleCollisionShape is a cylinder with rounded top and bottom. A capsule rotated upright is a good collision shape for a humanoid character since its roundedness reduces the risk of getting stuck on obstacles.

</div>

## Set Up AnimControl and AnimChannels

Create several AnimChannels, one for each animation that can happen simultaneously. In this example, you create one channel for walking and one for attacking. (Because the character can attack with its arms and walk with the rest of the body at the same time.)

```

private AnimChannel animationChannel;
private AnimChannel attackChannel;
private AnimControl animationControl;
...
public void simpleInitApp() {
 ...
 animationControl = model.getControl(AnimControl.class);
 animationControl.addListener(this);
 animationChannel = animationControl.createChannel();
 attackChannel = animationControl.createChannel();
 attackChannel.addBone(animationControl.getSkeleton().getBone("upperArm"));
 attackChannel.addBone(animationControl.getSkeleton().getBone("arm"));
 attackChannel.addBone(animationControl.getSkeleton().getBone("hand"));
 ...
}

```

The attackChannel only controls one arm, while the walking channels controls the whole character.

## Add ChaseCam / CameraNode

```

private ChaseCamera chaseCam;

...
public void simpleInitApp() {
 ...
 flyCam.setEnabled(false);
 chaseCam = new ChaseCamera(cam, model, inputManager);
 ...
}

```

## Handle Navigation

Configure custom key bindings for WASD keys that you will use to make the character walk. Then calculate the vector where the user wants the character to move. Note the use of the special `setWalkDirection()` method below.

```
// track directional input, so we can walk left-forward etc
private boolean left = false, right = false, up = false, down = fal
...
public void simpleInitApp() {
 ...
 // configure mappings, e.g. the WASD keys
 inputManager.addMapping("CharLeft", new KeyTrigger(KeyInput.KEY_A));
 inputManager.addMapping("CharRight", new KeyTrigger(KeyInput.KEY_D));
 inputManager.addMapping("CharForward", new KeyTrigger(KeyInput.KEY_W));
 inputManager.addMapping("CharBackward", new KeyTrigger(KeyInput.KEY_S));
 inputManager.addMapping("CharJump", new KeyTrigger(KeyInput.KEY_F));
 inputManager.addMapping("CharAttack", new KeyTrigger(KeyInput.KEY_G));
 inputManager.addListener(this, "CharLeft", "CharRight");
 inputManager.addListener(this, "CharForward", "CharBackward");
 inputManager.addListener(this, "CharJump", "CharAttack");
 ...
}
```

Respond to the key bindings by setting variables that track in which direction you will go. This allows us to steer the character forwards and to the left at the same time. **Note that no actual walking happens here yet!** We just track the input.

```

@Override
public void onAction(String binding, boolean value, float tpf) {
 if (binding.equals("CharLeft")) {
 if (value) left = true;
 else left = false;
 } else if (binding.equals("CharRight")) {
 if (value) right = true;
 else right = false;
 } else if (binding.equals("CharForward")) {
 if (value) up = true;
 else up = false;
 } else if (binding.equals("CharBackward")) {
 if (value) down = true;
 else down = false;
 } else if (binding.equals("CharJump"))
 character.jump();
 if (binding.equals("CharAttack"))
 attack();
}

```

The player can attack and walk at the same time. `Attack()` is a custom method that triggers an attack animation in the arms. Here you should also add custom code to play an effect and sound, and to determine whether the hit was successful.

```

private void attack() {
 attackChannel.setAnim("Dodge", 0.1f);
 attackChannel.setLoopMode(LoopMode.DontLoop);
}

```

Finally, the update loop looks at the directional variables and moves the character accordingly. Since this is a special kinematic CharacterControl, we use the `setWalkDirection()` method.

The variable `airTime` tracks how long the character is off the ground (e.g. when jumping or falling) and adjusts the walk and stand animations accordingly.

```

private Vector3f walkDirection = new Vector3f(0,0,0); // stop

private float airTime = 0;

```

```

public void simpleUpdate(float tpf) {
 Vector3f camDir = cam.getDirection().clone();
 Vector3f camLeft = cam.getLeft().clone();
 camDir.y = 0;
 camLeft.y = 0;
 camDir.normalizeLocal();
 camLeft.normalizeLocal();
 walkDirection.set(0, 0, 0);

 if (left) walkDirection.addLocal(camLeft);
 if (right) walkDirection.addLocal(camLeft.negate());
 if (up) walkDirection.addLocal(camDir);
 if (down) walkDirection.addLocal(camDir.negate());

 if (!character.onGround()) { // use !character.isOnGround() if the
 airTime += tpf;
 } else {
 airTime = 0;
 }

 if (walkDirection.lengthSquared() == 0) { //Use lengthSquared() (
 if (!"stand".equals(animationChannel.getAnimationName())) {
 animationChannel.setAnim("stand", 1f);
 }
 } else {
 character.setViewDirection(walkDirection);
 if (airTime > .3f) {
 if (!"stand".equals(animationChannel.getAnimationName())) {
 animationChannel.setAnim("stand");
 }
 } else if (!"Walk".equals(animationChannel.getAnimationName()))
 animationChannel.setAnim("Walk", 0.7f);
 }
}

walkDirection.multLocal(25f).multLocal(tpf); // The use of the first
character.setWalkDirection(walkDirection); // THIS IS WHERE THE WALK
}

```

This method resets the walk animation.

```
public void onAnimCycleDone(AnimControl control, AnimChannel channel
 if (channel == attackChannel) channel.setAnim("stand");
}

public void onAnimChange(AnimControl control, AnimChannel channel,
```



## See also

- <http://hub.jmonkeyengine.org/forum/topic/bettercharactercontrol-in-the-works/>
- [documentation](#), [physics](#), [input](#), [animation](#), [character](#), [NPC](#), [collision](#)

</div>

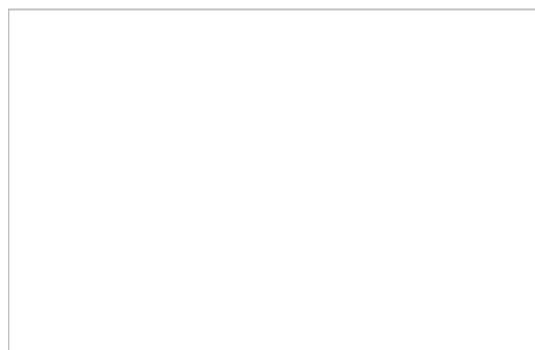
## title: Simple Water

# Simple Water

jMonkeyEngine offers a SimpleWaterProcessor that turns any quad (flat rectangle) into a reflective water surface with waves. You can use this quad for simple limited water surfaces such as water troughs, shallow fountains, puddles, shallow water in channels. The SimpleWaterProcessor has less performance impact on your game than the full featured [SeaMonkey WaterFilter](#); the main difference is that the SimpleWaterProcessor does not include under-water effects.

Here is some background info for JME3's basic water implementation:

- <http://www.jmonkeyengine.com/forum/index.php?topic=14740.0>
- [http://www.bonzaisoftware.com/water\\_tut.html](http://www.bonzaisoftware.com/water_tut.html)
- <http://www.gametutorials.com/Articles/RealisticWater.pdf>



</div>

## SimpleWaterProcessor

A JME3 scene with water can use a `com.jme3.water.SimplewaterProcessor` (which implements the SceneProcessor interface).

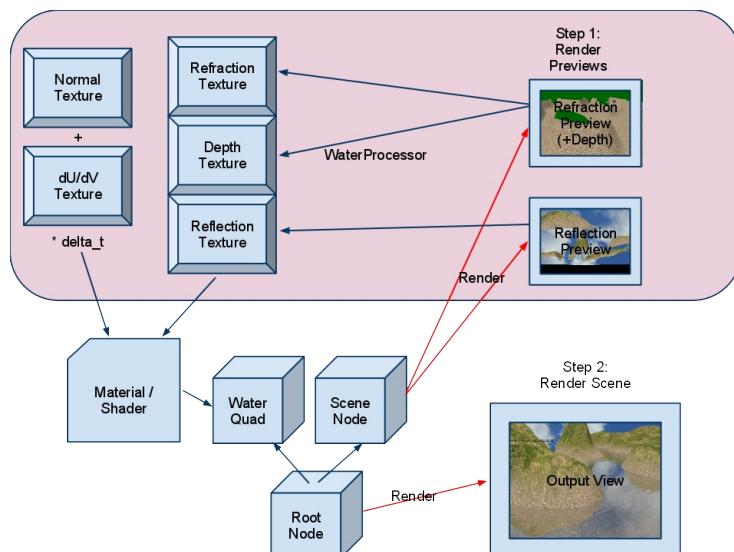
To achieve a water effect, JME3 uses shaders and a special material, `Common/MatDefs/Water/SimpleWater.j3md`. The water surface is a quad, and we use normal map and dU/dV map texturing to simulate the waves.

1. Every frame, we render to three texture maps:
  - For the water surface (reflection), we take a snapshot of the environment, flip it

upside down, and clip it to the visible water surface. Note that we do not actually use a “water texture” color map: The “texture” of the water is solely a distorted reflection.

- For the “wavy” distortion (refraction), we use the derivative of a normal map, a  $dU/dV$  map.
  - For the fogginess of water (depth) we use a depth map from the terrains z-buffer.
2. In the shaders, we add all of the texture maps together.
    - For the “bumpy” displacement of the waves, we use a normal map and a  $du/dv$  map that are shifted against each other over time to create the wave effect.
    - For the light reflection vectors on the water surface, we use the Fresnel formula, together with normal vectors.
    - We add specular lighting.
  3. (For the underwater caustics effect, we use splatted textures. – WIP/TODO)

## Usage



1. Create a `mainScene` Node
  - i. Attach the `mainScene` Node to the `rootNode`
2. Load your `scene` Spatial
  - i. Add a light source to the `scene` Spatial
  - ii. Attach the `scene` Spatial to the `mainScene` Node
3. Load your `sky` Geometry
  - i. Attach the `sky` Geometry to the `mainScene` Node
4. Create the SimpleWaterProcessor `waterProcessor`
  - i. Set the processor's `ReflectionScene` to the `mainScene` Spatial (!)
  - ii. Set the processor's Plane to where you want your water surface to be
  - iii. Set the processor's `WaterDepth`, `DistortionScale`, and `WaveSpeed`

- iv. Attach the processor to the `viewPort`
- 5. Create a Quad `quad`
  - i. Set the quad's `TextureCoordinates` to specify the size of the waves
- 6. Create a `water` Geometry from the Quad
  - i. Set the water's translation and rotation (same Y value as Plane above!)
  - ii. Set the water's material to the processor's output material
  - iii. Attach the `water` Geometry to the `rootNode`. (Not to the `mainScene`!)

## Sample Code

The sample code can be found in `jme3/src/jme3test/water/TestSimpleWater.java` and `jme3/src/jme3test/water/TestSceneWater.java`.

Here is the most important part of the code:

```
// we create a water processor
SimpleWaterProcessor waterProcessor = new SimpleWaterProcessor(assetManager);
waterProcessor.setReflectionScene(mainScene);

// we set the water plane
Vector3f waterLocation=new Vector3f(0, -6, 0);
waterProcessor.setPlane(new Plane(Vector3f.UNIT_Y, waterLocation));
viewPort.addProcessor(waterProcessor);

// we set wave properties
waterProcessor.setWaterDepth(40); // transparency of water
waterProcessor.setDistortionScale(0.05f); // strength of waves
waterProcessor.setWaveSpeed(0.05f); // speed of waves

// we define the wave size by setting the size of the texture coordinates
Quad quad = new Quad(400,400);
quad.scaleTextureCoordinates(new Vector2f(6f,6f));

// we create the water geometry from the quad
Geometry water=new Geometry("water", quad);
water.setLocalRotation(new Quaternion().fromAngleAxis(-FastMath.HALF_PI, Vector3f.UNIT_X));
water.setLocalTranslation(-200, -6, 250);
water.setShadowMode(ShadowMode.Receive);
water.setMaterial(waterProcessor.getMaterial());
rootNode.attachChild(water);
```



## Settings

You can lower the render size to gain higher performance:

```
waterProcessor.setRenderSize(128,128);
```

The deeper the water, the more transparent. (?)

```
waterProcessor.setWaterDepth(40);
```

A higher distortion scale makes bigger waves.

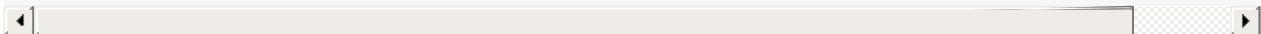
```
waterProcessor.setDistortionScale(0.05f);
```

A lower wave speed makes calmer water.

```
waterProcessor.setWaveSpeed(0.05f);
```

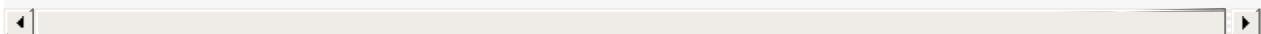
If your scene does not have a lightsource, you can set the light direction for the water:

```
waterProcessor.setLightDirection(new Vector3f(0.55f, -0.82f, 0.15f
```



Instead of creating a quad and specifying a plane, you can get a default waterplane from the processor:

```
Geometry waterPlane = waterProcessor.createWaterGeometry(10, 10);
waterPlane.setLocalTranslation(-5, 0, 5);
waterPlane.setMaterial(waterProcessor.getMaterial());
```



You can offer a switch to set the water Material to a static texture – for users with slow PCs.

```
</div>
```

# title: Android Support in the jMonkeyEngine

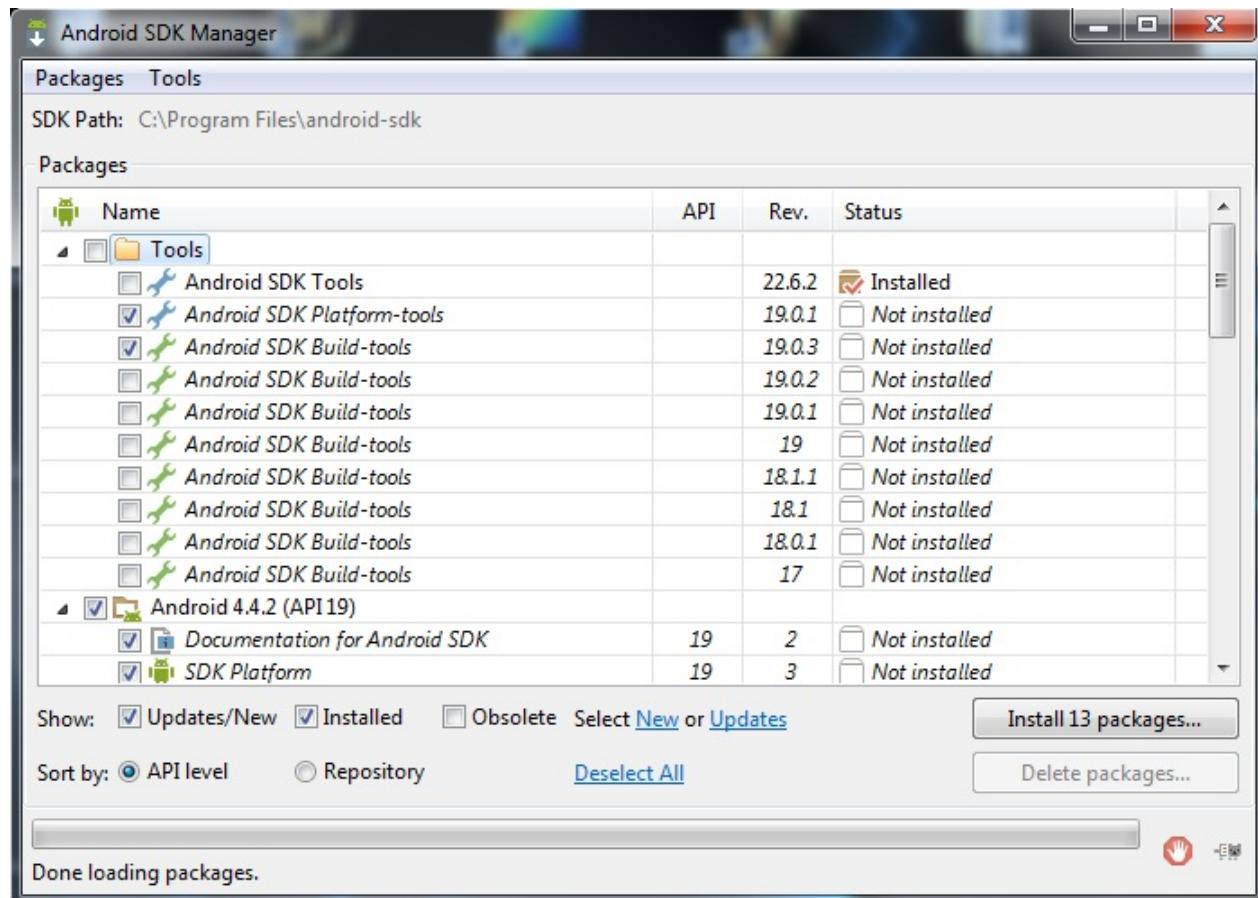
## Android Support in the jMonkeyEngine

jMonkeyEngine supports android deployment, and it's fully integrated in the SDK. The android support is in constant enhancement so if you have questions or suggestions, please leave a comment on the [Android forum thread](#)!

## Requirements

### Developer Requirements

- Install [jMonkeyEngine SDK](#) (Remember to install [JDK](#))
- Install [Android SDK Manager 22.6.2](#) or newer.
  1. (You don't need the ADT Bundle for Windows. Choose "USE AN EXISTING IDE" and download the SDK Tools)
  2. It's recommended to choose another destination folder.
  3. Start the SDK Manager and install the default selected 13 packages (accept licenses)



- (Optional) Install NBAndroid in the jMonkeyEngine SDK:
  1. Go to Tools→Plugins→Available Plugins.
  2. Install the “Android” plugin.

## User Requirements

- Android 2.3 device or better with development mode (“USB-Debugging”) enabled
- Graphic card that supports OpenGL ES 2.0 or better
- [Does my phone meet the requirements necessary to run jMonkeyEngine 3?](#)
- Remember to install the driver software on your computer

## Features

### jMonkeyEngine3 Android Integration

- JME app encapsulated in an Android activity (the AndroidHarness).
- JME display in a GLSurfaceView.
- Android touch events are encapsulated into JME touch event or translated into mouse events.
- Errors are handled universally in the Harness.

- Lifecycle management
  - Leaving the activity with the Back key destroys the app (quit).
  - Leaving the activity with the Home key freezes the app in the background. When you return, the app is in the same state as when you left it (pause).
- Currently supports all jmetests except:
  - Post processing filters. Functional but most of the time very slow
  - Shadows. Functional but most of the time very slow
  - Water. Functional but very slow.

## jMonkeyEngine SDK Android Integration

Mobile deployment is a “one-click” option next to Desktop/WebStart/Applet deployment in the jMonkeyEngine SDK.

- Automatic creation of Android harness and settings.
- Ant script build target creates APK file.
- Ant script run target executes APK on Android Emulator or mobile device.

## Instructions

1. Make sure you have installed the Android Project Support into the jMonkeyEngine SDK.
2. Open the jMonkeyEngine SDK Options>Mobile and enter the path to your Android SDK directory, and click OK. E.g. `c:\Program Files\android-sdk`

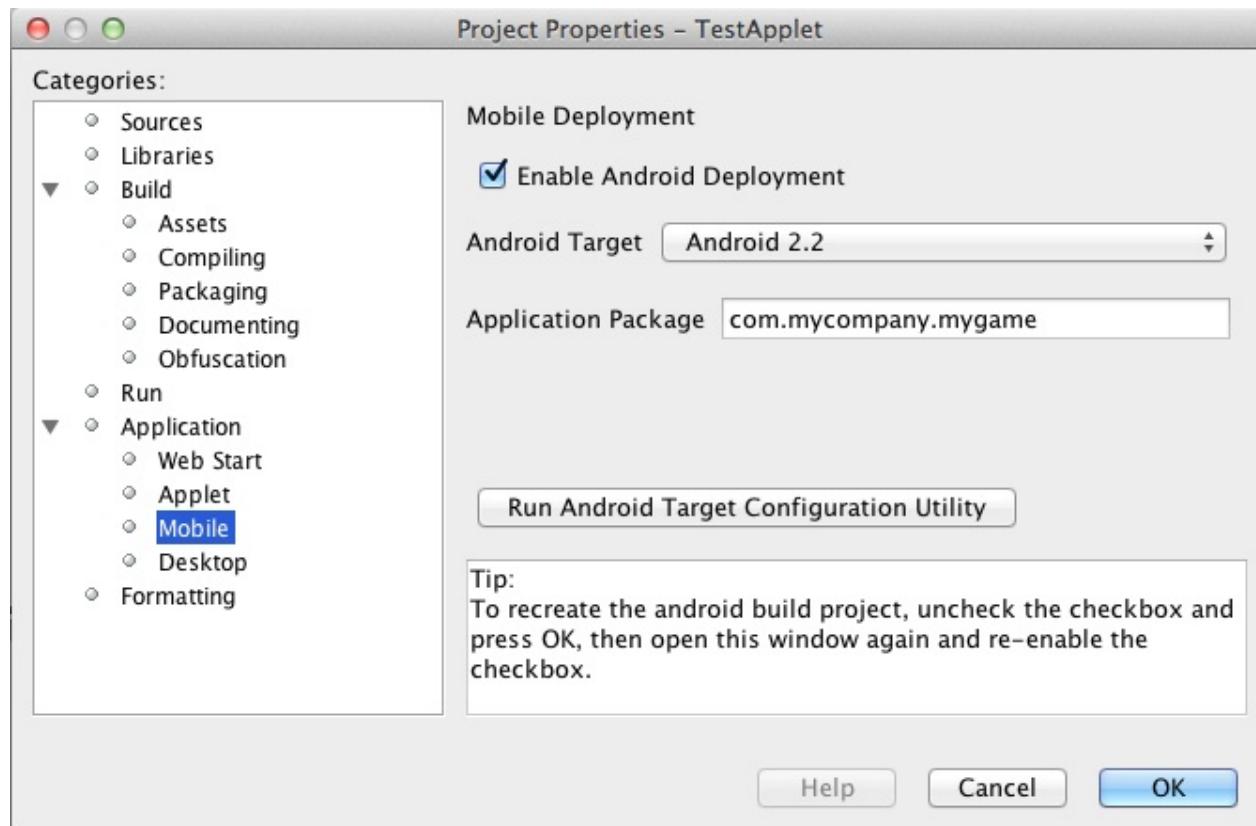
## Activate Android Deployment

1. Open an existing JME3 project, or create a new JME3 project.
2. Right-click the project node in the Projects Window and open the Properties.
3. In the Application>Android Properties, enable Android Deployment, and select an Android target. E.g. `Android 4.2.2`

This creates a “mobile” folder in your projects directory. This folder contains a complete android project with correct settings to run the application using the AndroidHarness.

4. (Restart the jMonkeyEngine)
5. A Mobile Files node appears in the Project window.

It lets you edit the `MainActivity.java`, the `AndroidManifest.xml`, and `build.properties`.



The Android deployment option creates a separate sub-project for android and makes the main project and associated libraries available to the sub-project as libraries. The sub-project can be edited using NBAndroid (see below) or using Eclipse or any other IDE that supports standard android projects. Normally you do not need to edit the android project files. Exceptions are described further below. *The libraries are first added to the android sub-project when the main project is built for the first time.*

## Build and Run

Open your game project in the jMonkeyEngine SDK.

### Building

1. Right-click the project node to build your project.  
An APK file is created in the “dist” folder.

### Running

1. Right-click the project node and choose Set Main Project.
2. Select the “Android Device” build configuration next to the Run toolbar button.
3. *Make sure “Compile on Save” is disabled in your project preferences.*
4. Run the application on a connected phone by right-clicking it and selecting “Run” or press the “Play” button in the toolbar.
5. The application will run and output its log to the “Output” window of the SDK.

6. When finished testing, click the “x” next to the run status in the lower right to quit the logging output.

The default android run target uses the default device set in the Android configuration utility. If you set that to a phone you can run the application directly on your phone, the emulator does not support OpenGL ES 2.0 and cannot run the engine.

When running your application on some Android devices, having a debugging session open via certain IDEs **will significantly lower performance** (some reports suggest a drop from 60 FPS to 4-8 FPS) until the debug session is detached or the application is started directly from the device.

Optionally, download [Jme3Beta1Demo.apk](#)

During build, the libraries and main jar file from the main project are copied to the android project libs folder for access. During this operation the desktop-specific libraries are replaced with android specific JARs.

**Be aware that logging has been identified as having a significant performance hit in Android applications. If getting poor performance please try turning logging either down or off and retesting.**

</div>

## Installing NBAndroid

Activating the nbandroid plugin in the jMonkeyEngine SDK is optional, but recommended. You do not need the nbandroid plugin for Android support to work, however nbandroid will not interfere and will in fact allow you to edit the android source files and project more conveniently. To be able to edit, extend and code android-specific code in Android projects, install NBAndroid from the update center:

1. Open Tools→Plugins→Settings
2. Go to Tools→Plugins→Available Plugins.
3. Install the NbAndroid plugin. (Will show up as Android)

\*If the android plugin is not in that list follow [these instructions](#).\*

## Notes

- The package name parameter is only used when creating the project and only sets the android MainActivity package name
- The needed android.jar comes with the plugin and is automatically added to the android build
- All non-android project libraries are automatically excluded from the android build. This

is currently hard-coded in the build script, check nbproject/mobile-impl.xml for the details.

- The main application class parameter for the AndroidHarness is taken from the jme3 project settings when enabling android deployment. Currently it is not updated when you change the main class package or name.
- When you disable the mobile deployment option, the whole “mobile” folder is deleted.
- The “errors” shown in the MainActivity are wrongly displayed only in the editor and will disappear when you install NBAndroid (see below).
- To sign your application, edit the mobile/build.properties file to point at valid keystore files.

## Android Considerations

You can use the jMonkeyEngine SDK to save (theoretically) any jMonkeyEngine app as Android app. But the application has to be prepared for the fact that Android devices have a smaller screen resolution, touchscreens instead of mouse buttons, and (typically) no keyboards.

- **Inputs:** Devise an alternate control scheme that works for Android users (e.g. using com.jme3.input.controls.TouchListener). This mobile scheme is likely quite different from the desktop scheme.
- **Effects:** Android devices do not support 3D features as well as PCs – or even not at all. This restriction includes post-processor filters (depth-of-field blur, bloom, light scattering, cartoon, etc), drop shadows, water effects, 3D Audio. Be prepared that these effects will (at best) slow down the application or (in the worst case) not work at all. Provide the option to switch to a low-fi equivalent!
- **Nifty GUI:** Use different base UI layout XML files for the mobile version of your app to account for a different screen resolution.

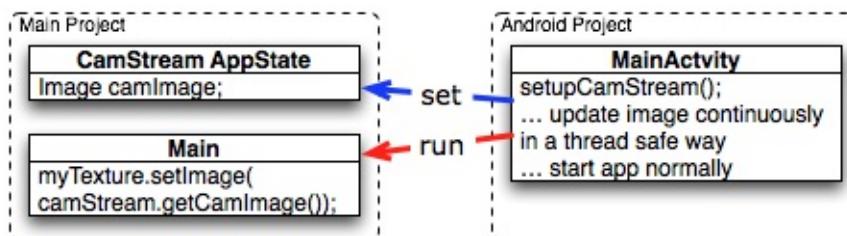
**Best Practice:** Ideally, you write the core application code in a way that it checks for the environment it's being run on, and automatically adapts the device's limitations by switching off effects, changing input mechanisms etc. Learn how to [read graphic card capabilities](#) here.

## Using Android specific functions

As described above, you should always try to design your application as platform independent as possible. If your game becomes a sudden hit on android, why not release it on Facebook as an applet as well? When your application is designed in a platform

independent way, you don't have to do much more but check a checkbox to deploy your application as an applet. But what if you want to for example access the camera of an android device? Inevitably you will have to use android specific api to access the camera.

Since the main project is not configured to access the android api directly, you have to install NBAndroid (see above) to be able to edit the created android project in the SDK. After installing, click the “open project” button and navigate to the “mobile” folder inside the main project folder (it should show up with an android “a” icon) and open it.



Although you will use

android specific api, using a camera is not exactly android specific and so you should try to design this part of the application as platform independent as possible as well. As an example, if you want to use the phones camera as an image input stream for a texture, you can create e.g. the AppState that manages the image and makes it available to the application inside the main project (no android code is needed). Then in the android part of the code you make a connection to the camera and update the image in the AppState. This also allows you to easily support cameras on other platforms in the same way or fallback to something else in case the platform doesn't support a camera.

Note that you have to build the whole project once to make (new) classes in the main project available to the android part.

## Signing an APK

When you have a mobile project in the “important files” section you have an “Android Properties” file.

There are 2 entries in this file :

key.store=path/to/your/keystore/on/your/drive/mykeystore.keystore  
key.alias=mykeystorealias

If those entries are filled, the apk will be signed during the build.

You'll be prompted when building to enter the password (twice). It will generate a signed apk in the dist folder of your project.

## More Info

There is currently no proper guidance of running on android. The SDK will later provide tools to adapt the materials and other graphics settings of the Android deployment version automatically.

- [Youtube Video on Android deployment](#)
- [Android Forum Thread \(beta\)](#)
- [Android Forum Thread \(alpha\)](#)

[documentation](#), [sdk](#), [android](#), [deployment](#), [tool](#)

</div>

# title: ATOMIX TUTORIALS

## ATOMIX TUTORIALS

### FIRST WORDS

Dear monkeys,

Long time ago I planed to make tutorial series in making games in JME3 and done some examples. I waited for our project to reach a stable status. May be now is the best time to show them up! A lot of changes have been made to the core and also a lot of excited and amazing contributions into the world of JME this day (you can name some).

OK, my project had changed to Atomix Tutorial Series (short as ATS).

These tutorials will lead you through adventures in game development in some games and genres. I will try to get much detail as I can. Hopefully this will become a valuable guidebook for monkeys want to join the jungle! 

Contact me as @atomix in the forum

Google+: <https://plus.google.com/u/1/113026606091014198679/about>

I decided not to do the voice since it will make you guys too difficult to follow cause I'm not a native speaker. So I will write the tutorials down step by step and also do slideshows and videos for them. Typos correction are always welcome ! 

If you are very new to Java, it's not recommended for you to take these tutorials at first. They require quite advanced techniques, suited for advanced Java developer which new to game programming, or one come from other engine. They are written detailed and intended like to squeeze all the juice out of the things.

---

## Content

- Chapter 1 : Card Game [cardsgame](#)
- Chapter 2 : RPG Game - Blade of the immortal [bladegame](#)
- Chapter 3 : Shot-em-all - FPS game [shootemgame](#)
- Chapter 4 : Mayan boy - 3D Platform Puzzle game [mayangame](#)

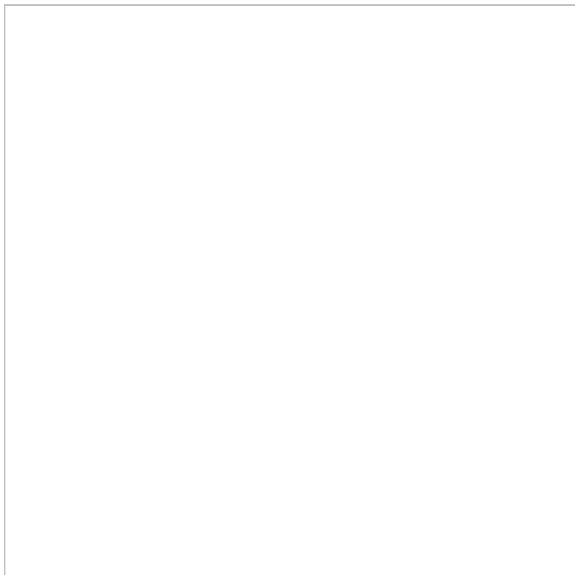
- Chapter 5 : EuroKick - Sport management game (Soccer) [kickgame](#)
- Chapter 6 : RTS Multiplayer game - HeavenRTS [heavenrtsgame](#)
- Chapter 7 : Ancient Monkey world [mmorpg](#)
- Chapter 8 : GreenBeret [greenberet](#)
- Chapter 9 : Prince of dark [prince](#)

## Extra stuffs:

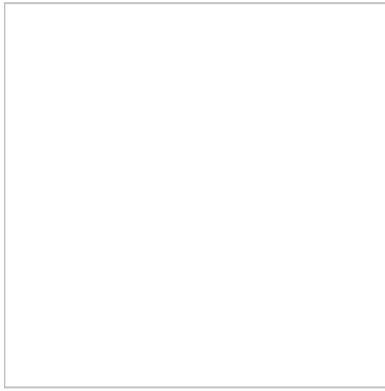
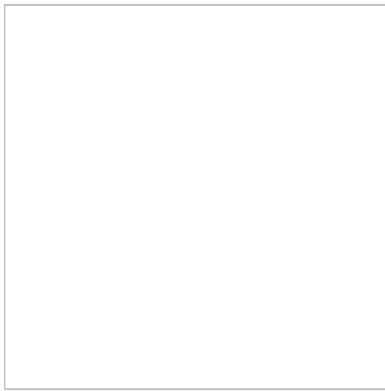
- Framework: [Atom framework](#) , [Researches MMORPG techs](#) , [Atom for Cloud framework](#)
- Modeling : [Character in Blender](#) , [Effects](#)
- Concept & Speed painting:
- Design : [design](#)
- AI : [AtomAI Framework and Tools](#) , [AI Researches](#)
- GUI : [AtomExGui](#)
- Scripting : [scripting](#), [AtomScripting](#)
- Everyday stuffs:

## Screenshots:

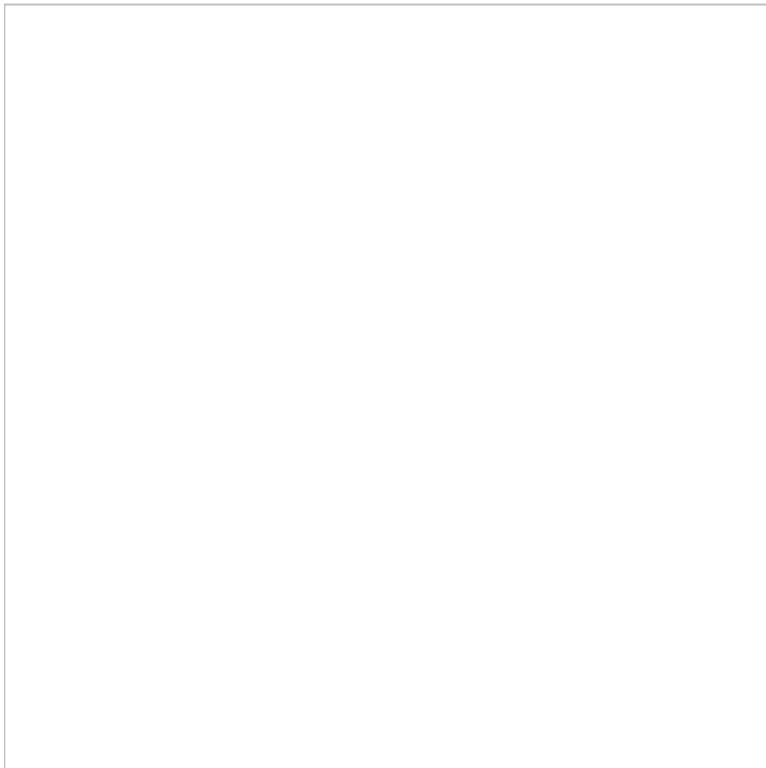
### Card game:

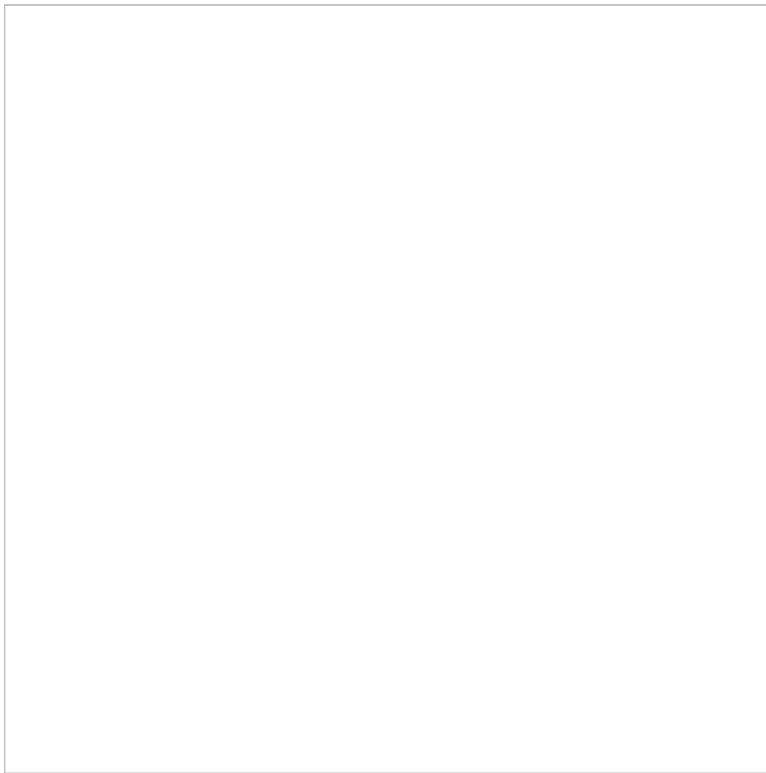


**Blade of the immortals** [bladegame](#)

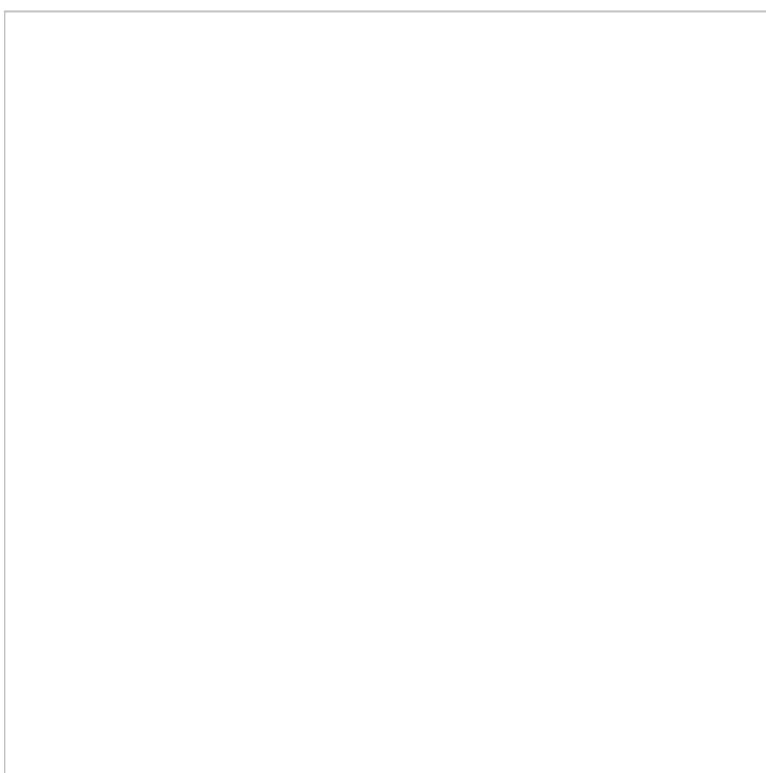


## **Euro Kick**





**Heaven – RTS Game.**



</div>

## **Videos**

</div>

## **title: Blendswap Arcade Game Contest**

### **Blendswap Arcade Game Contest**

The Blendswap Arcade contest is a game making contest where all entries must use the same scene from Blendswap. All participants have a week to create their game. It is currently planned to be a monthly contest.

</div>

### **Rules**

The rules are:

You may use or even make other assets, but;  
You may NOT extend the confines of the game level.  
You may use any existing open source code, including your own.  
The scene does not in any way dictate the gameplay.  
The game must be completely open source and made freely available  
Game submission must be an executable, preferably fully functional



Game entries are judged on:

Utilization - Appropriate & innovative use of the scene.

Completion - Completeness of the game: Keep it simple!

### **Blendswap Arcade Contest 19.05-27.05**

This was the first Blendswap arcade contest. It used this model:



<http://www.blendswap.com/blends/view/71013>

</div>

## Entries

<http://hub.jmonkeyengine.org/forum/topic/blendswap-arcade-submission-exterminator/>

<http://hub.jmonkeyengine.org/forum/topic/blendswap-arcade-submission-the-climb/>

<http://hub.jmonkeyengine.org/forum/topic/blendswap-arcade-submission-marbleway/>

<http://hub.jmonkeyengine.org/forum/topic/blendswap-arcade-shadow-unit/>

<http://hub.jmonkeyengine.org/forum/topic/blendswap-arcade-submission-fire-department/>

<http://hub.jmonkeyengine.org/forum/topic/blendswap-arcade-submission-hostage/>

<http://hub.jmonkeyengine.org/forum/topic/blendswap-arcade-submission-ponyrescue/>

<http://hub.jmonkeyengine.org/forum/topic/blendswap-arcade-submission-airrace/>

<http://hub.jmonkeyengine.org/forum/topic/blendswap-arcade-submission-meta-aware/>

<http://hub.jmonkeyengine.org/forum/topic/blendswap-arcade-submission-corporate-window-washer-strike-force/>

<http://hub.jmonkeyengine.org/forum/topic/blendswap-arcade-submission-catch-the-chickens/>

The entries in this list may not include all entries. It was not required to make a forum page.

</div>

## Winners

<http://hub.jmonkeyengine.org/2014/09/blendswap-arcade-at-long-last-we-have-a-final-verdict/>

</div>

## title: Building jMonkeyEngine 3 from the Sources

# Building jMonkeyEngine 3 from the Sources

We recommend downloading the [jMonkeyEngine SDK](#) - but of course you can also build the jMonkeyEngine yourself from the sources. In this case, you need the [Subversion](#) file version system installed (svn).

1. **Checkout:** Checkout the Subversion repository.

```
svn checkout http://jmonkeyengine.googlecode.com/svn/branches/3
```



- You can leave login and password empty

2. **Build:** Execute `ant jar`
  - This compiles the JAR files in `dist/libs/*`
3. **Javadoc:** Execute `ant javadoc`
  - This generates javadocs in the `dist/javadoc` directory.
4. **Run:** Execute `ant run`
  - This runs the TestChooser where you can browse examples.
5. **Use:** Create a Java SE project and place all JARs from the `dist/lib` directory on the classpath.
  - You can now extend your first game from `com.jme3.app.SimpleApplication`.

For a detailed description of the created jar files see [this list](#).

---

Learn more about:

- [Setting up JME3 on the commandline \(generic\)](#).
- [Building JME3 from the sources with NetBeans](#)

[documentation](#), [install](#)

</div>

## title: Setting up JME3 in Netbeans 6+

# Setting up JME3 in Netbeans 6+

You are welcome to try out the new jME3, and contribute patches and features! This document shows how to download, set up, build, and run the latest development version from the sources. These instructions work in NetBeans IDE 6 or better.

Note: In the following, always replace “~” with the path to your home directory.

## Downloading the Sources

Check out the sources from the repository. (The following NetBeans instructions are equivalent to executing `cd ~/NetBeansProjects; svn checkout`

`http://jmonkeyengine.googlecode.com/svn/branches/3.0final/engine jme3` on the commandline.)

1. In NetBeans go to Team > Subversion > Checkout
  - i. Repository URL: `https://jmonkeyengine.googlecode.com/svn`
  - ii. You can leave user/pw blank for anonymous access.
2. Click Next
  - i. Repository Folders: `branches/3.0final/engine`
  - ii. Enable the checkbox to Skip “engine” and only checkout its contents.
  - iii. Local Folder: `~/NetBeansProjects/jme3`
3. Click Finish and wait.

The jme3 project opens in the Project window. It already includes a working ANT build script for building and running.

Look into the Libraries node and confirm that the project depends on the following libraries in the classpath:

- j-ogg-oggd.jar
- j-ogg-vorbisd.jar
- jbullet.jar
- stack-alloc.jar
- vecmath.jar
- lwjgl.jar

- jME3-lwjgl-natives.jar
- jinput.jar
- eventbus.jar
- nifty-default-controls.jar
- nifty-examples.jar
- nifty-style-black.jar
- nifty.jar
- jglfont-core.jar
- xmlpull-xpp3.jar
- android.jar
- jME3-bullet-natives.jar
- gluegen-rt.jar
- joal.jar
- jogl-all.jar
- jME3-natives-joal.jar
- jME3-openal-soft-natives-android.jar

For a detailed description of the separate jar files see [this list](#).

## Build the Project and Run a Sample App

1. Right-click the jme3 project node and “Clean and Build” the project.
2. In the Projects window, open the `Test` folder which contains the sample apps.
3. Every file with a Main class (for example `jme3test.model/TestHoverTank.java` or `jme3test.game/CubeField.java`) is an app.
4. Right-click a sample app and choose “Run File” (Shift-F6).
5. Generally in sample apps:
  - i. the mouse and the WASD keys control movement
  - ii. the Esc key exits the application

## Optional: Javadoc Popups and Source Navigation in NetBeans

If you are working on the jme3 sources:

1. In the Projects window, right-click the jme3 project and choose Generate Javadoc. Wait.
2. Confirm in the Files window that the javadoc has been created in  
`~/NetBeansProjects/jme3/dist/javadoc`
3. In the editor, place the caret in a jme class and press ctrl-space to view javadoc.

If you are working on a game project that depends on jme3:

1. First follow the previous tip. (In the future, we may offer jme javadoc as download instead.)
2. In your game project, right-click the Libraries node and choose “Properties”.
3. In the Library properties, select jme3.jar and click the Edit button.
  - i. For the Javadoc field, browse to `~/NetBeansProjects/jme3/dist/javadoc` . Check “as relative path” and click select.
  - ii. For the Sources field, browse to `~/NetBeansProjects/jme3/src` . Check “as relative path” and click select.
  - iii. Click OK.
4. In the editor, place the caret in a jme class and press ctrl-space to view javadoc. Ctrl-click any jme3 method to jump to its definition in the sources.

This tip works for any third-party JAR library that you use. (You may have to download the javadoc/sources from their home page separately).

---

Sources used: [BuildJme3](#), [NetBeans tutorial from forum documentation](#), [install](#)

</div>

## **title: Contributions**

# **Contributions**

The following list contains additional content for jMonkeyEngine 3 contributed by users. They are, as is the engine itself, open-source - Feel free to download and use them for your projects. :)

To install a jMonkeyEngine SDK plugin, go to Tools→Plugins→Available Plugins.  
(Currently only jME 3.0 stable SDK, jME 3.1 alpha does not yet support plugins)  
</div>

## **Github Repo**

This seems to be the main repository for jmonkey contributions:

<https://github.com/jMonkeyEngine-Contributions>

## **Subversion Repo**

This seems to be the main repository for jME SDK plugins:

<https://code.google.com/p/jmonkeyplatform-contributions/source/browse/#svn%2Ftrunk>

## **Forum: Contributions**

This is the forum category where you can find other people's contributions or present your own contribution: <http://hub.jmonkeyengine.org/c/contribution-depot-jme3>

## **Other Repos**

There are other repositories for code sources. A list of weblinks follows:

TODO: Organize these links (e.g. alphabetically or by topic).

- <http://sourceforge.net/projects/jmonkeycsg/>
- [https://github.com/davidB/jme3\\_skel](https://github.com/davidB/jme3_skel)

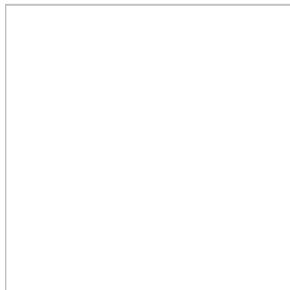
# Frameworks

Those are premade classes/functionalities that you can use.

TODO: Either remove these old entries or make them up-to-date and add more.

## ImagePainter

A fairly complete set of painting tools for editing jME3 Images from code



<b>Contact person</b>	<a href="#">zarch</a>
<b>Documentation</b>	<a href="#">Forum Post, full javadoc in plugin</a>
<b>Available as SDK plugin</b>	Yes
<b>Work in progress</b>	No

## tonegodGUI

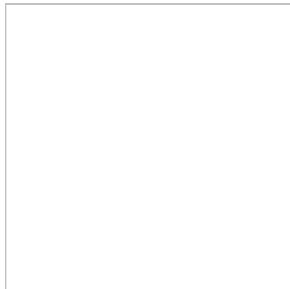


A Native GUI Library for JME3

<b>Contact person</b>	<a href="#">t0neg0d</a>
<b>Documentation</b>	<a href="#">Wiki Page</a>
<b>Available as SDK plugin</b>	Yes
<b>Work in progress</b>	Yes

## Shaderblow

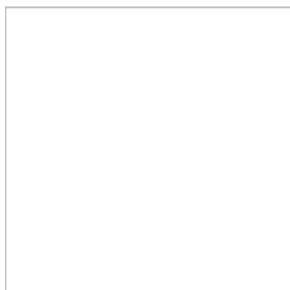
The “Shaderblow” library contains various shader effects, e.g. refraction, particles, forcefields, grayscale and much more.



<b>Contact person</b>	mifth
<b>Documentation</b>	<a href="#">Wiki Page</a>
<b>Available as SDK plugin</b>	Yes
<b>Work in progress</b>	Yes

## Cubes

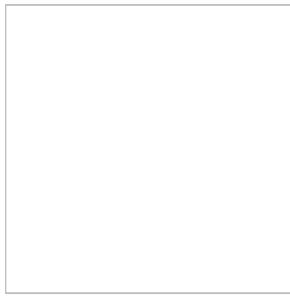
A framework for block worlds (also called “bloxel” or “minecraft-like”) - It offers an easy way to add/remove blocks, block skinning, useful tools and user-specific behaviours.



<b>Contact person</b>	destroflyer
<b>Documentation</b>	<a href="#">Wiki Page</a>
<b>Available as SDK plugin</b>	Yes
<b>Work in progress</b>	Yes

## Zay-ES Entity System

A self-contained thread-capable entity system.



<b>Contact person</b>	Paul Speed (pspeed)
<b>Documentation</b>	<a href="#">Wiki Page</a>
<b>Available as SDK plugin</b>	Yes
<b>Work in progress</b>	Seems fairly complete

## Assets packs

*No contributions yet*

</div>

## Want to commit something yourself?

If you have a framework/assets pack/whatever you want to contribute, please check out our [Contribution Depot](#).

</div>

## Forgot something?

Well, this is a wiki page - Please add projects that are available or keep the provided information up-to-date if you want.

</div>

## **title: Eclipse + jME3 + Android + JNI/NDK**

# **Eclipse + jME3 + Android + JNI/NDK**

This guide will show you how to effectively set up your Eclipse environment for not only a basic JME3 project, but an Android project, an assets project, and a JNI project (including the ability to build for Android through NDK).

While you do not have to create a project of each type to use Eclipse and this guide, note that the Android and JNI projects do require having a base Eclipse project to be applicable using this guide.

The code samples provided in this guide should work for any IDE, but setting up/configuring your IDE to do what Eclipse does in this particular guide may require some creative workarounds.

Please note that projects that use Android UI elements or are created using Eclipse **can not** be ported to iOS. See [this page](#) on how to create a cross-platform project using the jME SDK that **can also be edited in the Eclipse IDE**. This saves you the setup work outlined below.

</div>

## **Setting up JME3**

I will assume you already have downloaded (and built, if necessary) the JME3 binaries. If not, either download the [latest SDK release](#), [engine build](#) (recommended for new users), or take a quick detour over to [this guide](#) to build from source (recommended for advanced users).

One directory you should have on hand (or know where to find) is the location of the resulting library (JAR) files.

For the various methods of obtaining the libraries (as described above), the path is usually in one of the following places:

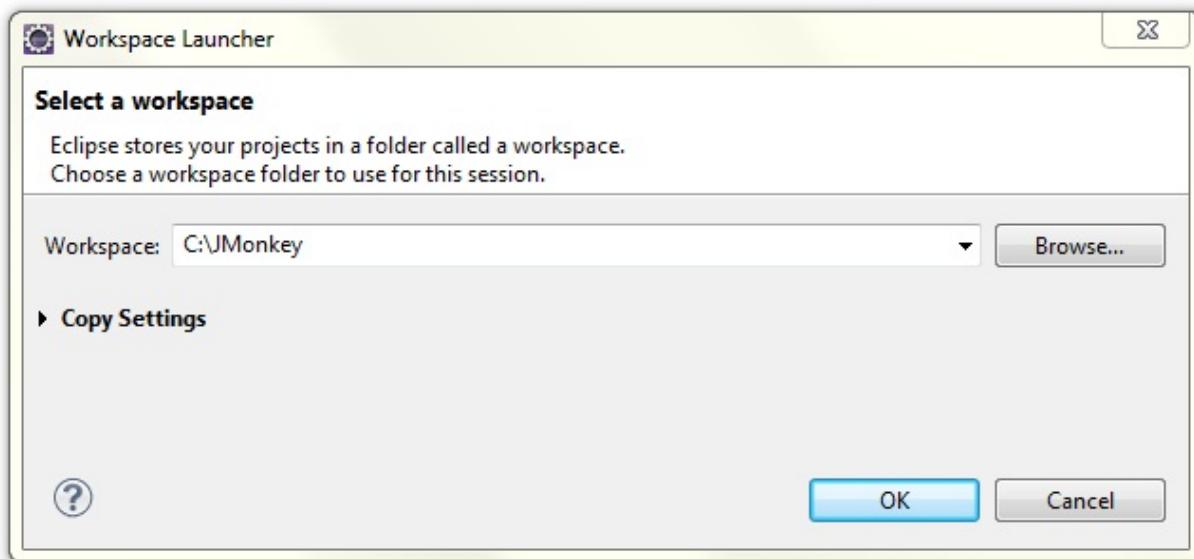
- **SDK:** [SDK Install Dir]/jmonkeyplatform/libs
- **Nightly:** /libs
- **Source:** [SDK Dir]/sdk/build/cluster/libs

Throughout the course of this guide, **do not copy JAR files** to any project directory - leave them in their original spots so they can be easily updated for when a JME3 update comes out or when you rebuild the source.

## Workspace

If you are unfamiliar with Eclipse and how it lays out/stores its configuration, one of the first things that you should be aware of is that almost all settings that are set at a workspace-level (i.e. through *Window → Preferences*) are stored in the workspace.

You can select your workspace either by starting Eclipse (if the initial prompt is enabled) or by navigating to *File → Switch Workspace → Other*.



For this guide, we're going to create a new folder called `JMonkey` in our `c:\` drive, however this folder can be anywhere on your drive (the same applies for users of other operating systems). We will also switch to it as a workspace.

</div>

## Base JME3 Project (Desktops)

### Prerequisites:

- JME3 (downloaded/extracted/built/installed) libraries located

It's time to create our base JME3 project, which is actually relatively easy. Go to *File → New → Java Project*.

NOTE: If you don't see *Java Project*, you are more than likely in a perspective other than Java or Java EE. Simply select *Project...* and then select *Java → Java Project* from the window that appears.

Give your project a name (we'll call ours `JMEclipse-Base`) and click *Finish*. The project should show up in the Project Explorer pane on the left.

First, we need to configure our build path. Right click the new project entry and go to *Build Path → Configure Build Path*, and then select the *Libraries* tab.

There are a number of libraries you're going to need to include, as the 'core' JME3 system requires them.

There will be many libraries in the previously identified `libs` folder that aren't initially used. Certain aspects of JME3 (for instance, collision detection) will require the appropriate JAR if your code requires that functionality. However, it is highly recommended you do NOT include every JAR in the `libs` folder due to the fact that exporting them all when they are not needed will result in a huge jar file (~60MB).

Click *Add External JARs...*, navigate to the JME3 `libs` folder (see *Setting up JME3* if you're not sure where this is), and select each of the following:

- jME3-core.jar
- jME3-desktop.jar
- jME3-jogl.jar
- jME3-lwjgl-natives.jar
- jME3-lwjgl.jar
- jogl-all.jar
- .lwjgl.jar
- vecmath.jar

Add the JARs, hit *OK*, and you've successfully set up the base JME3 build path for your project.

The last bit that must be added are assets. Note that while the SDK has a single way of managing your assets, assets can be handled in a number of ways in other IDEs.

The rule for assets is that their files **must be on the build path**. This means that you can either include the inner folders directly in your project, or you can create a separate project (and thus, a separate JAR) for your assets.

Alternatively, you can keep them in the local directory if you do not wish to package them with the JAR, though this may not be desirable for certain deployments (such as applets, Android apps, or JNLPs).

To include your assets within your base project, follow the *Assets* section without creating a

new project. However, do note that this will make the process of including these assets in your Android project, if you create one, much harder.

Finally, the SDK creates a base source file for you that ends up being the main JME3 entry point. Since we're using Eclipse, we'll have to manually create that file.

Right click your project, go to *New → Class*, and fill out the appropriate information:

- **Package:** your package name (i.e. `com.jme3.test`)
- **Name:** the class name (i.e. `MyGame`)
- **Superclass:** `com.jme3.app.SimpleApplication`

Then hit Finish. This will create and open the new class file. Eclipse should have automatically inserted the method “`simpleInitApp()`” for you. This is where all of your game's start up code will go.

Finally, create a `main` function as the Java entry point:

```
public static void main(String[] args) {
 MyGame app = new MyGame();
 app.start();
}
```

## Running

Running your JME3 project is quite simple in Eclipse. You can either [create a Java run configuration](#) or you can simply hit `CTRL + F11` to run the project (if Eclipse doesn't know what to do, it will ask).

</div>

## Packaging

Packaging your JME3 project is also very straightforward. Assuming you heeded the warning stated earlier about not including every JAR in the `libs` folder, you should be able to right click your base JME3 project, go to *Export...*, select *Java → Runnable JAR file*, specify an output location and click *Finish*.

If you have assets set up as a JAR, make sure to read the *Packaging* section under *Assets*.

## Assets Project

Each JME project has a set of assets that are used to load textures, models, and other resources used by your game.

As mentioned earlier, assets can be located/included in one of several ways. This section will describe how to include your project's assets through the use of a separate JAR file, which has the added advantage of allowing you to update assets without needing to update the JAR itself. If you have a dynamic class-path system set up, this could be very useful.

First, create another **generic** project by going to *File → New → Project... → General → Project* and giving it a name (we'll call ours `JMEclipse-Assets`).

We create a General (non-Java) project for cleanliness because our assets will not require any special build settings or the like.

For new users, it's a good idea to add the initial JME3 folders that the SDK creates, as they are referenced by many other guides on the web. To do this, for each of the following right click on the assets project and go to *New → Folder*, type in the name listed, and hit *Finish*:

- Interface
- MatDefs
- Materials
- Models
- Scenes
- Shaders
- Sounds
- Textures

Although this specific structure is what the JMonkeyEngine SDK generates upon the creation of a new project, it is by no means the only way to structure your project. All asset loading methods will work with folder names other than those listed above.

</div>

## Packaging

Packaging your assets is also a simple process. Right click the assets project and click *Export...* and then select *Java → Jar file*. It will show a list of files you can export; make sure to uncheck all files such as `.classpath`, `.project`, and any `.jardesc` files you may have created. As well, ensure only the resources and assets you want to export are checked.

Check *Export generated class files and resources*, select a destination for the JAR file, and check *Compress the contents of the JAR file*, *Add directory entries*, and *Overwrite existing files without warning*. Click *Finish*.

Optionally, you can click *Next* and specify a `.jardesc` file by checking *Save the description*

of this JAR in the workspace and specifying a location for a `.jardesc` file. That way you can simply double-click the `.jardesc` file and easily re-export your assets when they change (remember, always refresh your project before exporting!).

</div>

## Android Project

### Prerequisites:

- JME3 (downloaded/extracted/built/installed) libraries located
- JME3 Base Project created (as described above)
- Android SDK downloaded and the Android 8 (2.2) target installed (higher API versions work too but may limit compatibility when deploying)
- Assets compiled into a JAR (see *Packaging under Assets*)
- [Eclipse ADT plugin](#) installed

The Android project is a slightly more involved setup project, but is still quite simple, even for new users.

To start, create another Android project by going *File* → *New* → *Project...* → *Android* → *Android Application Project*.

Fill out the following information and then click Next:

- **Application Name:** name of your application (i.e. `JMEclipse Test Project` )
- **Project Name:** name of the project in the workspace (i.e. `JMEclipse-Android` )
- **Package Name:** name of the base package, preferably the same as the one used in the base project (we'll re-use `com.jme3.test` )
- **Minimum Required SDK:** API 8 (Must be AT LEAST SDK 8 for OpenGL ES 2 and JNI)

Most of the lower options are defaulted based off of your ADT configuration and should work as-is.

After clicking *Next*, uncheck *Create activity* (JME3 provides a base activity class). You can check/uncheck *Create custom launcher icon* at your own preference.

Make sure that *Mark this project as a library* is unchecked and hit *Finish* (or *Next* if you chose to create a custom launcher icon; this will take you to a customization page, after which you will be forced to finish).

First, we need to set up our build path. Surprisingly enough, it's much easier than the base project, though it is done a little differently.

At the time of this guide's writing, the latest release of the ADT/Eclipse plugin creates a `libs` folder within your project structure. This special folder automatically includes all of its contents on the build path.

Normally, you would drop the JAR files directly into this folder. However, this is undesirable as future releases/builds of JME3 would require you to re-copy all of the JAR files. Instead, we will simply link them.

For each of the following, right click the `libs` folder within your Android project and go to *New → File*, click *Advanced* », check *Link to file in the file system*, click *Browse...*, navigate to the JME3 `libs` folder (as identified in the *Setting up JME3* section above), double click the listed JAR file, and then click *Finish*:

- jME3-android.jar
- jME3-core.jar

There will be many libraries in the previously identified `libs` folder that aren't initially used. Certain aspects of JME3 (for instance, collision detection) will require the appropriate JAR if your code requires that functionality. However, it is highly recommended you do NOT include every JAR in the `libs` folder due to the fact that exporting them all when they are not needed will result in a huge jar file (~60MB).

As well, repeat the above step for your compiled assets JAR (see *Packaging under Assets*).

Now that the core JME3 libraries have been added, we'll need to include our base project's code. To do this, right click on the Android project and go to *Build Path → Configure Build Path*, select the *Projects* tab, click *Add*, and select the base project (in our case, `JMEclipse-Base` ).

Lastly, select the *Order and Export* tab. Ensure that your base project (i.e. `JMEclipse-Base` ), *Android Private Libraries*, *Android Dependencies*, and optionally *Google APIs* (if you have that target enabled) are checked. This step is important, or your project's libraries/assets will NOT be exported into the end APK.

Click *OK*, and your project's build path will be set up.

The next step is to create the application's activity and edit `AndroidManifest.xml` to configure the project to actually use our JME3 project.

First, right click on the Android project and go to *New → Class*, entering the following information and hitting *Finish*:

- **Package:** your application package (it's best to use the package specified in the project creation dialog; for this guide, we'll re-use `com.jme3.test` )
- **Name:** the activity class' name (i.e. `JMEclipseActivity` )
- **Superclass:** `com.jme3.app.AndroidHarness`

This will create a new activity class. In the resulting file, create a default constructor and add the following code:

```
public JMEclipseActivity()
{
 // Set the application class to run
 appClass = "com.jme3.test.MyGame";

 // Try ConfigType.FASTEST; or ConfigType.LEGACY if you have problems
 eglConfigType = ConfigType.BEST;

 // Exit Dialog title & message
 exitDialogTitle = "Quit game?";
 exitDialogMessage = "Do you really want to quit the game?";

 // Choose screen orientation
 screenOrientation = ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE;

 // Invert the MouseEvents X (default = true)
 mouseEventsInvertX = true;

 // Invert the MouseEvents Y (default = true)
 mouseEventsInvertY = true;
}
```

Pay close attention to the values above; `appClass` MUST be the full package + class name of the class in your base project that extends `SimpleApplication` (for advanced users, this is actually a subclass of `com.jme3.app.Application`).

## Running

Running your Android project is just like [running any other Android project](#). Assuming you've set up your build path correctly as instructed above, your application should deploy to any device/emulator and run as expected.

</div>

## Packaging / Deploying

Packaging your Android project is too vast to entirely cover in this guide. As this step is different for each project, I will simply link to this guide to [signing and exporting your APK](#) as it outlines the most common steps to exporting your Android application to be uploaded directly to Google Play (fmlly. App Market).

## Native (JNI + NDK) Project

### Prerequisites:

- JME3 Base Project created (as described above)
- [Eclipse CDT plugin](#) installed
- At least one configured toolchain for compiling on desktop platforms (Cygwin/MinGW/MSVC/GCC/etc.)
- Familiarity with JNI and how native libraries are included in a Java application's architecture (this section will assume you do)
- JDK for Java 6 or above

### If additionally building for Android:

- Android SDK downloaded and the Android 8 (2.2) target installed (higher [API](#) versions work too but may limit compatibility when deploying)
- Android NDK downloaded
- Optional: [NDK Eclipse plugin](#) installed (although I haven't seen a real need for it quite yet - it's mainly for building/launching native Activities)

It should be mentioned that this section will not go into C/C++ best practices, sample code, or any details about the inner workings of JNI; instead, this section simply shows you how to set up the environment/configurations to create a near-seamless build environment for including your native code in both your base project as well as your Android project (if you've created one).

Building JNI is actually quite straightforward (assuming you know how the C/C++ build process works). Even for Android, the NDK provides a slick system for building/including your compiled binaries in your project.

First, create a new C/C++ project by going to *File* → *New* → *Project...* → *C/C++* → *C++ Project* and clicking *Next*, giving it a name (we'll use `JMEclipse-Native`), expanding *Shared Library* and selecting *Empty Project*, then selecting a toolchain (select the most appropriate for compiling on your immediate desktop/platform, even if you plan on compiling for Android). Click *Finish*.

Next, we need to configure our build settings. Right click the native project, click *Properties* and go to *C/C++ Build* → *Settings*. Select the *Build Artifact* tab, ensure the drop down menu says *Shared Library*, and change the *Artifact name* field to what you want to call your eventual JNI module.

What you call your artifact is VERY important to how Java loads your library. For instance, linux dynamic library (.SO) objects require that the library have the “lib” prefix. Keep this in mind when specifying an artifact name.

If you plan on building for Android, you must include an Android makefile in your project. In order for the build process to be as seamless as possible, this guide sets it up unlike most tutorials instruct.

To do this, simply create a new file in your native project called `Android.mk` and [set it up accordingly](#).

Even though the native project is not the Android project's directory, keep the `LOCAL_PATH` variable set to `my-dir`. This is important!

Next, we need to create a configuration for the Android target.

This step is only relevant/possible if you have the NDK Eclipse plugin installed. Otherwise, you will need to install awk/make (using Cygwin if on Windows) and build through the command line manually.

To do this, right click your native project, click *Properties*, and go to *C/C++ Build* → *Tool Chain Editor*. For the configuration, click *Manage Configurations...* and create new configuration(s) based on the current default configurations, putting “Android” in the name. Lastly, click *OK*.

For each of the Android configurations you just created, select them in the *Configuration* drop down menu, set *Current toolchain* to *Android GCC*, set *Current builder* to *Android Builder*, then click *Select Tools...*, remove everything from the right side and replace the last item with *Android GCC Compiler*. Click *Apply* and then *OK*.

After the toolchains have been set up, go to *C/C++ Build* and select the *Builder Settings* tab. For each of the Android configurations, select them in the *Configuration* drop down menu and change the *Build directory* to the Android project directory by clicking *Workspace...* and selecting the Android project (in our case it'd be `JMEclipse-Android` ).

The last step in the Android setup is to create a symlink called `jni` (case sensitive) inside your Android project root that points to the root of your native project:

- **Windows:** CMD prompt → `cd path\to\Android\project` → `mklink /J jni path\to\native\project`
- **Linux/Mac:** Terminal → `cd path/to/Android/project` → `ln -sv path/to/native/project jni`

What this has essentially done is created a separate project for C/C++ building while tricking the Android project into thinking the source files are located directly in the Android project itself. With this configuration scheme, you can easily build regular shared libraries for desktop platforms as well as build/install the Android libraries using the NDK without the need to have a copy of the source code inside of the `jni` folder of an Android project. By setting the workspace for the Android builder and creating a link, it executes the NDK builder inside your Android project while serving the source files as if they existed in the `jni` folder. Lastly, some additional include paths need to be added. Within the properties window, go to *C/C++ General → Paths and Symbols* and select the *Includes* tab.

Under *Languages*, select *GNU C++* (or the correct C++ equivalent) and then *Add....* Specify the absolute path to your JDK's `include` folder, check *Add to all configurations*, and hit *OK*.

For Windows platforms, repeat the above step for the `win32` directory within the JDK's `include` folder.

After hitting *OK*, you are now set to write your JNI code.

</div>

## Building

Building your native project is fairly straightforward.

First, right click the native project, go to *Build Configurations*, select the configuration you want to build, and then right click the native project again and select *Build Project*. Short of packaging, that's all there is to it.

</div>

## Packaging / Deploying

Packaging and deploying your native libraries is a two-sided topic. For Android, the NDK build script installs these for you, and they are packaged directly into the APK. For desktop applications, however, there are a multitude of ways to package your libraries - all of which are too vast to be included in this guide.

These libraries, however, are simply JNI libraries and should be loaded as such. A simple web search will explain how JNI works and how to load these libraries.

</div>

## title: jMonkeyEngine 3.0 Feature Overview

# jMonkeyEngine 3.0 Feature Overview

See also: [Feature comparison](#) and [requirements](#).

</div>

## Software Development Kit: jMonkeyEngine SDK

- [Creates jME3-ready Java projects](#)
  - Preconfigured classpath
  - Bundled with compatible JDK
  - [Bundled with Blender conversion tools and more](#)
  - [Asset Manager](#) for loading multi-media files and 3D models including asset name code completion
  - [Non-proprietary Ant build scripts](#)
  - jME3-ready Javadoc popups, [sample code projects](#), and code snippet palette
- [Full-featured Java and XML code editor](#)
- [Plugins](#)
  - [File Version Control](#)
  - [Debugger and Profiler \(optional\)](#)
  - [Converters and Importers for game assets \(3D models etc\)](#)
  - [3D Scene Viewer and Scene Composer](#)
  - [Material editor](#)
  - [Shader Node editor](#)
  - [Terrain generation, painting, and editing](#)
  - [Custom font creator](#)
  - [Support for custom Asset Packs with your models, textures, and more](#)
  - [Procedural texture creator \(NeoTexture\)](#)
  - [Level of Detail \(LOD\) generator](#)
  - ... and much more...
- [Deployment](#)
  - Generates desktop executables for Win, Mac, Linux

- Generates mobile executables for [Android](#), iOS support is in the works.
- Generates JNLP WebStart and Java Browser Applets
- [Based on the NetBeans Platform](#)
  - Supports all NetBeans IDE plugins

&lt;/div&gt;

## Physics

- [JBullet physics binding](#)
  - [Physical characters](#)
  - [Physical joints and hinges](#)
  - [Ray-cast vehicle](#)
  - [Ragdoll physics](#)
- [Multi-threaded physics](#)
- [Mesh-accurate collision shapes](#)

&lt;/div&gt;

## Supported Formats

- Models: Blender (.blend)
- Models: Ogre3D model (.mesh.xml, .skeleton.xml, .material), Ogre3D dotScene (.scene)
- Models: Wavefront (.OBJ, .MTL)
- Models: Collada
- Models: 3DS
- Textures: .DDS, .HDR, .PFM, .TGA, .JPG, .PNG, .GIF.
- Font: BMFont fonts (.FNT)
- Audio: Waveform (.WAV), Ogg/Vorbis (.OGG)
- jME3 binary files (models and scenes): .j3o
- jME3 materials: .j3m
- jME3 material definitions: .j3md
- jME3 filter post processors: .j3f

&lt;/div&gt;

## Shaders

- GLSL support
- Shader libraries
- Shader permutations
- [Shader Nodes](#)

</div>

## Material Lighting

- Per-pixel lighting
- Multi-pass lighting
- Phong Lighting
  - Diffuse Map
  - Alpha Map
  - Glow Map
  - Specular Map
  - Normal Map, Parallax Map (a.k.a. bump mapping)
- Tangent shading
- Reflection

</div>

## Material Textures

- Texturing
  - material colors (ambient, diffuse, specular/shininess, glow),
  - color map, light map,
  - transparency, translucency, alpha map, alpha testing with falloff threshold,
  - sphere map, cube map,
  - texture scale,
  - wireframe
  - color ramp texture
- Multi-texturing through shaders
- UV textures
- Splat textures, Phong lit or unshaded, supports diffuse and normal maps
- [Texture Atlas, handling of packed Textures](#)

</div>

# Asset System

- Asset importing
  - Animation
  - Meshes
  - Textures
  - Scenes
  - Materials
  - Shaders
- Multi-threaded asset loading via HTTP
- Loading scenes from .ZIP files
- Sharable AssetPacks

</div>

# Special Effects

- Particles: Smoke, fire, explosions, etc
- Post processing / 2D Filter Effects
  - Reflective Water
  - Shadow mapping
  - High Dynamic Range rendering
  - Screen Space Ambient Occlusion
  - Light Scattering
  - Cartoon Effect
  - Fog
  - Bloom
  - Depth of Field Blur

</div>

# Terrain

- Geomipmapped hightmap terrain
- Import Ogre3D dotScene format
- SkyBox and SkyDome
- Terrain lighting

</div>

## GUI / HUD

- Orthogonal (Billboard) node
- Nifty GUI integration

</div>

## Miscellaneous

- Application States and Controls to implement game logic
- Cinematics and motion paths
- Camera System
  - Normal or parallel view
  - Multiple views
- Swing canvas (e.g. for Applets)
- Input handling
  - Mouse, keyboard, joystick
  - Combo moves

</div>

## Networking

- SpiderMonkey API

</div>

# title: General System Requirements for Applications Built with jMonkeyEngine



**Please expand this list!** I've copied this page from the original jME2 hardware compatibility list and will work on getting this up to date for jME3 -Skye

This page lists the compatibility of various different graphic cards and the latest version of jME3

To fill in a new entry add the name of your graphics card in the first column (Card Name), which driver version the card is running on in the second column (Driver Version). Then put a 'Y' for yes and a 'N' for no in the next three columns as to whether your card supports the given feature.

- Basic Features should be considered things like rendering primitive shapes and models fully textured and lit.
- Advanced Features should be considered things like Render to Texture and Cloth effects.
- Shader Features should be anything that uses shaders such as Bloom and Motion Blur.

## General System Requirements for Applications Built with jMonkeyEngine

Please see [requirements](#)

For a detailed list of Graphic Card/Driver combinations and which OpenGL features they support see <http://www.delphi3d.net/hardware/index.php>  
[http://www.opengl.org/wiki/Hardware\\_specifics](http://www.opengl.org/wiki/Hardware_specifics)

</div>

### ATI/AMD

Card Name	Driver Version	Basic Features?	Advanced Features (No Shader)?	Shader Features?	Notes
Radeon HD 4850 512MB	11.8 running on Linux Mint 12 x64 11.12 running on Windows 7 Professional x64	Y	Y	Y	All tests works fine.
Radeon HD 5850 1GB	11.12 running on Windows 7 Professional x64	Y	Y	Y	All tests works fine.
Radeon HD 6900 2GB	11.5 running on Windows 7 x64	Y	Y	Y	All tests works fine.

&lt;/div&gt;

## Intel

Card Name	Driver Version	Basic Features?	Advanced Features (No Shader)?	Shader Features?	Notes
Intel GMA 950	Mac OS X 10.5	Y	Y	N	Works in OpenGL1 compatibility mode.

&lt;/div&gt;

## NVidia

<b>Card Name</b>	<b>Driver Version</b>	<b>Basic Features?</b>	<b>Advanced Features (No Shader)?</b>	<b>Shader Features?</b>	<b>Notes</b>
NVidia GeForce GT330M 512M	Mac OS X 10.7.2	Y	Y	Y	All tests works fine.
NVidia GeForce GT540M 1024M	Windows 7 280.26	Y	Y	Y	All tests works fine.
NVidia GeForce GTX480 1536M	Windows 7 285.62	Y	Y	Y	All tests works fine.

&lt;/div&gt;

## SiS

<b>Card Name</b>	<b>Driver Version</b>	<b>Basic Features?</b>	<b>Advanced Features (No Shader)?</b>	<b>Shader Features?</b>	<b>Notes</b>

&lt;/div&gt;

## **title: iOS Deployment**

### **iOS Deployment**

To use iOS deployment you need a computer running MacOSX and a version of Xcode 4.0+ installed. To deploy to a device or the Apple App Store, you need an Apple developer account.

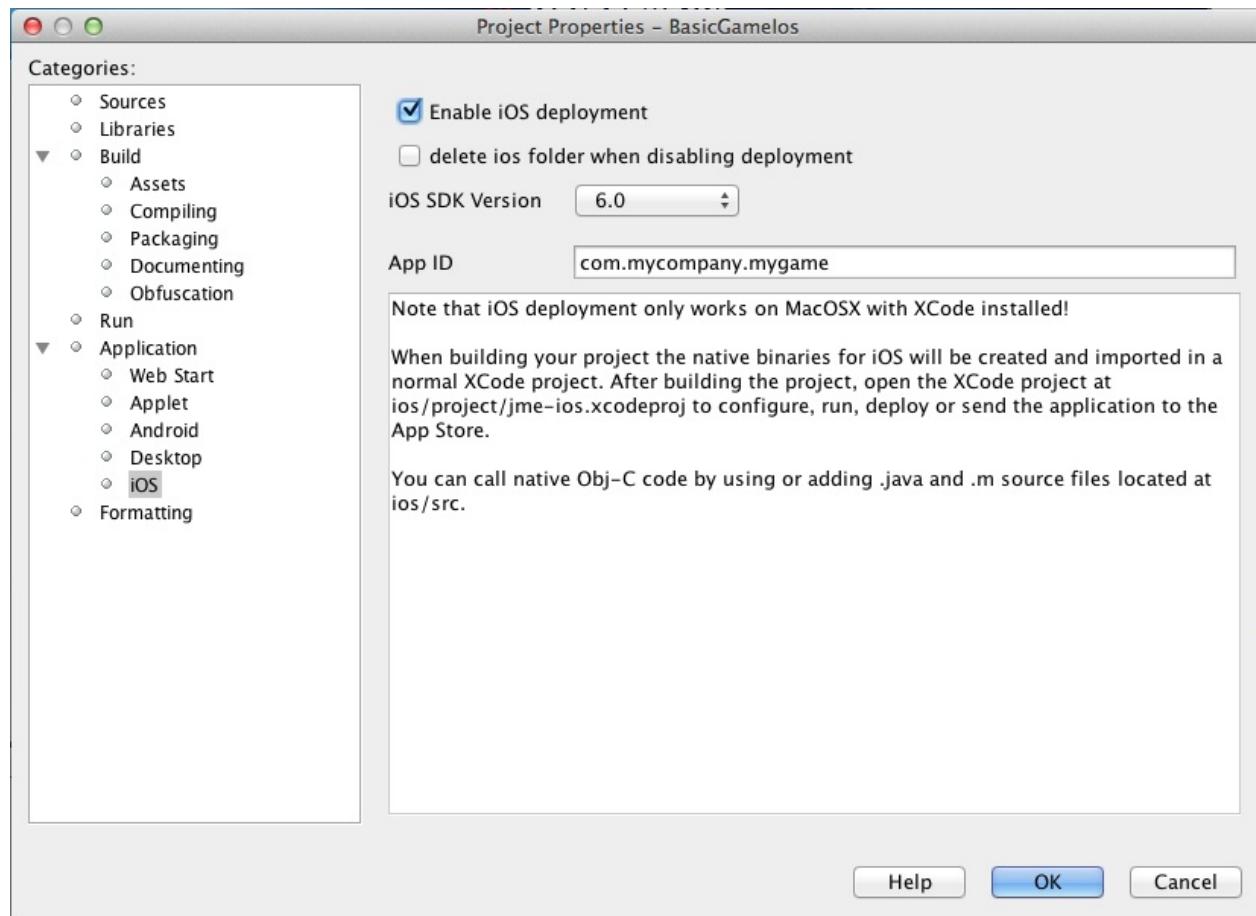
Note that at the moment iOS deployment is in alpha state.

iOS deployment works via cross-compilation to native iOS ARM code, there is no virtual machine running on the device. The Avian JVM supports this feature while maintaining general compatibility to OpenJDK and JNI for native access. The minimum compatible iOS deployment target is 4.3.

To install the iOS deployment plugin, go to Tools→Plugins and under “Available plugins” select the “iOS Support” plugin.

### **Enabling iOS deployment**

To enable iOS deployment, go to the project settings and under “Application→iOS” select the “Enable iOS deployment” checkbox, adapt the application ID and then press OK.



After enabling deployment, a new `ios` directory is created in the project root that contains a `project` and a `src` folder. The `ios/project` folder contains an Xcode project that you will use to build and run the final iOS application for both iPhone and iOS. The `ios/src` folder contains java and native source files for bridging iOS and native code, you can add `.java` and `.m` files with your own iOS code here.

When you enable iOS deployment for the first time or any time that the Avian library and OpenJDK is updated, they will be extracted to your SDK settings folder, wait until it has been extracted before building an iOS-enabled project.

## Building the iOS binaries

The iOS binaries are automatically built when you have iOS deployment enabled and build your project in the jME3 SDK.

When the iOS binaries are built, all needed classes, including a complete copy of the OpenJDK7 classes are run through a proguard process that strips out the unnecessary classes for the project and optimizes the code for the platform. This happens without changing the naming structure so that reflection etc. still works. If necessary, adapt the proguard options in the `ios` properties file.

After the iOS classpath has been created the avian compiler is used to create a native .o file from the classpath for both arm (device) and i386 (simulator). Furthermore the other needed avian .o files are extracted and a library list is compiled which is referenced in the Xcode project.

If an error occurs about jni.h not being found, either install the SDK for 10.9 in XCode or set the header search path in the XCode project settings, in the default project thats

```
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX
10.9.sdk/System/Library/Frameworks/JavaVM.framework/Headers/
```

```
</div>
```

## Running and deploying the application

To run the application, open the Xcode project under `ios/project` in Xcode and press the run button. You can make changes to the UI and native invocation classes in the Xcode project as well. From here you can also deploy the application to your devices or the App Store.

Note that you should also adapt the project settings like application name and registration package in Xcode before deploying the final application.

```
</div>
```

## Creating native and java code for iOS

To bridge between native and java code, JNI is used like in a normal java application. The `ios/src` folder is for Java and C/Obj-C source files that are specific to your iOS application. In these java files you have access to the full project classpath as well as the iOS-specific jME3 classes.

The `JmeAppHarness.java` class is initialized and called from native code through the default project and you can extend it to perform other native operations. It has a simple native popup method. The `JmeAppHarness.m` file contains the native method needed for that popup.

Effectively native code can reside in both the Xcode project and in the `ios/src` folder. To keep the dependencies clean and make code reusable you should try to put generic native code that does not depend on the Xcode project in the `ios/src` folder. You can also mix and match ARC and non-ARC code through this by converting the main project to use ARC and putting code with manual memory management in the `ios/src` folder.

Java code for iOS should be in the `ios/src` folder as well for clean separation, its also the only place where they will be compiled with a reference to the iOS specific jME classes. For information on how to connect your application code and device specific code, see the [notes](#)

in the android deployment documentation.

[documentation](#), [iOS](#), [Mac](#), [MacOS](#), [deployment](#), [platform](#)

</div>

# **title: jMonkeyEngine 3 -- Source Structure**

## **jMonkeyEngine 3 -- Source Structure**

An overview of the source structure of the JME3 project. In order to support both Desktop and Android Java platforms, it was necessary to split the source code into several parts. This wiki page describes the packages and their purpose. Status: Up-to-date for JME3 beta.

### **Structure of src directory**

You can build jME using the included build.xml script: `ant clean; ant jar; ant run` When building the sources in a project created with another IDE, include every folder under `src` in the project as its own separate source root.

### **Core**

<b>Source Package</b>	<b>Description</b>
src/core	The main package. Must always be included, as all other packages depend on it.
src/core-effects	Core effects like Water, PSSM etc.
src/core-data	Basic material definitions, shaders and fonts that are needed by most jME3 applications.
src/core-plugins	Important asset plugins, such as .j3o model loader, .obj loader, font loader, basic image loaders.
src/desktop	Must be included if deploying on desktop, applet or web start. <small>Android</small>
src/android	Must be included if deploying on the Android platform. <small>Desktop</small>
src/lwjgl	LWJGL OpenGL display implementation. <small>Android</small>

### **Physics**

Source Package	Description
src/jbullet	Game Physics Engine, based on the jBullet framework. <a href="#">Bullet</a>
src/bullet	Game Physics Engine, based on the native Bullet framework. <a href="#">jBullet</a>
src/bullet-common	Classes common between native and java Bullet implementations.
src/bullet-native	Native Bullet implementation C++ classes. <a href="#">jBullet</a>

Note: Only one of the physics libraries can be used at a time as they replace each other.

## Plugins and Extra packages

Source Package	Description
src/ogre	Ogre3D model and scene loader. Supports skeletal and vertex animation, scene loading, and materials.
src/xml	Provides an XML im/exporter.
src/jogg	OGG/Vorbis loader to play .ogg sound files.
src/niftygui	Support for custom Graphical User Interfaces.
src/blender	Blender model importer
src/networking	SpiderMonkey networking package
src/terrain	Terrain generation tools

## Tests, Games and Tools

Source Package	Description
src/test	Small sample Applications that demo individual jME3 features. ← jme3_test-data.jar
src/test-data	Data assets (jme3_test-data.jar) required by jme3_test samples.
src/tools	Tools and programs that help you use jme3.

## Structure of lib directory

JME3 depends on the following JARs and native libraries in the `lib` directory. The JAR libraries must be on the classpath.

The jME3-\*natives\*.jar bundles contain the native libraries, those are necessary `.dll`, `.jnilib`, `lib*.so` files. You do not need to manually include native libraries on the `java.library.path!` jME3 handles the extraction of natives automatically via the JAR bundles.

- lib/android:
  - android.jar
- lib/bullet:
  - android, jME3-bullet-natives-android.jar, jME3-bullet-natives.jar, jarcontent (natives)
- lib/jbullet:
  - asm-all.jar, jbullet.jar, stack-alloc.jar, vecmath.jar
- lib/jogg:
  - j-ogg-oggd.jar, j-ogg-vorbisd.jar
- lib/lwjgl:
  - jME3-lwjgl-natives.jar, jinput.jar,.lwjgl.jar
- lib/niftygui:
  - nifty.jar, nifty-javadoc.jar, xmlpull-xpp3.jar, eventbus.jar
  - nifty-default-controls-javadoc.jar, nifty-default-controls.jar,
  - nifty-examples.jar, nifty-examples-javadoc.jar, nifty-style-black.jar

</div>

## Structure of jMonkeyEngine3 JARs

After the build is complete (in the `dist` directory), you see that the jMonkeyEngine library is split up over several JAR files. This allows for better separation of the parts for different operating systems, projects etc.

JAR file	Purpose	External Dependence
dist/lib/jME3-core.jar	Platform-independent core libraries (math, animation, scenegraph, Wavefront OBJ model support, etc)	None
dist/lib/jME3-effects.jar	Core jME3 effects (Water, SSAO etc)	None
dist/lib/jME3-desktop.jar	Desktop PC only jME3 libraries	None
dist/lib/jME3-plugins.jar	Basic import plugins (OgreXML models and j3o XML)	None
dist/lib/jME3-blender.jar	Blender model import plugin (Desktop only)	None
dist/lib/jME3-networking.jar	“Spidermonkey” networking library	None
dist/lib/jME3-jogg.jar	J-OGG audio plugin	j-ogg-vorbisd.jar, j-ogg-oggd.jar
dist/lib/jME3-terrain.jar	Terrain system	None
dist/lib/jME3-jbullet.jar	jBullet physics	jbullet.jar, vecmath.jar, stack-alloc.jar, asm-all-3.1.jar
dist/lib/jME3-bullet.jar	Bullet physics (only jBullet <b>or</b> Bullet can be used)	jME3-bullet-natives.jar
dist/lib/jME3-niftygui.jar	NiftyGUI support	nifty.jar, nifty-default-controls.jar, eventbus.jar, xmlpull-xpp3.jar
dist/lib/jME3-lwjgl.jar	LWJGL Desktop Renderer	lwjgl.jar, jME3-lwjgl-natives.jar, jinput.jar
dist/lib/jME3-android.jar	Android Renderer	Android system

Optional:

- nifty-examples.jar
- jME3-testdata.jar
- nifty-style-black.jar (default nifty style)

## API Structure

For details see the <http://jmonkeyengine.org/javadoc/>.

</div>

## Data File Types

Path	File types	purpose
/Common/MatDefs/*/	.glslib	Standard ShaderLibs
/Common/MatDefs/*/	.j3md	Standard Material Definitions
/Common/Materials/*/	.j3m	Standard Material
/Interface/Fonts/	.fnt + .png	Standard Fonts

See also supported [File Types](#).

</div>

# title: JME3 Documentation: Materials

## JME3 Documentation: Materials

Work in progress. :)

Default Definition	Usage
Common/MatDefs/Misc/SimpleTextured.j3md	Textured: Use with mat.setTexture() and TextureKey
Common/MatDefs/Misc/SolidColor.j3md	Colored: Use with mat.setColor() and RGBAColor
Common/MatDefs/Misc/Sky.j3md	A solid skyblue. Or use with custom texture.
Common/MatDefs/Misc/WireColor.j3md	Transparent, show only outline
Common/MatDefs/Misc/Particle.j3md	
Common/MatDefs/Misc>ShowNormals.j3md	Shows a color gradient calculated from surface normals.
Common/MatDefs/Misc/VertexColor.j3md	
Common/MatDefs/Gui/Gui.j3md	
Common/MatDefs/Hdr/LogLum.j3md	
Common/MatDefs/Hdr/ToneMap.j3md	
Common/MatDefs/Light/Lighting.j3md	For texture lighting effects like Bump- and NormalMaps
Common/MatDefs/Light/Reflection.j3md	
Common/MatDefs/Shadow/PostShadow.j3md	
Common/MatDefs/Shadow/PreShadow.j3md	

---

## **title: jMonkeyEngine3 Math for Dummies**

### **jMonkeyEngine3 Math for Dummies**



- You can open this presentation in [Fullscreen Mode](#).
- When using [Safari](#) as a browser the presentation is animated.

</div>

## **title:**

### **jME3 Math Video Tutorials**

- [Video: Trigonometry](#)
- [Video: Vector Math](#)
- [Video: Dot Product](#)
- [Video: Spatial.lookAt\(\)](#)
- [Video: Camera](#)
- [Video: Rotations \(Part 1\)](#)
- [Video: Rotations \(Part 2\)](#)
- [Video: Cross Product](#)
- [Video: Rays](#)
- [Video: Reflections \(Part 1\)](#)
- [Video: Reflections \(Part 2\)](#)
- [Video: Reflections \(Part 3\)](#)
- [Video: Line of Sight](#)

## **title: Introduction to Mathematical Functionality**

# **Introduction to Mathematical Functionality**

It's a fact of life, math is hard. Unfortunately, 3D graphics require a fair bit of knowledge about the subject. Fortunately, jME is able to hide the majority of the details away from the user. Vectors are the fundamental type in the 3D environment, and it is used extensively. Matrices are also a basic necessity of 3D for representing linear systems. [Quaternions](#) are perhaps the most powerful and complicated of the basic types and are used for rotation in jME.

I'll discuss how these are used in the system for the core functionality. Including Transforming, Visibility Determination, Collision Detection, and the Coordinate System. Note, that these are low level details. Further chapters will discuss how to use these various systems from a high level perspective.

To get a visual introduction to math in jME3 for the absolute beginner, check out our [Math for Dummies](#) introduction class.

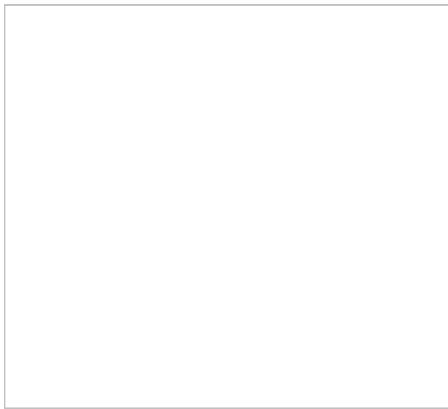
## **Coordinate System**

### **Definition**

A *coordinate system* consists of an origin (single point in space) and three coordinate axes that are each unit length and mutually perpendicular. The axes can be written as the column of a Matrix,  $R = [U_1|U_2|U_3]$ . In fact, this is exactly how CameraNode works. The coordinate system defined by Camera is stored in a Matrix.

jME uses a Right-Handed coordinate system (as OpenGL does).

The definition of a coordinate system is defined in jME by the properties sent to Camera. There are no error checks to insure that: 1) the coordinate system is right-handed and 2) The axes are mutually perpendicular. Therefore, if the user sets the axes incorrectly, they are going to experience very odd rendering artifacts (random culling, etc).



## Homogenous coordinates

Homogenous coordinates have an additional  $W$  value tacked on to the end. The  $XYZ$  values are to be divided by  $W$  to give the true coordinates.

This has several advantages, one technical, some relevant to application programmers:

Technically, it simplifies some formulae used inside the vector math. For example, some operations need to apply the same factor to the  $XYZ$  coordinates. Chain multiple operations of that kind (and vector math tends to do that), and you can save a lot of multiplications by simply keeping the scaling factor around and doing the multiplication to  $XYZ$  at the end of the pipeline, in the 3D card (which does accept homogenous coordinates). It also simplifies some formulae, in particular anything that is related to rotations.

For application programmers, this means you can express infinitely long vectors that still have a direction - these tend to be used in lighting. Just use a  $W$  value of 0.0.

## Transformations

Transformations define an operation that converts points from one coordinate system to another. This includes translation, rotation and scaling. In jME, local transforms are used to represent the positioning of objects relative to a parent coordinate system. While, world transforms are used to represent the positioning of objects in a global coordinate system.

## Visibility Determination

Visibility Determination concerns itself with minimizing the amount of data that is sent to the graphics card for rendering. Specifically, we do not want to send data that will not be seen. Data not sent to the graphics card is said to be culled. The primary focus of this section is Frustum Culling based on the Camera's view frustum. In essence, this frustum creates six standard view planes. The BoundingVolume of an object is tested against the frustum planes

to determine if it is contained in the frustum. If at any point the object's bounding is outside of the plane, it is tossed out and no longer processed for rendering. This also includes any children that it managed, allowing fast culling of large sections of the scene.

# Fundamental Types

## ColorRGBA

### Definition

ColorRGBA defines a color value in the jME library. The color value is made of three components, red, green and blue. A fourth component defines the alpha value (transparent) of the color. Every value is set between [0, 1]. Anything less than 0 will be clamped to 0 and anything greater than 1 will be clamped to 1.

**Note:** If you would like to “convert” an ordinary RGB value (0-255) to the format used here (0-1), simply multiply it with: 1/255.

### jME Class

ColorRGBA defines a few static color values for ease of use. That is, rather than:

```
ColorRGBA red = new ColorRGBA(1, 0, 0, 1);
object.setSomeColor(red);
```

you can simply say:

```
object.setSomeColor(ColorRGBA.red)
```

ColorRGBA will also handle interpolation between two colors. Given a second color and a value between 0 and 1, the owning ColorRGBA object will have its color values altered to this new interpolated color.

# Matrix

See [Matrix3f Javadoc](#)  
and [Matrix4f Javadoc](#)

## Definition

A Matrix is typically used as a *linear transformation* to map vectors to vectors. That is:  $Y = MX$  where X is a Vector and M is a Matrix applying any or all transformations (scale, [rotate](#), [translate](#)).

There are a few special matrices:

*zero matrix* is the Matrix with all zero entries.

0	0	0
0	0	0
0	0	0

The *Identity Matrix* is the matrix with 1 on the diagonal entries and 0 for all other entries.

1	0	0
0	1	0
0	0	1

A Matrix is *invertible* if there is a matrix  $M^{-1}$  where  $MM^{-1} = M^{-1}M = I$ .

The *transpose* of a matrix  $M = [m_{ij}]$  is  $M^T = [m_{ji}]$ . This causes the rows of  $M$  to become the columns of  $M^T$ .

1	1	1		1	2	3
2	2	2	$\Rightarrow$	1	2	3
3	3	3		1	2	3

A Matrix is symmetric if  $M = M^T$ .

X	A	B
A	X	C
B	C	X

Where X, A, B, and C equal numbers

jME includes two types of Matrix classes: `Matrix3f` and `Matrix4f`. `Matrix3f` is a  $3 \times 3$  matrix and is the most commonly used (able to handle scaling and rotating), while `Matrix4f` is a  $4 \times 4$  matrix that can also handle translation.

## Transformations

Multiplying a vector with a Matrix allows the vector to be transformed. Either rotating, scaling or translating that vector.

## Scaling

If a *diagonal Matrix*, defined by  $D = [d_{ij}]$  and  $d_{ij} = 0$  for  $i \neq j$ , has all positive entries it is a *scaling matrix*. If  $d_i$  is greater than 1 then the resulting vector will grow, while if  $d_i$  is less than 1 it will shrink.

## Rotation

A *rotation matrix* requires that the transpose and inverse are the same matrix ( $R^{-1} = R^T$ ).

The *rotation matrix*  $R$  can then be calculated as:  $R = I + (\sin(\text{angle})) S + (1 - \cos(\text{angle}))S^2$  where  $S$  is:

0	$u_2$	$-u_1$
$-u_2$	0	$u_0$
$u_1$	$-u_0$	0

## Translation

Translation requires a  $4 \times 4$  matrix, where the vector  $(x,y,z)$  is mapped to  $(x,y,z,1)$  for multiplication. The *Translation Matrix* is then defined as:

M	T
$S^T$	1

where M is the  $3 \times 3$  matrix (containing any rotation/scale information), T is the translation vector and  $S^T$  is the transpose Vector of T. 1 is just a constant.

## jME Class

Both Matrix3f and Matrix4f store their values as floats and are publicly available as (m00, m01, m02, ..., mNN) where N is either 2 or 3.

Most methods are straight forward, and I will leave documentation to the Javadoc.

## Vector

See [Vector3f Javadoc](#)  
and [Vector2f Javadoc](#)

## Definition

Vectors are used to represent a multitude of things in jME, points in space, vertices in a triangle mesh, normals, etc. These classes (Vector3f in particular) are probably the most used class in jME.

A Vector is defined by an n-tuple of real numbers.  $\mathbf{V} = \langle V_1, V_2, \dots, V_n \rangle$ .

We have two Vectors (2f and 3f) meaning we have tuples of 2 float values or 3 float values.

## Operations

### Multiplication by Scalar

A Vector can be multiplied by a scalar value to produce a second Vector with the same proportions as the first.  $a\mathbf{V} = \mathbf{Va} = \langle aV_1, aV_2, \dots, aV_n \rangle$

### Addition and Subtraction

Adding or subtracting two Vectors occurs component-wise. That is the first component is added (subtracted) with the first component of the second Vector and so on.

$$\mathbf{P} + \mathbf{Q} = \langle P_1 + Q_1, P_2 + Q_2, \dots, P_n + Q_n \rangle$$

### Magnitude

The *magnitude* defines the length of a Vector. A Vector of magnitude 1 is *unit length*.

For example, if  $\mathbf{V} = (x, y, z)$ , the magnitude is the square root of  $(x^2 + y^2 + z^2)$ .

A Vector can be *normalized* or made *unit length* by multiplying the Vector by (1/magnitude).

### Dot Products

The dot product of two vectors is defined as:  $\mathbf{P} \cdot \mathbf{Q} = P_x Q_x + P_y Q_y + P_z Q_z$

Using the dot product allows us to determine how closely two Vectors are pointing to the same point. If the dot product is negative they are facing in relatively opposite directions, while positive tells us they are pointing in the relative same direction.

If the dot product is 0 then the two Vectors are *orthogonal* or 90 degrees off.

## Cross Product

The Cross Product of two Vectors returns a third Vector that is perpendicular to the two Vectors. This is very useful for calculating surface normals.

$$\mathbf{P} \times \mathbf{Q} = \langle P_y Q_z - P_z Q_y, P_z Q_x - P_x Q_z, P_x Q_y - P_y Q_x \rangle$$

## jME Class

Vector3f and Vector2f store their values (x, y, z) and (x, y) respectively as floats. Most methods are straight forward, and I will leave documentation to the Javadoc.

## Quaternion

See [Quaternion Javadoc](#)

### Definition

Quaternions define a subset of a hypercomplex number system. Quaternions are defined by ( $i^2 = j^2 = k^2 = ijk = -1$ ). jME makes use of Quaternions because they allow for compact representations of rotations, or correspondingly, orientations, in 3D space. With only four float values, we can represent an object's orientation, where a rotation matrix would require nine. They also require fewer arithmetic operations for concatenation.

Additional benefits of the Quaternion is reducing the chance of [Gimbal Lock](#) and allowing for easily interpolation between two rotations (spherical linear interpolation or slerp).

While Quaternions are quite difficult to fully understand, there are an exceeding number of convenience methods to allow you to use them without having to understand the math behind it. Basically, these methods involve nothing more than setting the Quaternion's x,y,z,w values using other means of representing rotations. The Quaternion is then contained in the [Spatial](#) as its local rotation component.

Quaternion  $\mathbf{q}$  has the form

$$\mathbf{q} = \langle w, x, y, z \rangle = w + xi + yj + zk$$

or alternatively, it can be written as:

$\mathbf{q} = \mathbf{s} + \mathbf{v}$ , where  $\mathbf{s}$  represents the scalar part corresponding to the w-component of  $\mathbf{q}$ , and  $\mathbf{v}$  represents the vector part of the (x, y, z) components of  $\mathbf{q}$ .

Multiplication of Quaternions uses the distributive law and adheres to the following rules with multiplying the imaginary components (i, j, k):

```
i2 = j2 = k2 = -1
ij = -ji = k
jk = -kj = i
ki = -ik = j
```

However, Quaternion multiplication is *not* commutative, so we have to pay attention to order.

$$\mathbf{q}_1\mathbf{q}_2 = s_1s_2 - \mathbf{v}_1 \cdot \mathbf{v}_2 + s_1\mathbf{v}_2 + s_2\mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2$$

Quaternions also have conjugates where the conjugate of  $\mathbf{q}$  is  $(s - \mathbf{v})$

These basic operations allow us to convert various rotation representations to Quaternions.

## Angle Axis

You might wish to represent your rotations as Angle Axis pairs. That is, you define a axis of rotation and the angle with which to [rotate](#) about this axis. [Quaternion](#) defines a method `fromAngleAxis` (and `fromAngleNormalAxis`) to create a Quaternion from this pair. This is actually used quite a bit in jME demos to continually rotate objects. You can also obtain a Angle Axis rotation from an existing Quaternion using `toAngleAxis`.

## Example - Rotate a Spatial Using fromAngleAxis

```
//rotate about the Y-Axis by approximately 1 pi
Vector3f axis = Vector3f.UNIT_Y;
// UNIT_Y equals (0,1,0) and does not require to create a new object
float angle = 3.14f;
s.getLocalRotation().fromAngleAxis(angle, axis);
```

## Three Angles

You can also represent a rotation by defining three angles. The angles represent the rotation about the individual axes. Passing in a three-element array of floats defines the angles where the first element is X, second Y and third is Z. The method provided by Quaternion is `fromAngles` and can also fill an array using `toAngles`

## Example - Rotate a Spatial Using fromAngles

```
//rotate 1 radian on the x, 3 on the y and 0 on z
float[] angles = {1, 3, 0};
s.getLocalRotation().fromAngles(angles);
```

## Three Axes

If you have three axes that define your rotation, where the axes define the left axis, up axis and directional axis respectively) you can make use of `fromAxes` to generate the Quaternion. It should be noted that this will generate a new [Matrix](#) object that is then garbage collected, thus, this method should not be used if it will be called many times. Again, `toAxes` will populate a Vector3f array.

### Example - Rotate a Spatial Using fromAxes

```
//rotate a spatial to face up ~45 degrees
Vector3f[] axes = new Vector3f[3];
axes[0] = new Vector3f(-1, 0, 0); //left
axes[1] = new Vector3f(0, 0.5f, 0.5f); //up
axes[2] = new Vector3f(0, 0.5f, 0.5f); //dir

s.getLocalRotation().fromAxes(axes);
```

## Rotation Matrix

Commonly you might find yourself with a [Matrix](#) defining a rotation. In fact, it's very common to contain a rotation in a [Matrix](#) create a Quaternion, rotate the Quaternion, and then get the [Matrix](#) back. Quaternion contains a `fromRotationMatrix` method that will create the appropriate Quaternion based on the give [Matrix](#). The `toRotationMatrix` will populate a given [Matrix](#).

### Example - Rotate a Spatial Using a Rotation Matrix

```
Matrix3f mat = new Matrix3f();
mat.setColumn(0, new Vector3f(1,0,0));
mat.setColumn(1, new Vector3f(0,-1,0));
mat.setColumn(2, new Vector3f(0,0,1));

s.getLocalRotation().fromRotationMatrix(mat);
```

As you can see there are many ways to build a Quaternion. This allows you to work with rotations in a way that is conceptually easier to picture, but still build Quaternions for internal representation.

## Slerp

One of the biggest advantages to using Quaternions is allowing interpolation between two rotations. That is, if you have an initial Quaternion representing the original orientation of an object, and you have a final Quaternion representing the orientation you want the object to face, you can do this very smoothly with slerp. Simply supply the time, where time is [0, 1] and 0 is the initial rotation and 1 is the final rotation.

### Example - Use Slerp to Rotate Between two Quaternions

```
/*
You can interpolate rotations between two quaternions using spherical
interpolation (slerp).
*/
Quaternion Xroll45 = new Quaternion();
Xroll45.fromAngleAxis(45 * FastMath.DEG_TO_RAD, Vector3f.UNIT_X);
//
Quaternion Yroll45 = new Quaternion();
Yroll45.fromAngleAxis(45 * FastMath.DEG_TO_RAD, Vector3f.UNIT_Y);

//the rotation half - way between these two

Quaternion halfBetweenXroll45Yroll45 = new Quaternion();
halfBetweenXroll45Yroll45.slerp(Xroll45, Yroll45, 0.5f);
geom2.setLocalRotation(halfBetweenXroll45Yroll45);
```

## Multiplication

You can concatenate (add) rotations: This means you turn the object first around one axis, then around the other, in one step.

```
Quaternion myRotation = pitch90.mult(roll45); /* pitch and roll */
```

To rotate a Vector3f around its origin by the Quaternion amount, use the multLocal method of the Quaternion:

```
Quaternion myRotation = pitch90;
Vector3f myVector = new Vector3f(0,0,-1);
myRotation.multLocal(myVector);
```

# Utility Classes

Along with the base Math classes, jME provides a number of Math classes to make development easier (and, hopefully, faster). Most of these classes find uses throughout the jME system internally. They can also prove beneficial to users as well.

## Fast Math

See [FastMath Javadoc](#)

### Definition

FastMath provides a number of convenience methods, and where possible faster versions (although this can be at the sake of accuracy).

### Usage

FastMath provides a number of constants that can help with general math equations. One important attribute is `USE_FAST_TRIG` if you set this to true, a look-up table will be used for trig functions rather than Java's standard Math library. This provides significant speed increases, but might suffer from accuracy so care should be taken.

There are five major categories of functions that FastMath provides.

### Trig Functions

- cos and acos - provide [cosine](#) and [arc cosine](#) values (make use of the look-up table if `USE_FAST_TRIG` is true)
- sin and asin - provide [sine](#) and [arc sine](#) values (make use of the look-up table if `USE_FAST_TRIG` is true)
- tan and atan - provide [tangent](#) and [arc tangent](#) values

### Numerical Methods

- ceil - provides the ceiling (smallest value that is greater than or equal to a given value and an integer) of a value.
- floor - provides the floor (largest value that is less than or equal to a given value and an integer) of a value.
- exp - provides the [euler number](#) (e) raised to the provided value.
- sqr - provides the square of a value (i.e. value \* value).
- pow - provides the first given number raised to the second.
- isPowerOfTwo - provides a boolean if a value is a power of two or not (e.g. 32, 64, 4).
- abs - provides the [absolute value](#) of a given number.
- sign - provides the sign of a value (1 if positive, -1 if negative, 0 if 0).
- log - provides the [natural logarithm](#) of a value.
- sqrt - provides the [square root](#) of a value.
- invSqrt - provides the inverse [square root](#) of a value (1 / sqrt(value)).

## Linear Algebra

- LERP - calculate the [linear interpolation](#) of two points given a time between 0 and 1.
- determinant - calculates the [determinant](#) of a 4×4 matrix.

## Geometric Functions

- counterClockwise - given three points (defining a triangle), the winding is determined. 1 if counter-clockwise, -1 if clockwise and 0 if the points define a line.
- pointInsideTriangle - calculates if a point is inside a triangle.
- sphericalToCartesian - converts a point from [spherical coordinates](#) to [cartesian coordinates](#).
- cartesianToSpherical - converts a point from [cartesian coordinates](#) to [spherical coordinates](#).

## Misc.

- newRandomFloat - obtains a random float.

</div>

## Line

See [Line Javadoc](#)

</div>

## Definition

A line is a straight one-dimensional figure having no thickness and extending infinitely in both directions. A line is defined by two points **A** and **B** with the line passing through both.

</div>

## Usage

jME defines a Line class that is defined by an origin and direction. In reality, this Line class is typically used as a *line segment*. Where the line is finite and contained between these two points.

`random` provides a means of generate a random point that falls on the line between the origin and direction points.

</div>

## Example 1 - Find a Random Point on a Line

```
Line l = new Line(new Vector3f(0,1,0), new Vector3f(3,2,1));
Vector3f randomPoint = l.random();
```

</div>

## Plane

See [Plane Javadoc](#)

</div>

## Definition

A plane is defined by the equation  $\mathbf{N} \cdot (\mathbf{X} - \mathbf{X}_0) = 0$  where  $\mathbf{N} = (a, b, c)$  and passes through the point  $\mathbf{X}_0 = (x_0, y_0, z_0)$ .  $\mathbf{X}$  defines another point on this plane  $(x, y, z)$ .

$\mathbf{N} \cdot (\mathbf{X} - \mathbf{X}_0) = 0$  can be described as  $(\mathbf{N} \cdot \mathbf{X}) + (\mathbf{N} \cdot -\mathbf{X}_0) = 0$

or

$$(ax + by + cz) + (-ax_0 - by_0 - cz_0) = 0$$

$$\text{where } (-ax_0 - by_0 - cz_0) = d$$

Where  $d$  is the negative value of a point in the plane times the unit vector describing the orientation of the plane.

This gives us the general equation:  $(ax + by + cz + d = 0)$

</div>

## Usage in jME

jME defines the Plane as  $ax + by + cz = -d$ . Therefore, during creation of the plane, the normal of the plane ( $a,b,c$ ) and the constant  $d$  is supplied.

The most common usage of Plane is [Camera](#) frustum planes. Therefore, the primary purpose of Plane is to determine if a point is on the positive side, negative side, or intersecting a plane.

Plane defines the constants:

- `NEGATIVE_SIDE` - represents a point on the opposite side to which the normal points.
- `NO_SIDE` - represents a point that lays on the plane itself.
- `POSITIVE_SIDE` - represents a point on the side to which the normal points.

These values are returned on a call to `whichSide`.

</div>

## Example 1 - Determining if a Point is On the Positive Side of a Plane

```
Vector3f normal = new Vector3f(0,1,0);
float constant = new Vector3f(1,1,1).dot(normal);
Plane testPlane = new Plane(normal, constant);

int side = testPlane.whichSide(new Vector3f(2,1,0));

if(side == Plane.NO_SIDE) {
 System.out.println("This point lies on the plane");
}
```

</div>

## Example 2 - For the Layperson

Using the standard constructor `Plane(Vector3f normal, float constant)`, here is what you need to do to create a plane, and then use it to check which side of the plane a point is on.

```
package test;

import java.util.logging.Logger;

import com.jme.math.*;

/**
 * @author Nick Wiggill
 */

public class TestPlanes
{
 public static final Logger logger = Logger.getLogger(LevelGraphBu

 public static void main(String[] args) throws Exception
 {
 //***Outline.
 //This example shows how to construct a plane representation us
 //com.jme.math.Plane.

 //We will create a very simple, easily-imagined 3D plane. It wi
 //be perpendicular to the x axis (it's facing). It's "centre" (
 //such a thing exists in an infinite plane) will be positioned
 //unit along the positive x axis.

 //***Step 1.
 //The vector that represents the normal to the plane, in 3D spa
 //Imagine a vector coming out of the origin in this direction.
 //There is no displacement yet (see Step 2, below).
 Vector3f normal = new Vector3f(5f,0,0);

 //***Step 2.
 //This is our displacement vector. The plane remains facing in
 //direction we've specified using the normal above, but now we
 //are actually giving it a position other than the origin.
 //We will use this displacement to define the variable "constan
 //needed to construct the plane. (see step 3)
 Vector3f displacement = Vector3f.UNIT_X;
```

```

//or

//Vector3f displacement = new Vector3f(1f, 0, 0);

/**Step 3.
//Here we generate the constant needed to define any plane. This
//is semi-arcane, don't let it worry you. All you need to
//do is use this same formula every time.
float constant = displacement.dot(normal);

/**Step 4.
//Finally, construct the plane using the data you have assembled
Plane plane = new Plane(normal, constant);

/**Some tests.
logger.info("Plane info: "+plane.toString()); //trace our plane

Vector3f p1 = new Vector3f(1.1f,0,0); //beyond the plane (further)
Vector3f p2 = new Vector3f(0.9f,0,0); //before the plane (closer)
Vector3f p3 = new Vector3f(1f,0,0); //on the plane

logger.info("p1 position relative to plane is "+plane.whichSide(p1));
logger.info("p2 position relative to plane is "+plane.whichSide(p2));
logger.info("p3 position relative to plane is "+plane.whichSide(p3));
}
}

```

## Ray

See [Ray Javadoc](#)

</div>

### Definition

Ray defines a line that starts at a point **A** and continues in a direction through **B** into infinity.

This Ray is used extensively in jME for [Picking](#). A Ray is cast from a point in screen space into the scene. Intersections are found and returned. To create a ray supply the object with two points, where the first point is the origin.

```
</div>
```

## Example 1 - Create a Ray That Represents Where the Camera is Looking

```
Ray ray = new Ray(cam.getLocation(), cam.getDirection());
```

```
</div>
```

# Rectangle

See [Rectangle Javadoc](#)

```
</div>
```

## Definition

Rectangle defines a finite plane within three dimensional space that is specified via three points (A, B, C). These three points define a triangle with the forth point defining the rectangle ( (B + C) - A ).

```
</div>
```

## jME Usage

Rectangle is a straight forward data class that simply maintains values that defines a Rectangle in 3D space. One interesting use is the `random` method that will create a random point on the Rectangle. The [Particle System](#) makes use of this to define an area that generates [Particles](#).

```
</div>
```

## Example 1 : Define a Rectangle and Get a Point From It

```
Vector3f v1 = new Vector3f(1,0,0);
Vector3f v2 = new Vector3f(1,1,0);
Vector3f v3 = new Vector3f(0,1,0);
Rectangle r = new Rectangle(v1, v2, v3);
Vector3f point = r.random();
```

</div>

# Triangle

See [Triangle Javadoc](#)

</div>

## Definition

A triangle is a 3-sided polygon. Every triangle has three sides and three angles, some of which may be the same. If the triangle is a right triangle (one angle being 90 degrees), the side opposite the 90 degree angle is the hypotenuse, while the other two sides are the legs. All triangles are [convex](#) and [bicentric](#).

</div>

## Usage

jME's Triangle class is a simple data class. It contains three [Vector3f](#) objects that represent the three points of the triangle. These can be retrieved via the `get` method. The `get` method, obtains the point based on the index provided. Similarly, the values can be set via the `set` method.

</div>

## Example 1 - Creating a Triangle

```
//the three points that make up the triangle
Vector3f p1 = new Vector3f(0,1,0);
Vector3f p2 = new Vector3f(1,1,0);
Vector3f p3 = new Vector3f(0,1,1);
Triangle t = new Triangle(p1, p2, p3);
```

</div>

## Tips and Tricks

</div>

## How do I get height/width of a spatial?

Cast the spatial to com.jme3.bounding.BoundingBox to be able to use getExtent().

```
Vector3f extent = ((BoundingBox) spatial.getWorldBound()).getExtent
float x = ((BoundingBox)spatial.getWorldBound()).getXExtent();
float y = ((BoundingBox)spatial.getWorldBound()).getYExtent();
float z = ((BoundingBox)spatial.getWorldBound()).getZExtent();
```

</div>

## How do I position the center of a Geometry?

```
geo.center().move(pos);
```

</div>

### See Also

- [Rotate](#)
- [Quaternion](#)

</div>

# title: Matrix

## Matrix

See [Javadoc of Matrix3f](#)

and [Javadoc of Matrix4f](#)

### Definition

A Matrix is typically used as a *linear transformation* to map vectors to vectors. That is:  $\mathbf{Y} = \mathbf{MX}$  where  $\mathbf{X}$  is a Vector and  $\mathbf{M}$  is a Matrix applying any or all transformations (scale, rotate, translate).

There are a few special matrices:

*zero matrix* is the Matrix with all zero entries.

0	0	0
0	0	0
0	0	0

The *Identity Matrix* is the matrix with 1 on the diagonal entries and 0 for all other entries.

1	0	0
0	1	0
0	0	1

A Matrix is *invertible* if there is a matrix  $M^{-1}$  where  $MM^{-1} = M^{-1}M = I$ .

The *transpose* of a matrix  $M = [m_{ij}]$  is  $M^T = [m_{ji}]$ . This causes the rows of  $M$  to become the columns of  $M^T$ .

1	1	1		1	2	3
2	2	2	⇒	1	2	3
3	3	3		1	2	3

A Matrix is *symmetric* if  $M = M^T$ .

X	A	B
A	X	C
B	C	X

Where X, A, B, and C equal numbers

jME includes two types of Matrix classes: `Matrix3f` and `Matrix4f`. `Matrix3f` is a  $3 \times 3$  matrix and is the most commonly used (able to handle scaling and rotating), while `Matrix4f` is a  $4 \times 4$  matrix that can also handle translation.

## Transformations

Multiplying a [Vector](#) with a Matrix allows the [Vector](#) to be transformed. Either rotating, scaling or translating that [Vector](#).

## Scaling

If a *diagonal Matrix*, defined by  $D = [d_{ij}]$  and  $d_{ij} = 0$  for  $i \neq j$ , has all positive entries it is a *scaling matrix*. If  $d_i$  is greater than 1 then the resulting [Vector](#) will grow, while if  $d_i$  is less than 1 it will shrink.

## Rotation

A *rotation matrix* requires that the transpose and inverse are the same matrix ( $R^{-1} = R^T$ ). The *rotation matrix*  $R$  can then be calculated as:  $R = I + (\sin(\text{angle})) S + (1 - \cos(\text{angle})) S^2$  where  $S$  is:

0	$u_2$	$-u_1$
$-u_2$	0	$u_0$
$u_1$	$-u_0$	0

## Translation

Translation requires a  $4 \times 4$  matrix, where the [Vector](#)  $(x,y,z)$  is mapped to  $(x,y,z,1)$  for multiplication. The *Translation Matrix* is then defined as:

M	T
$S^T$	1

where M is the  $3 \times 3$  matrix (containing any rotation/scale information), T is the translation [Vector](#) and  $S^T$  is the transpose Vector of T. 1 is just a constant.

</div>

## jME Class

Both Matrix3f and Matrix4f store their values as floats and are publicly available as (m00, m01, m02, ..., mNN) where N is either 2 or 3.

Most methods are straight forward, and I will leave documentation to the Javadoc.

</div>

## title: jME3 with Maven

# jME3 with Maven

You can use jME3 with maven compatible build systems, the official maven repository for jME3 is at <http://updates.jmonkeyengine.org/maven/>

The group id for all jME3 libraries is com.jme3, the following artifacts are currently available (version 3.0.10):

- jme3-core - Core libraries needed for all jME3 projects
- jme3-effects - Effects libraries for water and other post filters
- jme3-networking - jME3 networking libraries (aka spidermonkey)
- jme3-plugins - Loader plugins for OgreXML and jME-XML
- jme3-jogg - Loader for jogg audio files
- jme3-terrain - Terrain generation API
- jme3-blender - Blender file loader, only works on desktop renderers
- jme3-jbullet - Physics support using jbullet (desktop only) **Only jme3-jbullet OR jme3-bullet can be used**
- jme3-bullet - Physics support using native bullet, needs jme3-bullet-natives or jme3-bullet-natives-android (alpha)
- jme3-bullet-natives - Native libraries needed for bullet (not jbullet) on desktop (alpha)
- jme3-bullet-natives-android - Native libraries needed for bullet (not jbullet) on android (alpha)
- jme3-niftygui - NiftyGUI support for jME3
- jme3-desktop - Parts of the jME3 API that are only compatible with desktop renderers, needed for image loading on desktop
- jme3-lwjgl - Desktop renderer for jME3
- jme3-android - Android renderer for jME3
- jme3-ios - iOS renderer for jME3

For a basic desktop application to work you need to import at least

- jme3-core
- jme3-desktop
- jme3-lwjgl

For a basic android application to work you need to import at least

- jme3-core
- jme3-android

</div>

## Gradle

If you happen to be using Gradle, you'll first need to add the repository, perhaps so it looks like this:

```
repositories {
 mavenCentral()
 maven {
 url 'http://updates.jmonkeyengine.org/maven'
 }
}
```

Next you'll need to add dependencies on all the JARs – here's what it looks like for all desktop-related JARs, selecting the latest patch version:

```
dependencies {
 compile 'com.jme3:jme3-core:3.0.+'
 compile 'com.jme3:jme3-effects:3.0.+'
 compile 'com.jme3:jme3-networking:3.0.+'
 compile 'com.jme3:jme3-plugins:3.0.+'
 compile 'com.jme3:jme3-jogg:3.0.+'
 compile 'com.jme3:jme3-terrain:3.0.+'
 compile 'com.jme3:jme3-blender:3.0.+'
 compile 'com.jme3:jme3-jbullet:3.0.+'
 compile 'com.jme3:jme3-niftygui:3.0.+'
 compile 'com.jme3:jme3-desktop:3.0.+'
 compile 'com.jme3:jme3-lwjgl:3.0.+'
}
```

If you'd rather factor out the “3.0” bit, you can also do this:

```
def jmonkeyengine_version = '3.0'

dependencies {
 compile "com.jme3:jme3-core:$jmonkeyengine_version.+"
 compile "com.jme3:jme3-effects:$jmonkeyengine_version.+"
 compile "com.jme3:jme3-networking:$jmonkeyengine_version.+"
 compile "com.jme3:jme3-plugins:$jmonkeyengine_version.+"
 compile "com.jme3:jme3-jogg:$jmonkeyengine_version.+"
 compile "com.jme3:jme3-terrain:$jmonkeyengine_version.+"
 compile "com.jme3:jme3-blender:$jmonkeyengine_version.+"
 compile "com.jme3:jme3-jbullet:$jmonkeyengine_version.+"
 compile "com.jme3:jme3-niftygui:$jmonkeyengine_version.+"
 compile "com.jme3:jme3-desktop:$jmonkeyengine_version.+"
 compile "com.jme3:jme3-lwjgl:$jmonkeyengine_version.+"
}
```

</div>

# title: Quaternion

## Quaternion

See [Javadoc](#)

### Definition

Quaternions define a subset of a hypercomplex number system. Quaternions are defined by ( $i^2 = j^2 = k^2 = ijk = -1$ ). jME makes use of Quaternions because they allow for compact representations of rotations, or correspondingly, orientations, in 3D space. With only four float values, we can represent an object's orientation, where a rotation matrix would require nine. They also require fewer arithmetic operations for concatenation.

Additional benefits of the Quaternion is reducing the chance of [Gimbal Lock](#) and allowing for easily interpolation between two rotations (spherical linear interpolation or slerp).

While Quaternions are quite difficult to fully understand, there are an exceeding number of convenience methods to allow you to use them without having to understand the math behind it. Basically, these methods involve nothing more than setting the Quaternion's x,y,z,w values using other means of representing rotations. The Quaternion is then contained in [Spatial](#) as its local rotation component.

Quaternion **q** has the form

$$\mathbf{q} = \langle w, x, y, z \rangle = w + xi + yj + zk$$

or alternatively, it can be written as:

$\mathbf{q} = \mathbf{s} + \mathbf{v}$ , where **s** represents the scalar part corresponding to the w-component of **q**, and **v** represents the vector part of the (x, y, z) components of **q**.

Multiplication of Quaternions uses the distributive law and adheres to the following rules with multiplying the imaginary components (i, j, k):

```
i2 = j2 = k2 = -1
ij = -ji = k
jk = -kj = i
ki = -ik = j
```

However, Quaternion multiplication is *not* commutative, so we have to pay attention to order.

$$q_1 q_2 = s_1 s_2 - \mathbf{v}_1 \cdot \mathbf{v}_2 + s_1 \mathbf{v}_2 + s_2 \mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2$$

Quaternions also have conjugates where the conjugate of  $\mathbf{q}$  is  $(s - \mathbf{v})$

These basic operations allow us to convert various rotation representations to Quaternions.

## Angle Axis

You might wish to represent your rotations as Angle Axis pairs. That is, you define a axis of rotation and the angle with which to rotate about this axis. Quaternion defines a method `fromAngleAxis` (and `fromAngleNormalAxis`) to create a Quaternion from this pair. This is actually used quite a bit in jME demos to continually rotate objects. You can also obtain a Angle Axis rotation from an existing Quaternion using `toAngleAxis`.

### Example - Rotate a Spatial Using fromAngleAxis

```
//rotate about the Y-Axis by approximately 1 pi
Vector3f axis = Vector3f.UNIT_Y; // this equals (0, 1, 0) and does
float angle = 3.14f;
s.getLocalRotation().fromAngleAxis(angle, axis);
```

## Three Angles

You can also represent a rotation by defining three angles. The angles represent the rotation about the individual axes. Passing in a three-element array of floats defines the angles where the first element is X, second Y and third is Z. The method provided by Quaternion is `fromAngles` and can also fill an array using `toAngles`

### Example - Rotate a Spatial Using fromAngles

```
//rotate 1 radian on the x, 3 on the y and 0 on z
float[] angles = {1, 3, 0};
s.getLocalRotation().fromAngles(angles);
```

## Three Axes

If you have three axes that define your rotation, where the axes define the left axis, up axis and directional axis respectively) you can make use of `fromAxes` to generate the Quaternion. It should be noted that this will generate a new [Matrix](#) object that is then

garbage collected, thus, this method should not be used if it will be called many times. Again, `toAxes` will populate a `Vector3f` array.

## Example - Rotate a Spatial Using fromAxes

```
//rotate a spatial to face up ~45 degrees
Vector3f[] axes = new Vector3f[3];
axes[0] = new Vector3f(-1, 0, 0); //left
axes[1] = new Vector3f(0, 0.5f, 0.5f); //up
axes[2] = new Vector3f(0, 0.5f, 0.5f); //dir

s.getLocalRotation().fromAxes(axes);
```

## Rotation Matrix

Commonly you might find yourself with a `Matrix` defining a rotation. In fact, it's very common to contain a rotation in a `Matrix` create a Quaternion, rotate the Quaternion, and then get the `Matrix` back. Quaternion contains a `fromRotationMatrix` method that will create the appropriate Quaternion based on the give `Matrix`. The `toRotationMatrix` will populate a given `Matrix`.

## Example - Rotate a Spatial Using a Rotation Matrix

```
Matrix3f mat = new Matrix3f();
mat.setColumn(0, new Vector3f(1,0,0));
mat.setColumn(1, new Vector3f(0,-1,0));
mat.setColumn(2, new Vector3f(0,0,1));

s.getLocalRotation().fromRotationMatrix(mat);
```

As you can see there are many ways to build a Quaternion. This allows you to work with rotations in a way that is conceptually easier to picture, but still build Quaternions for internal representation.

## Slerp

One of the biggest advantages to using Quaternions is allowing interpolation between two rotations. That is, if you have an initial Quaternion representing the original orientation of an object, and you have a final Quaternion representing the orientation you want the object to

face, you can do this very smoothly with slerp. Simply supply the time, where time is [0, 1] and 0 is the initial rotation and 1 is the final rotation.

## Example - Use Slerp to Rotate Between two Quaternions

```
Quaternion q1;
Quaternion q2;

//the rotation half-way between these two
Quaternion q3 = q1.slerp(q2, 0.5f);
```

</div>

## title: jMonkeyEngine3 Requirements

# jMonkeyEngine3 Requirements

This page shows software and hardware requirements from two points of view:

- For the Java developers who create games; and
- For the users (your customers) who play the games that you created.

## For Developers

These are the minimum requirements for developers who create games using the jMonkeyEngine SDK:

Operating system	Mac <u>OS X</u> , Windows, Linux, Solaris
Memory (JVM heap size)	> 40 <u>MB</u> + memory for assets
CPU	> 1 <u>GHz</u>
Graphic card	AMD/ATI Radeon 9500, NVIDIA GeForce 5 FX, Intel GMA 4500, or better supporting OpenGL 2.0 or better (native libraries are included in download)
Java Development Kit	JDK 6 or higher Your development team gets the Java Development Kit for free from <a href="http://www.java.com">http://www.java.com</a> ; for Mac <u>OS</u> , see <a href="#">apple.com</a> . At least intermediate Java experience is required.

We recommend using the [jMonkeyEngine SDK](#) as game development environment (IDE). However, any third-party Java IDE (and even a text editor and the commandline) will work fine together with the jME framework if you are experienced with the tool of your choice. For requirements of other IDEs, please see the third party's documentation.

</div>

## For Users

These are the minimum requirements for your customers, users who play the games that were created using the jMonkeyEngine3 framework:

Operating system	Mac <u>OS X</u> , Windows, Linux, Solaris
Memory (JVM heap size)	> 10 <u>MB</u> + memory for assets (models, textures, sounds, etc, of a particular game)
CPU	> 1 <u>GHz</u>
Graphic card	AMD/ATI Radeon 9500, NVIDIA GeForce 5 FX, Intel GMA 4500, or better supporting OpenGL 2.0 or better (native libraries are included in download)
Android Devices	(For mobile games only) Android 2.2 <u>OS</u> , graphics card supporting OpenGL 2
Java Runtime	Java 5 or higher The Java Virtual Machine (JVM) is required to run jME games. The JVM is often preinstalled, if not, your users get it for free from <a href="http://www.java.com">http://www.java.com</a> ; for Mac <u>OS</u> , see <a href="http://apple.com">apple.com</a> .

Make sure to inform your users about these requirements.

</div>

## **title:**

Rise of Mutants project by BigBootsTeam.

Project is hosted here: <https://code.google.com/p/rise-of-mutants/source/list>

Get a local copy of the Rise of Mutants repository with this command: hg clone  
<https://paulgeraskin@code.google.com/p/rise-of-mutants/>

Features:

- World Editor is Blender (export to Ogre scene).
- Entity System.
- Ready Framework.
- Ready Gameplay.
- Model Viewer tool.
- LightBlow Shader is used.

ScreenShot: <http://i.imgur.com/uFRw4.jpg>

Video: [http://www.youtube.com/watch?v=\\_aFSEJlISyI&feature=player\\_embedded](http://www.youtube.com/watch?v=_aFSEJlISyI&feature=player_embedded)

## title: 3-D Rotation

# 3-D Rotation

*Bad news: 3D rotation is done using matrix calculus.*

*Good news: If you do not understand calculus, there are two simple rules how you get it right.*

**3D rotation** is a crazy mathematical operation where you need to multiply all vertices in your object by four floating point numbers; the multiplication is referred to as concatenation, the array of four numbers {x,y,z,w} is referred to as quaternion. Don't worry, the 3D engine does the tough work for you. All you need to know is:

**The Quaternion** is an object capable of deep-freezing and storing a rotation that you can apply to a 3D object.

</div>

## Using Quaternions for Rotation

To store a rotation in a Quaternion, you must specify two things: The angle and the axis of the rotation.

- The rotation angle is defined as a multiple of the number PI.
- The rotation axis is defined by a vector: Think of them in terms of “pitch”, “yaw”, and “roll”.

Example:

```
/* This quaternion stores a 180 degree rolling rotation */
Quaternion roll180 = new Quaternion();
roll180.fromAngleAxis(FastMath.PI , new Vector3f(0,0,1));
/* The rotation is applied: The object rolls by 180 degrees. */
thingamajig.setLocalRotation(roll180);
```

So how to choose the right numbers for the Quaternion parameters? I'll give you my cheat-sheet:

Rotation around Axis?	Use this Axis Vector!	Examples for this kind of rotation
X axis	(1,0,0)	A plane pitches. Nodding your head.
Y axis	(0,1,0)	A plane yaws. A vehicle turns. Shaking your head.
Z axis	(0,0,1)	A plane rolls or banks. Cocking your head.

Note: These are the three most common examples – technically you can rotate around any axis expressed by a vector.

Angle?	Use Radians!	Examples
45 degrees	FastMath.PI / 4	eighth of a circle
90 degrees	FastMath.PI / 2	quarter circle, 3 o'clock
180 degrees	FastMath.PI	half circle, 6 o'clock
270 degrees	FastMath.PI * 3 / 2	three quarter circle, 9 o'clock
360 degrees	FastMath.PI * 2	full circle, 12 o'clock <input type="checkbox"/>
g degrees	FastMath.PI * g / 180	any angle <input type="checkbox"/> g

**Important:** You must specify angles in [Radians](#) (multiples or fractions of PI). If you use degrees, you will just get useless results.

How to use these tables to specify a certain rotation:

1. Pick the appropriate vector from the axis table.
2. Pick the appropriate radians value from the angle table.
3. Create a Quaternion to store this rotation. `... fromAngleAxis( radians , vector )`
4. Apply the Quaternion to a node to rotate it. `... setLocalRotation(...)`

Quaternion objects can be used as often as you want, so give them meaningful names, like `roll190, pitch45, yaw180`.

[More about Quaternions...](#)

</div>

## Code Sample

```
/* We start out with a horizontal object */
Cylinder cylinder = new Cylinder("post", 10, 10, 1, 10);
cylinder.setModelBound(new BoundingBox());
/* Create a 90-degree-pitch Quaternion. */
Quaternion pitch90 = new Quaternion();
pitch90.fromAngleAxis(FastMath.PI/2, new Vector3f(1,0,0));
/* Apply the rotation to the object */
cylinder.setLocalRotation(pitch90);
/* Update the model. Now it's vertical. */
cylinder.updateModelBound();
cylinder.updateGeometry();
```

</div>

## Interpolating Rotations

You can specify two rotations, and then have jme calculate (interpolate) the steps between two rotations:

- com.jme3.math.Quaternion, slerp() – store an interpolated step between two rotations
  - [com.jme.math.Quaternion](#)
  - [com.jme.math.TransformQuaternion](#)

## "Adding" Rotations

You can concatenate (add) rotations: This means you turn the object first around one axis, then around the other, in one step.

```
Quaternion myRotation = pitch90.mult(roll45); /* pitch and roll */
```

</div>

## Troubleshooting Rotations

Does the object end up in an unexpected location, or at an unexpected angle? If you are getting weird results, check the following:

1. 3-D transformations are non-commutative! This means it often makes a huge difference whether you first move a node and then rotate it around an axis, or first rotate the node around an axis and then move it. Make sure your code does what you mean to do.
2. Are you intending to rotate around the object's origin along an axis, or around another pivot point outside the object? If you are trying to *rotate an object around a pivot point*, you have to create an (invisible) pivot node first, and attach the object to it. Then apply the rotation to the *parental pivot node*, not to the child object itself!
3. Did you enter the angle in degrees (0 - 360°) or radians (0 - 2\*PI)? A 3D engine expects radians, so make sure to convert your values! Formula:  $g^\circ = \text{FastMath.PI} * g / 180$

## Tip: Transformation Matrix

This here is just about rotation, but there are three types of 3-D transformation: [rotate](#), [scale](#), and [translate](#).

You can do all transformations in individual steps (and then update the objects geometry and bounds), or you can combine them and transform the object in one step. If you have a lot of repetitive movement going on in your game it's worth learning more about Transformation Matrices for optimization. JME can also help you interpolate the steps between two fixed transformations.

</p>

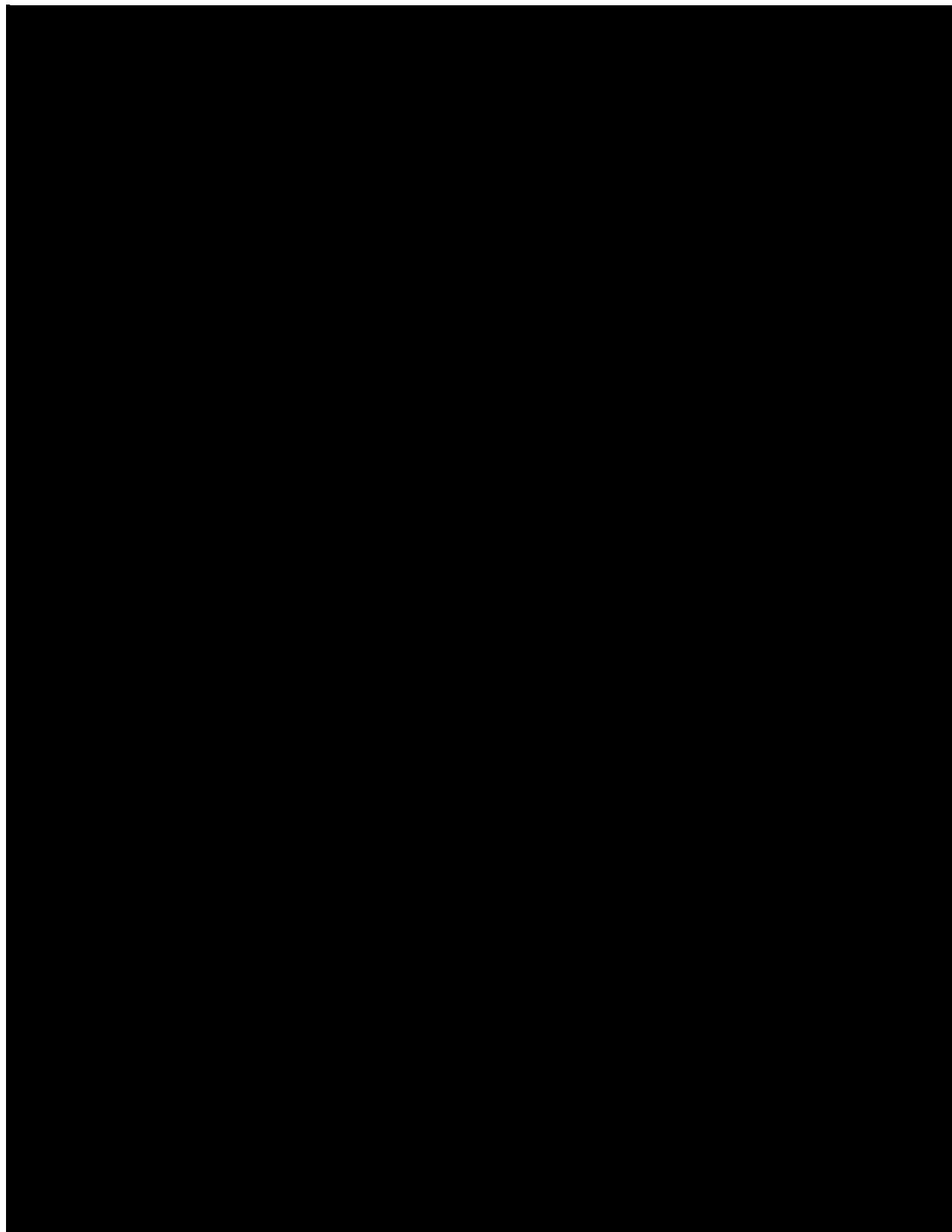
- com.jme3.math.Transform, interpolateTransforms() – interpolate a step between two transformations
  - [com.jme.math.TransformMatrix](#)

</div>

---

## **title: jMonkeyEngine3 Scene Graph for Dummies**

### **jMonkeyEngine3 Scene Graph for Dummies**



- You can open this presentation in [Fullscreen Mode](#).
- When using [Safari](#) as a browser the presentation is animated.

</div>

# **title: Scripting in JME3 with Groovy. All about it!**

## **Scripting in JME3 with Groovy. All about it!**

**Monkeys,**

For anyone who still ask for something like “**Scripting**” with **JME3**, here is it, about it at once!

We nearly reach 3.0, i’m so excited about it and want to write something for it. In this post you will have a good run from Zero to Hero with Groovy and Monkey :p Not kidding!



**Quick question:** What this related to my other tuts ?

[atomixtuts] **Answer:** As I wrote the others, I thought I should write this first, because if no one know about Groovy, no one can understand a single line of my code , and it’s bad!  
</div>

## **Introduction:**

We will go from basic setup of Groovy in JMP and then example by example every aspect of game you can develop with Groovy. More advanced topic come at the end, eg: the way to get the speed of Java, meta-programing, a lot of programming patterns (PP), even AI and Constraint Programming... Sounds insanse? Nah ah...

Let me know if you want to write or to read other topics of Groovy or Scripting. Thanks!  
Let’s start!

</div>

## **Content:**

1. GET STARTED
  2. LEARN GROOVY
    - i. Syntax and Gotchas
    - ii. Meta-programming
    - iii. Groovy Builder – SwingBuilder
    - iv. Groovy – for smarty
    - v. GPars
  3. EMBED GROOVY
    - i. Groovy in JME3
    - ii. Groovy in JMP
    - iii. Groovy everywhere (snippets)
  4. INTO THE GAME
    - i. Basic Scripts
    - ii. Event – Trigger Manager
    - iii. Configurations with Groovy
    - iv. AI Tricks with Groovy
    - v. Build Script with Groovy
  5. DESIGN PATTERNS IN GROOVY GAME (WIP)
  6. ADVANCED TRICKS
    - i. Hack the JVM with Groovy
    - ii. Codegen
    - iii. Groovy – Almighty God!
- 

## GET STARTED

If you already know about Groovy and just curious about how to intergrate Groovy into JME3. Go straight to Part [into\\_the\\_game](#)

</div>

## WHAT IS GROOVY?

Groovy... 

is an agile and dynamic language for the Java Virtual Machine builds upon the strengths of Java but has additional power features inspired by languages like Python, Ruby and Smalltalk

<http://groovy.codehaus.org/>

## Highlight

The shortest highlight of the language can be found here:

[http://en.wikipedia.org/wiki/Groovy\\_%28programming\\_language%29](http://en.wikipedia.org/wiki/Groovy_%28programming_language%29)

- Most valid Java files are also valid Groovy files. Although the two languages are similar, Groovy code can be more compact, because it does not require all the elements that Java requires. This makes it possible for Java programmers to gradually learn Groovy by starting with familiar Java syntax before acquiring more Groovy idioms.
- Groovy features not available in Java include both static and dynamic typing (with the def keyword), closures, operator overloading, native syntax for lists and associative arrays (maps), native support for regular expressions, polymorphic iteration, expressions embedded inside strings, additional helper methods, and the safe navigation operator “?.” to automatically check for nulls (for example, “variable?.method()”, or “variable?.field”).
- Since version 2 Groovy also supports modularity, being able to ship only the needed jars according to the project needs, thus reducing the size of groovy's lib, type checking, static compilation, Project Coin syntax enhancements, multicatch blocks and ongoing performance enhancements using JDK7's invoke dynamic instruction.
- Groovy provides native support for various markup languages such as XML and [HTML](#), accomplished via an inline DOM syntax. This feature enables the definition and manipulation of many types of heterogeneous data assets with a uniform and concise syntax and programming methodology.[citation needed]
- Unlike Java, a Groovy source code file can be executed as an (uncompiled) script if it contains code outside any class definition, is a class with a main method, or is a Runnable or GroovyTestCase. A Groovy script is fully parsed, compiled, and generated before execution (similar to Perl and Ruby). (This occurs under the hood, and the compiled version is not saved as an artifact of the process.)

## SETUP GROOVY?

If you used JMP to code your game (I don't know about Eclipse users, sorry). Go to Update Center to install Groovy plugin, then download the lastest Groovy (ver2.1) and wrap it as a Library. You are ready for the adventure!

<http://groovy.codehaus.org/Download?nc>

## WHAT CAN BE SCRIPT

**or “TO SCRIPT OR NOT TO SCRIPT, is the PROBLEM”?**

**Everything.** You can do almost every thing with Groovy just like with Java.

In this post i will show example by example every aspect of game you can develop with Groovy.

**Pros:**

- Scripting is very common and intuitive way to do game programing. It's common because it's shorter, cleaner, easy to read, maintain and re-use.
- Groovy is young but developed by very talent people, a lot of devoted contributors.
- Web and Enterprise in your hand. Ever heard of Grails <http://grails.org/>?
- Multi-additions to fullfil Java. God-like in Swing, ORM, XML...
- Performance improved recently: If you worry about the performance, , in the next release, it can even get to the speed of Java, and soon to be a very competitive opponent to Scala! Read this? <http://java.dzone.com/articles/groovy-20-performance-compared>

**Cons:** It's good, but what about the down-side?

- Can not run in Android, yet!
- Some things can be wrong without noticed, appeared in run-time like every scripting language
- Still a performance problem.

## WHEN TO USE SCRIPTING:

Some obvious but always existing problems of Scripting.

First every scripting language got the same type-safe dilemma. If you invest too much into Scripting, you fall immediately into the mess that hidden errors which are always very hard to find, only show up in run-time. The balance between benefit and hell of Scripting is thin. Duck-typing is not always a win-win.

## Not type-safe

As Groovy support Duck-typing, is almost impossible to know the type, methods of the object you want to use. This can be improved if you are in Static mode but this mode simply not what we really want with Scripting purpose?

So, as the question had been asked by a forum's member:

Heh. I'd love to go Groovy myself, but I've been finding it very hard for me to explore the set of methods that a passed-in object supports.

### Answer:

From my experience, just ask you self, how "natural" your code are coded, in **OOP** sense:

**Chicken.eat(rice)** You know what methods and their parameter's type, and name.

**Monkey.eat(banana)** You know what common in classes in a package. Without knowing the inheritance and interface they implemented.

**Human.eat([chicken,rice,banana])** You can guess Human are derivated from Monkey and code are coded flexible, ex: methods are multi-type, optional param. etc...

If it have that level of "natural" sense, you don't have to learn by heart at all, so use scripting in the situation.

In other hand, this very related to IDE support for such language. If you watch closely, Groovy going to have better support in Netbean:

[https://blogs.oracle.com/netbeansgroovy/entry/groovy\\_refactoring\\_in\\_netbeans](https://blogs.oracle.com/netbeansgroovy/entry/groovy_refactoring_in_netbeans)

### NOTE:

- You **CAN** use GROOVY for Java as Lua for C++ (even much more better)
- You **CAN** get GROOVY run as FAST as Java
- You **CAN** let GROOVY seamlesly intergrated with Java and other JVM languages.
- Last but not least, Groovy **kick asses!** :p

## LEARN GROOVY

If you already know about Groovy and just curious about how to intergrate Groovy into JME3. Go straight to Part [into\\_the\\_game](#)

First, Groovy is much more shorter – cleaner than Java. It seamlessly get Java to the world of functional programming, like Python, Haskell, etc, but still make Java developer feel at home. You can read much more in the Groovy site and the internet, so I will not blow it up.

Anyway, let's learn some Groovy syntax, I bet you can master it in 3 hours!

GOTO [groovy\\_learn](#)

## Groovy – for smarty

So, what you can do with Groovy?  everything, even get laid!

I means use your imagination. I give you some examples:

- Fasten the build process
- Replace almost the configuration
- Extract infos from XML and text, web...
- Convert RenderMonkey, FXComposer shaders
- Script the Dialoge, Cinematic,...
- Make In-game Editor, JMP's plugins
- Make a whole freaking game
- Even feed my dogs ...

[10 more]

What I want to say is **Groovy** is for smarty, master it and it save you *freaking big times* !

Java and Groovy are a sweetest combination of programing languages I ever tried beside of dozen of others.

Some of the example above will be include in this post or in my AtomScript project!

</div>

## Official examples & Misc

Here are some website that you can find a lot of examples from simple to complicated tasks:

<http://groovy.codehaus.org/Cookbook+Examples>

<http://www.groovyexamples.org/>

<http://snipplr.com/all/language/groovy>

[http://rosettacode.org/wiki/Rosetta\\_Code](http://rosettacode.org/wiki/Rosetta_Code) ← learn Groovy and java if you come from other programming languages.

## GPars

If you already know Groovy, I recommend you to try **GPars! Groovy Parallel Systems**. Why? Because it's **#\$kin awesome**, that's why? Every smart monkey and Java developer should know about it, to build apps and games!

*The GPars framework offers Java developers intuitive and safe ways to handle Java or Groovy tasks concurrently. Leveraging the enormous flexibility of the Groovy programming language and building on proven Java technologies, we aim to make concurrent programming for multi-core hardware intuitive, robust and enjoyable.*

<http://gpars.codehaus.org/>

I will explain some concepts and usages of GPars that help me a lot in JME3's game and other tasks!

GOTO [gpars\\_usecases](#)

</div>

## EMBED GROOVY

First I recommend all who don't know much about Groovy read this official documentation  
<http://groovy.codehaus.org/Embedding+Groovy>

Groovy is very suitable for embedding in Java application, even game. Our intention here is to get Groovy to work with JME in few ways. Some common problems, difficulties may arised cause of the differencies, uncompatiable between Java-Groovy-Native OpenGL.

So technical problem and requirement will be dicussed first, then the Design of the integration is sketched, at last the full implementation. The full source code are in the AtomScript project!

</div>

## OVERVIEW

## TECH PROBS

## NEED OF POWERFUL SCRIPTING SYSTEM



## DESIGN & ARCHITECTURE

Slide

## IMPLEMENTATION

Slide

## Groovy in JME3

ScriptEngine

ScriptBase

Tools

## Groovy in JMP

**ScriptBaseTopComponent**

**ScriptEngineModule**

**Advanced Tricks to get JMP Scripted**

**Groovy everywhere (snippets)**

**Extract infos from XML and text, web...**

**Convert RenderMonkey, FXComposer shaders**

GOTO [snippets](#)

## INTO THE GAME

Grab the example code from the AtomScript project link

</div>

**Basic Scripts**

**Rotate the wheel**

**Travel a tree**

**Queue a task**

## GroovyAppState

### ClosureCondition

GOTO [groovy\\_basicscripts](#)

### Event – Trigger - Manager

The first idea come to my mind when think of game programming is a game cycle-update or events.

In fact, frequently update and sudden event is quite opposite paradigm, the point is to get the best of both world in one design. But can we? At least I can answer partly yes. And such sollution I've seen in big database system use the same hyrid concept.

I also saw in the forum, guys had conversation about Entity System, which partly envolve such design... But this one it's different. It's not general, I means that the code below tent to be used in kind of RTS game like War-craft of Starcraft, and I precisely model it like those two games. And the codes are very short, extremely short, show the power of Groovy in the usecase.

GOTO [groovy\\_event](#)

### Configurations with Groovy

Think about the way to config your game's screen resolution, keyboard, database connection, without have to write and parse java property or XML files. Groovy script is text file but much more powerful, like it has variables, methods (def), loop (for), conditions (if-else)...etc to build complicated things (like a program), compared to just plain text. In short Groovy can replace almost every configuration task you can imagine. This topic about using Groovy scrips for that purpose.

GOTO [groovy\\_config](#)

### AI Tricks with Groovy

As in the introduction above I said this wiki will include everything about Scripting... So, it should also include AI (Artificial Intelligent) ... But I'm not going to tell you all about AI in this wiki, it should be more in another wiki of some AI professiors. I just want to show how a quick implementation of simple AI models can be coded in Groovy:

GOTO [ai](#)

## Finite State Machine

What is the most simple but affective techique to make AI. It's FSM

## Decision Tree

Builder

## Pattern Matching

Regexp

## Simple Chatbot

Builder + Closure

## Simple Goalbase Agent



## Simple Path finding

Use Groovy extension

## Simple Steering behavior



## Build Script with Groovy

Groovy can use Ant and Maven in a snap. but wait... it also has its own build extension named Gradle.

<http://www.gradle.org/>

Check this out: For JME3 Desktop:

For JME3 Android: <http://tools.android.com/tech-docs/new-build-system/user-guide>

# DESIGN PATTERNS IN GROOVY GAME (WIP)

# ADVANCED TRICKS

## Hack the JVM with Groovy

### Codegen

This should be in another wiki but somehow is super fit for an example of advanced Groovy usage. The project CodeGen - Code generator is my first Groovy project. It's tented to be a general code generator for Java, Groovy, GLSL and can also be a fun playground for non-developer. It inspirated by the concept of:

Alice <http://www.alice.org/index.php>

GreenFoot <http://www.greenfoot.org/door>

and an old plugin of PGI - a JME forum's member : PgiLogic

<http://hub.jmonkeyengine.org/forum/topic/dead-combinable-logic-framework/>

It's going to be in a suite for making Jme3 Games : Atom framework. Visit : GOTO  
[atom\\_framework](#) GOTO [codegen](#)

### Groovy – Almighty God!

### Get to the speed of Java

### Extension and Modulize

### Database and ORM

### DSL

### Visit the Moon

## CONCLUSION

After reading for a while, I guess you are in love with Groovy already. You're welcome! 

This page **CAN NOT** be a full description of Groovy... but a snapshot of its good with a few home grown codes for your JME3 game!

Beside of knowing the power and the weaknesses of the language and the way to use it in your everyday life. If you want to have the full snippets, download AtomScript project.

Any correction are welcome!

</div>

## title: Setting up JME3 in Eclipse

# Setting up JME3 in Eclipse

For development with the jMonkeyEngine 3, we recommend to use the jMonkeyEngine SDK.

Alternatively, you can use your favorite IDE: In this tutorial we show how to download and set up the latest nightly build of the jMonkeyEngine 3 for use with the Eclipse IDE.

Instructions for [NetBeans IDE](#) are also available.

## Downloading jME3

The currently available JAR binaries are the nightly builds.

1. Download the most recent zipped build from  
<http://updates.jmonkeyengine.org/stable/3.0/engine>
2. Unzip the file and save it as `jME3_2012-xx-xx` in your home directory (`$HOME`). You should see the following files and directories:
  - `lib/` – The jMonkeyEngine binaries, and libraries used by the jMonkeyEngine. (Don't remove)
  - `TestChooser.exe` – Run this file to see various feature demos. (optional)
  - `javadoc/` – jME3 API documentation. (optional)

## Creating a New Game Project

- In Eclipse, choose File > New > Java Project
- Project Name: HelloJME3
- Click Finish

The new project appears in the Explorer.

## Setting up Dependencies

Your project depends on the jMonkeyEngine libraries and needs to know where they are.

1. Right-click the project in the explorer and choose `Build Path > Add External Archives`

2. In the “JAR selection” dialog, browse to the `$HOME/jME3_2012-xx-xx` directory.
3. Select all JARs in the `lib` directory and click Open.

All necessary JAR libraries are now on the classpath and should appear in the Referenced Libraries list. For a detailed description of the separate jar files see [this list](#).

## Setting up Assets

The easiest way to make sure the asset manager can access the assets is by adding the assets folder to the classpath.

1. Go to Project Properties
2. Select Java Build Path
3. Under the Source tab click add folder
4. Add your Assets folder

## Writing a Simple Application

1. From the menu call File > New > New Package. Name the src package for example `hello`.
2. From the menu call File > New > Class.
  - Select package `hello`.
  - Name the class for example `MyGame`.
  - Superclass: `com.jme3.app.SimpleApplication`
  - Make sure that the checkbox to Create the `main()` Method is active
  - Make sure that the checkbox to Inheriting Abstract Methods is active
  - Click Finish.

You can now continue to write [your first jme3 application!](#)

[documentation](#), [install](#), [eclipse](#)  
</div>

## title: Setting up JME3 in Netbeans 6.x

# Setting up JME3 in Netbeans 6.x

For development with the jMonkeyEngine 3, we recommend to use the jMonkeyEngine [SDK](#).

Alternatively, you can use your favorite IDE: In this tutorial we show how to download and set up the latest nightly build of the jMonkeyEngine 3 for use with the NetBeans IDE.

Instructions for [Eclipse](#) are also available.

Note that the jMonkeyEngine SDK is built in top of the NetBeans Platform, and is identical to the NetBeans IDE for Java (plus some unique NetBeans plugins). Basically it's redundant and unnecessary to set up jME for NetBeans – but if you want to, it's easily possible.

</div>

## Downloading jME3

The currently available JAR binaries are the nightly builds.

1. Download the most recent zipped build from <http://nightly.jmonkeyengine.org/>
2. Unzip the file and save it as `jME3_xx-xx-2010` in your `NetBeansProjects` directory. You should see the following files and directories:
  - i. `lib/` – The jMonkeyEngine binaries and libraries. (Don't remove.)
  - ii. `TestChooser.exe` – Run this file to see various feature demos. (optional)
  - iii. `javadoc/` – jME3 API documentation. (optional)

## Creating a Project

In NetBeans, choose File > New Project, select Java > Java Application, click Next.

- Project Name: HelloJME3
- Project Location: `~/NetBeansProjects`
- Create main() Class: No
- Set as Main Project: Yes.
- Click Finish

The new project appears in the Projects window.

# Setting up Dependencies

Your project depends on the jMonkeyEngine libraries and needs to know where they are.

1. In the Projects window, right-click the project's Libraries node and choose "Add JAR/Folder".
2. In the "Add JAR/Folder" dialog, browse to the `NetBeansProjects/jME3_xx-xx-2010` directory.
3. Select all JARs in `lib/` and click Select.

The necessary libraries are now on the classpath and should appear in the Libraries list.

## Optional: Configuring the JavaDoc Popups in NetBeans

Configuring Javadoc popups for jme3 classes in NetBeans can be very helpful during the development phase of your game.

1. In the HelloJME3 project
  - i. Right-click the Libraries node and choose Properties from the context menu
  - ii. Select the jMonkeyEngine3.jar library entry from the list and click Edit.
    - i. Javadoc: Browse to JME3's `javadoc` directory and click select.
    - ii. Sources: Browse to JME3's `src` directory and click select.
  - iii. Click OK.

Open a class of your HelloJME3 project, place the caret into a jme3 class, and press **ctrl-space** to see the javadoc popup, as well as code-completion.

## Build & Run Tips in NetBeans

How to build and run in NetBeans:

- Clean and build the whole project by pressing Shift-F11.
- Run any file that is open in the editor and has a main() class by pressing Shift-F6.
- Run the Main class of the project by pressing F6.

Tips for configuring the main class in NetBeans:

- Right-click the HelloJME3 project and choose "Set as main project". Now you can use the toolbar buttons (clean&build, run, debug, etc) to control this project.
- Right-click the HelloJME3 project and choose Properties. Go to the Run section.

- Under Main Class, specify the class that will be executed when you run the whole project. For now, enter `hello.HelloJME3`.

## Writing a Simple Application

You can now continue to write [your first jme3 application!](#)

[documentation](#), [install](#)

</div>

## **title:**

### **jME3 Introduction to Shaders Video Tutorials**

- [Video: General Introduction](#)
- [Video: Introduction to materials](#)
- [Video: Introduction to shaders](#)
- [Video: Simple Color Shader](#)
- [Video: Flattening vertices Shader](#)
- [Video: Rainbow color Shader](#)
- [Video: Animating a waving flag \(Part 1\)](#)
- [Video: Animating a waving flag \(Part 2\)](#)
- [Video: Dissolve shader](#)

## **title:**

Shaders for JMonkey Engine.

Project is hosted here: <http://code.google.com/p/jme-glsl-shaders/>

Get a local copy of the jme-glsl-shaders repository with this command: hg clone <https://paulgeraskin@code.google.com/p/jme-glsl-shaders/>

ScreenShot: [http://dl.dropbox.com/u/26887202/123/ShaderBlow\\_03.jpg](http://dl.dropbox.com/u/26887202/123/ShaderBlow_03.jpg)

---

Other resources to help you with shaders and GameDesign/LevelDesign:

Unity Shader Bible - [http://code.google.com/p/jme-glsl-shaders/downloads/detail?name=FastMobileShaders\\_siggraph2011.zip&can=2&q=](http://code.google.com/p/jme-glsl-shaders/downloads/detail?name=FastMobileShaders_siggraph2011.zip&can=2&q=)

Valve Shader Bible - [http://code.google.com/p/jme-glsl-shaders/downloads/detail?name=D3DTutorial10\\_Half-<br>Life2\\_Shading.pdf&can=2&q=](http://code.google.com/p/jme-glsl-shaders/downloads/detail?name=D3DTutorial10_Half-<br>Life2_Shading.pdf&can=2&q=)

Best LevelDesign/GameDesign Resource - <http://www.worldofleveldesign.com/>

Software for NormalMaps making - [http://shadermap.com/shadermap\\_pro.php](http://shadermap.com/shadermap_pro.php)

Software for CubeMaps editing -  
<http://developer.amd.com/archive/gpu/cubemapgen/pages/default.aspx>

Software for cool Sprites making - <http://www.rigzsoft.co.uk/>

## title: Starting a JME3 application from the Commandline

# Starting a JME3 application from the Commandline

Although we recommend the jMonkeyEngine [SDK](#) for developing JME3 games, you can use any IDE (integrated development environment) such as [NetBeans](#) or [Eclipse](#), and even work freely from the commandline. Here is a generic IDE-independent “getting started” tutorial.

This example shows how to set up and run a simple application (HelloJME3) that depends on the jMonkeyEngine3 libraries.

The directory structure will look as follows:

```
jme3/
jme3/lib
jme3/src
...
HelloJME3/
HelloJME3/lib
HelloJME3/assets
HelloJME3/src
...
```

## Installing the JME3 Framework

To install the development version of jme3, [download the nightly build](#), unzip the folder into a directory named `jme3`. The filenames here are just an example, but they will always be something like `jME3_xx-xx-2011`.

```
mkdir jme3
cd jme3
unzip jME3_01-18-2011.zip
```

Alternatively, you can build JME3 from the sources. (Recommended for JME3 developers.)

```
svn checkout https://jmonkeyengine.googlecode.com/svn/branches/3.0f
cd jme3
ant run
cd ..
```

If you see a Test Chooser application open now, the build was successful. **Tip:** Use just `ant` instead of `ant run` to build the libraries without running the demos.

## Sample Project Directory Structure

First we set up the directory and source package structure for your game project. Note that the game project directory `HelloJME3` is on the same level as your `jme3` checkout. In this example, we create a Java package that we call `hello` in the source directory.

```
mkdir HelloJME3
mkdir HelloJME3/src
mkdir HelloJME3/src/hello
```

## Libraries

Next you copy the necessary JAR libraries from the download to your project. You only have to do this set of steps once every time you download a new JME3 build. For a detailed description of the separate jar files see [this list](#).

```
mkdir HelloJME3/build
mkdir HelloJME3/lib
cp jme3/lib/*.* HelloJME3/lib
```

If you have built JME3 from the sources, then the copy paths are different:

```
mkdir HelloJME3/build
mkdir HelloJME3/lib
cp jme3/dist/*.* HelloJME3/lib
```

## Sample Code

To test your setup, create the file `HelloJME3/src/hello/HelloJME3.java` with any text editor, paste the following sample code, and save.

```
package hello;

import com.jme3.app.SimpleApplication;
import com.jme3.material.Material;
import com.jme3.math.Vector3f;
import com.jme3.scene.Geometry;
import com.jme3.scene.shape.Box;
import com.jme3.math.ColorRGBA;

public class HelloJME3 extends SimpleApplication {

 public static void main(String[] args){
 HelloJME3 app = new HelloJME3();
 app.start();
 }

 @Override
 public void simpleInitApp() {
 Box b = new Box(Vector3f.ZERO, 1, 1, 1);
 Geometry geom = new Geometry("Box", b);
 Material mat = new Material(assetManager,
 "Common/MatDefs/Misc/Unshaded.j3md");
 mat.setColor("Color", ColorRGBA.Blue);
 geom.setMaterial(mat);
 rootNode.attachChild(geom);
 }
}
```

## Build and Run

We build the sample application into the build directory...

```
cd HelloJME3
javac -d build -cp "lib/eventbus-1.4.jar:lib/j-ogg-oggd.jar:lib/j-c
```

... and run it.

```
cd build
java -cp "../lib/eventbus-1.4.jar:../lib/j-ogg-oggd.jar:../lib/j-og
```

Note: If you use Windows, the classpath separator is ";" instead of ":".

If a settings dialog pops up, confirm the default settings. You should now see a simple window with a 3-D cube. Use the mouse and the WASD keys to move. It works!

## Recommended Asset Directory Structure

For [multi-media files, models, and other assets](#), we recommend creating the following project structure:

```
cd HelloJME3
mkdir assets
mkdir assets/Interface
mkdir assets/Materials
mkdir assets/MatDefs
mkdir assets/Models
mkdir assets/Scenes
mkdir assets/Shaders
mkdir assets/Sounds
mkdir assets/Textures
```

This directory structure will allow [SimpleApplication](#)'s default [AssetManager](#) to load media files from your `assets` directory, like in this example:

```
import com.jme3.scene.Spatial;
...
 Spatial elephant = assetManager.loadModel("Models/Elephant/Elephant.meshxml");
 rootNode.attachChild(elephant);
...

```

You will learn more about the asset manager and how to customize it later. For now feel free to structure your assets (images, textures, models) into further sub-directories, like in this example the `assets/models/Elephant` directory that contains the `elephant.meshxml` model and its materials.

## Next Steps

Now follow the [tutorials](#) and write your first jMonkeyEngine game.

[documentation](#), [install](#)

</div>

## **title: 3D Game Development Terminology**

# **3D Game Development Terminology**

Before you start, make certain you are familiar with the following concepts and terminology.

## **3D Graphics and Audio**

**OpenGL** is the Open Graphics Library, a platform-independent specification for rendering 2D/3D computer graphics. For Java, there are two implementations of OpenGL-based renderers:

1. Lightweight Java Game Library (LWJGL).
2. Java OpenGL (JOGL)

**OpenAL** is the Open Audio Library, a platform-independent 3D audio [API](#).

## **Context, Display, Renderer**

The **jME Context** makes settings, renderer, timer, input and event listeners, display system, accessible to a JME game.

- The **jME Display System** is what draws the custom JME window (instead of Java Swing).
- The **Input System** is the component that lets you respond to user input: Mouse clicks and movements, keyboard presses, and joystick motions.
- The **Renderer** is what does all the work of calculating how to draw the 3D scenegraph to the 2D screen.
  - The **Shader** is a programmable part of the rendering pipeline. The jME3 game engine uses it to offer advanced customizable materials.

## **Geometry**

## **Polygon, Mesh, Vertex**

Most visible objects in a 3D scene are made up of polygon meshes – characters, terrains, buildings, etc. A mesh is a grid-like structure that represents a complex shape. The advantage of a mesh is that it is mathematically simple enough to render in real time, and detailed enough to be recognizable.

Every shape is reduced to a number of connected polygons, usually triangles; even round surfaces such as spheres are reduced to a grid of triangles. The polygons' corner points are called vertices. Every vertex is positioned at a coordinate, all vertices together describe the outline of the shape.

You create 3D meshes in tools called mesh editors, e.g in Blender. The jMonkeyEngine can load finished meshes (=models) and arrange them to scenes, but it cannot edit the mesh itself.

## Materials: Color, Lighting/Shading

What we call “color” is merely part of an object's light reflection. The onlooker's brain uses shading and reflecting properties to infer an object's shape and material. Factors like these make all the difference between chalk vs milk, skin vs paper, water vs plastic, etc! ([External examples](#))

## Color

### Ambient color

- The uniform base color of the mesh – what it looks like when not influenced by any light source.
- Usually similar to the Diffuse color.
- This is the minimum color you need for an object to be visible.

### Diffuse color

- The base color of the mesh plus shattered light and shadows that are caused by a light source.
- Usually similar to the Ambient color.

## Light Sources

## Emissive color

- The color of light emitted by a light source or glowing material.
- Only glowing materials such as lights have an emissive color, normal objects don't have this property.
- Often white (sun light).

## Reflections

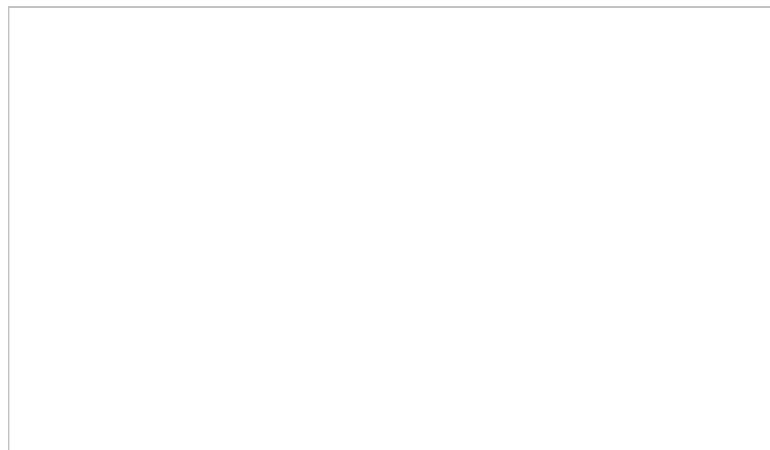
### Shininess

- Degree of shininess of a surface (1-128).
- Shiny objects have small, clearly outlined specular highlights. (E.g. glass, water, silver)
- Normal objects have wide, blurry specular highlights. (E.g. metal, plastic, stone, polished materials)
- Uneven objects are not shiny and have no specular highlights. (E.g. cloth, paper, wood, snow)

Set the Specular color to ColorRGBA.Black to switch off shininess.

### Specular Color

- If the material is shiny, then the Specular Color is the color of the reflected highlights.
- Usually the same as the emissive color of the light source (e.g. white).
- You can use colors to achieve special specular effects, such as metallic or iridescent reflections.
- Non-shiny objects have a black specular color.



## Materials: Textures

Textures are part of Materials. In the simplest case, an object could have just one texture, the Color Map, loaded from one image file. When you think back of old computer games you'll remember this looks quite plain.

The more information you (the game designer) provide additionally to the Color Map, the higher the degree of detail and realism. Whether you want photo-realistic rendering or "toon" rendering (Cel Shading), everything depends on the quality of your materials and textures. Modern 3D graphics use several layers of information to describe one material, each mapped layer is a texture.

Got no textures? [Download free textures from opengameart.org](#). Remember to keep the copyright notice together with the textures!

</div>

## Texture Mapping

### Color Map / Diffuse Map

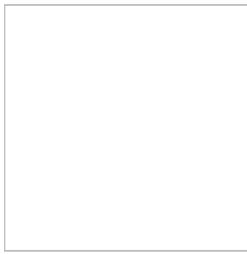


- A plain image file or a procedural texture that describes an object's visible surface.
- The image can have alpha channels for transparency.
- **A Color Map is the minimum texture.** You can map more textures as optional improvements.
- Color Maps are unshaded. The same is called Diffuse Map in a Phong-illuminated material, because this texture defines the basic colors of light that are *diffused* by this object.

### Bump Map

Bump maps are used to describe detailed shapes that would be too hard or simply too inefficient to sculpt in a mesh editor. There are two types:

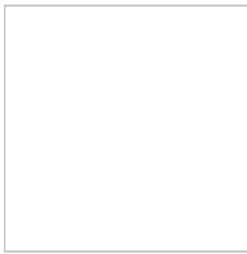
- You use Normal Maps to model tiny details such as cracks in walls, rust, skin texture, or a canvas weave ( ([More on BumpMaps](#))).
- You use Height Maps to model large terrains with valleys and mountains.



## Height Map

- A height map is a grayscale image looking similar to a terrain map used in topography. Brighter grays represent higher areas and darker grays lower areas.
- A heightmap can represent 256 height levels and is mostly used to roughly outline terrains.
- You can draw a heightmap by hand in any image editor.

## Normal Map



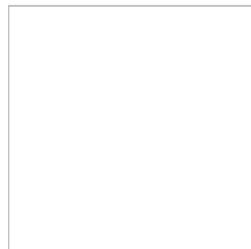
- A well-done Normal Map makes a shape more detailed – without the need to add costly polygons to the mesh. It contains shading information that makes the object appear smoother and more fine-grained.
- When you open a Normal Map in an image editor, it looks like a false-color version of the Color Map. Normal maps however are never used for coloring, instead, each the color values encode displacement data of bumps and cracks on the surface. Displacement data is represented by the Surface Normals of the slopes, hence the name.
- You cannot draw or edit normal maps by hand, professional designers use software to calculate them off high-quality 3D models. You can either buy a professional texture set, or find free collections that include Normal Maps.

## Specular Map



- A Specular Map further improves the realism of an object's surface: It contains extra information about shininess and makes the shape appear more realistically illuminated.
- Start out with a copy of the Diffuse Map in a medium gray that corresponds to the average shininess/dullness of this material. Then add lighter grays for smoother, shinier, more reflective areas; and darker grays for duller, rougher, worn-out areas. The resulting image file looks similar to a grayscale version of the Diffuse Map.
- You can use colors in a Specular map to create certain reflective effects (fake iridescence, metallic effect).

## Seamless Tiled Textures



Tiles are a very simple, commonly used type of texture. When texturing a wide area (e.g. walls, floors), you don't create one huge texture – instead you tile a small texture repeatedly to fill the area.

A seamless texture is an image file that has been designed or modified so that it can be used as tiles: The right edge matches the left edge, and the top edge matches the bottom edge. The onlooker cannot easily tell where one starts and the next one ends, thus creating an illusion of a huge texture. The downside is that the tiling becomes painfully obvious when the area is viewed from a distance. Also you cannot use it on more complex models such as characters.

See also this tutorial on [How to make seamless textures in Photoshop](#).

## UV Maps / Texture Atlas

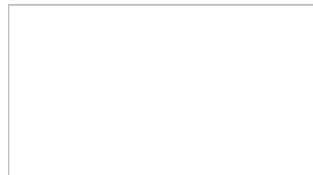


Creating a texture for a cube is easy – but what about a character with a face and extremities? For more complex objects, you design the texture in the same ways as a flat sewing pattern: One image file contains the outline of the front, back, and side of the object, next to one another. Specific areas of the flat texture (UV coordinates) map onto certain

areas of your 3D model (XYZ coordinates), hence the name UV map. Using UV Maps (also known as Texture Atlas), one model can have different textures on each side. You create one corresponding UV map for each texture.

Getting the seams and mappings right is crucial: You must use a graphic tool like Blender to create UV Maps (Texture Atlas) and store the coordinates correctly. It's worth the while to learn this, UV mapped models look a lot more professional.

## Environment Mapping



Environment Mapping or Reflection Mapping is used to create the impression of reflections and refractions in real time. It's faster (but less accurate) than the raytracing methods used in offline rendering applications.

You create a Cube Map to represent your environment; Sphere Maps are also possible, but often look distorted. Basically you give the environment map a set of images showing a "360° view" of the background scene – very similar to a skybox. The renderer maps the environment on the texture of the reflective surface, which results in an acceptable "glass/mirror/water" effect. Just like a skybox, the reflection map is static, so dynamic things (such as the player walking) are not part of the reflection. (!)

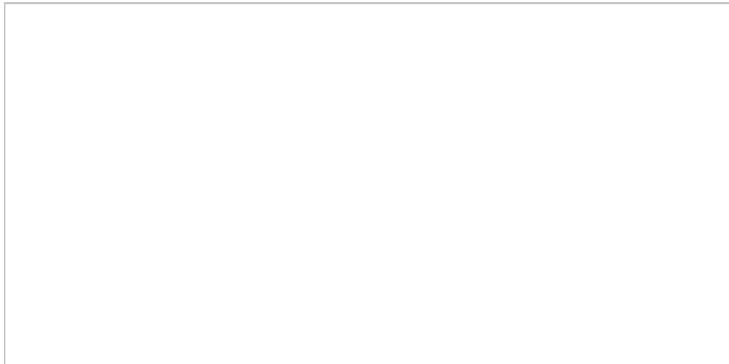
See also: [Water](#).

## MIP Map Texture

MIP Map means that you provide one texture in two or three resolutions in one file (MIP = "multum in parvo" = "many in one"). Depending on how close (or far) the camera is, the engine automatically renders a more (or less) detailed texture for the object. Thus objects look detailed at close up, but also look good when viewed from far away. Good for everything, but requires more time to create and more space to store textures. If you don't provide custom ones, the jMonkeyEngine creates basic MIP maps automatically as an optimization.

## Procedural Textures

A procedural texture is generated from repeating one small image, plus some pseudo-random, gradient variations (called Perlin noise). Procedural textures look more natural than static rectangular textures, and they look less distorted on spheres. On big meshes, their repetitiveness is much less noticeable than with tiled seamless textures. Procedural textures are ideal for irregular large-area textures like grass, soil, rock, rust, and walls. Use the [jMonkeyEngine SDK NeoTexture plugin](#) to create them.



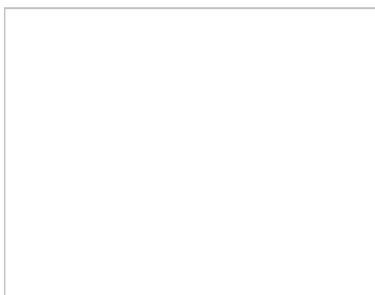
See also: [Creating Materials in Blender](#), [Blender: Every Material Known to Man](#)

## Animation

In 3D games, Skeletal Animation is used for animated characters, but in principle the skeleton approach can be extended to any 3D mesh (for example, an opening crate's hinge can be considered a primitive joint).

Unless you animate a 3D cartoon, realism of animated characters is generally a problem: Movement can look alien-like mechanical or broken, the character appears hollow, or as if floating. Professional game designers invest a lot of effort to make characters animate in a natural way, including [motion capture](#).

## Rigging and Skinning



An animated character has an armature: An internal skeleton (Bones) and an external surface (Skin). The Skin is the visible outside of the character and it includes clothing. The Bones are not visible and are used to interpolate (calculate) the morphing steps of the skin.

JME3, the game engine, only loads and plays your recorded animations. You must use a tool (such as Blender) to set up (rig, skin, and animate) a character.

1. **Rigging:** The Construction of a character's skeleton.
  - Create as few Bones as possible to decrease complexity.
  - Bones are connected in a parent-child hierarchy: Moving one bone can pull another bone with it (e.g. arm pulls hand).
  - Bones follow a certain naming scheme so the 3D engines know what's what.
2. **Skinning:** The association of individual bones with the corresponding skin sections.
  - Each Bone is connected to a part of the Skin. Animating the (invisible) Bone pulls the (visible) Skin with it.  
E.g. the thigh Bone is connected to the upper leg Skin.
  - One part of the Skin can be affected by more than one bone (e.g. knee, elbow).
  - The connection between bones and skin sections is gradual: You assign weights how much each skin polygon is affected by any bone's motion.  
E.g. when the thigh bone moves, the leg is fully affected, the hips joints less so, and the head not at all.
  - jMonkeyEngine supports hardware skinning (on the GPU, not on the CPU).
3. **Keyframe Animation:** A keyframe is one recorded snapshot of a motion sequence.
  - A series of keyframes makes up one animation.
  - Each model can have several animations. Each animation has a name to identify it (e.g. "walk", "attack", "jump").
  - You specify in your game code which keyframe animation to load, and when to play it.

What is the difference between animation (rigging, skinning, keyframes) and transformation (rotation, scaling, moving, "slerp")?

- Transformation is simpler than animation. Sometimes, transforming a geometry already makes it look like it is animated: For example, a spinning windmill, a pulsating alien ball of energy, moving rods of a machine. Transformations can be easily done with JME3 methods.
- Animations however are more complex and are encoded in a special format (keyframes). They distort the skin of the mesh, and complex series of motions be "recorded" (in external tools) and played (in JME3).

</div>

## Kinematics

- Forward kinematics: "Given the angles of all the character's joints, what is the position of the character's hand?"

- Inverse kinematics: “Given the position of the character's hand, what are the angles of all the character's joints?”

</div>

## Controller and Channel

In the JME3 application, you register animated models to the Animation Controller. The controller object gives you access to the available animation sequences. The controller has several channels, each channel can run one animation sequence at a time. To run several sequences, you create several channels, and run them in parallel.

</div>

## Artificial Intelligence (AI)

Non-player (computer-controlled) characters (NPCs) are only fun in a game if they do not stupidly bump into walls, or blindly run into the line of fire. You want to make NPCs “aware” of their surroundings and let them make decisions based on the game state – otherwise the player can just ignore them. The most common use case is that you want to make enemies interact in a way so they offer a more interesting challenge for the player.

“Smart” game elements are called artificially intelligent agents (AI agents). An AI agent can be used to implement enemy NPCs as well as trained pets; you also use them to create automatic alarm systems that lock doors and “call the guards” after the player triggers an intruder alert.

The domain of artificial intelligence deals, among other things, with:

- **Knowledge** – Knowledge is *the data* to which the AI agent has access, and on which the AI bases its decisions. Realistic agents only “know” what they “see and hear”. This implies that information can be hidden from the AI to keep the game fair. You can have an all-knowing AI, or you can let only some AI agents share information, or you let only AI agents who are close know the current state.

Example: After the player trips the wire, only a few AI guards with two-way radios start moving towards the player's position, while many other guards don't suspect anything yet.

- **Goal Planning** – Planning is about how an AI agent *takes action*. Each agent has the priority to achieve a specific goal, to reach a future state. When programming, you split the agent's goal into several subgoals. The agent consults its knowledge about the current state, chooses from available tactics and strategies, and prioritizes them. The

agent repeatedly tests whether the current state is closer to its goal. If unsuccessful, the agent must discard the current tactics/strategy and try another one.

Example: An agent searches the best path to reach the player base in a changing environment, avoiding traps. An agent chases the player with the goal of eliminating him. An agent hides from the player with the goal of murdering a VIP.

- **Problem Solving** – Problem solving is about how the agent *reacts to interruptions*, obstacles that stand between it and its goal. The agent uses a given set of facts and rules to deduct what state it is in – triggered by perceptions similar to pain, agony, boredom, or being trapped. In each state, only a specific subset of reactions makes sense. The actual reaction also depends on the agent's goal since the agent's reaction must not block its own goal!

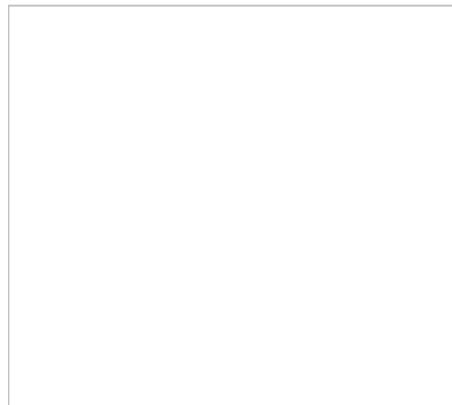
Examples: If player approaches, does the agent attack or conceal himself or raise alarm? While agent is idle, does he lay traps or heal self or recharge magic runes? If danger to own life, does the agent try to escape or kamikaze?

More advanced AIs can also learn, for example using neural networks.

There are lots of resources explaining interesting AI algorithms:

- [A\\* \(A-Star\) pathfinding for beginners](#)
- [A\\* \(A-star\) pathfinding theory](#)
- ["Z-Path" algorithm \(backwards pathfinding\)](#)
- [GOAP -- Goal-Oriented Action Planning](#)
- [Neuroph -- Java Neural Networks](#)
- ...

## Math



</div>

## Coordinates

Coordinates represent a location in a coordinate system. Coordinates are relative to the origin at (0,0,0). In 3D space, you need to specify three coordinate values to locate a point: X (right), Y (up), Z (towards you). Similarly, -X (left), -Y (down), -Z (away from you). In contrast to a vector (which looks similar), a coordinate is a location, not a direction.

</div>

## The Origin

The origin is the central point in the 3D world, where the three axes meet. It's always at the coordinates (0,0,0).

**Example:** `Vector3f origin = new Vector3f( Vector3f.ZERO );`

</div>

## Vectors

A vector has a length and a direction, like an arrow in 3D space. A vector starts at a coordinate (x1,y1,z1) or at the origin, and ends at the target coordinate (x2,y2,z2). Backwards directions are expressed with negative values.

**Example:**

```
Vector3f v = new Vector3f(17f , -4f , 0f); // starts at (0/0/0)
Vector3f v = new Vector3f(8f , 0f , 33f).add(new Vector3f(0f , -
```

## Unit Vectors

A *unit vector* is a basic vector with a length of 1 world unit. Since its length is fixed (and it thus can only point at one location anyway), the only interesting thing about this vector is its direction.

- `Vector3f.UNIT_X` = ( 1, 0, 0) = right
- `Vector3f.UNIT_Y` = ( 0, 1, 0) = up
- `Vector3f.UNIT_Z` = ( 0, 0, 1) = forwards
- `Vector3f.UNIT_XYZ` = 1 wu diagonal right-up-forwards

Negate the components of the vector to turn its direction, e.g. negating right (1,0,0) results in left (-1,0,0).

</div>

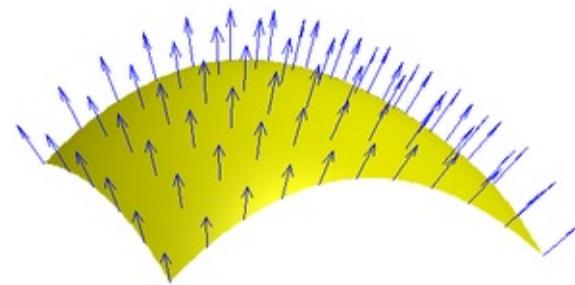
## Normalized Vectors

A *normalized vector* is a custom *unit vector*. A normalized vector is not the same as a *(surface) normal vector*. When you normalize a vector, it still has the same direction, but you lose the information where the vector originally pointed.

**Example:** You normalize vectors before calculating angles.

</div>

## Surface Normal Vectors



A surface normal is a vector that is perpendicular (orthogonal) to a plane. You calculate the Surface Normal by calculating the cross product.

</div>

## Cross Product

The cross product is a calculation that you use to find a perpendicular vector (an orthogonal, a “right angle” at 90°). In 3D space, speaking of an orthogonal only makes sense with respect to a plane. You need two vectors to uniquely define a plane. The cross product of the two vectors, `v1 × v2`, is a new vector that is perpendicular to this plane. A vector perpendicular to a plane is a called *Surface Normal*.

**Example:** The x unit vector and the y unit vector together define the x/y plane. The vector perpendicular to them is the z axis. JME can calculate that this equation is true:

```
(Vector3f.UNIT_X.cross(Vector3f.UNIT_Y)).equals(Vector3f.UNIT_Z) == true
```

</div>

## Transformation

Transformation means rotating (turning), scaling (resizing), or translating (moving) objects in 3D scenes. 3D engines offer simple methods so you can write code that transforms nodes.

Examples: Falling and rotating bricks in 3D Tetris.

</div>

## Slerp

Slerp is how we pronounce spherical linear interpolation when we are in a hurry. A slerp is an interpolated transformation that is used as a simple “animation” in 3D engines. You define a start and end state, and the slerp interpolates a constant-speed transition from one state to the other. You can play the motion, pause it at various percentages (values between 0.0 and 1.0), and play it backwards and forwards. [JavaDoc: slerp\(\)](#)

Example: A burning meteorite Geometry slerps from “position p1, rotation r1, scale s1” in the sky down to “p2, r2, s2” into a crater.

[Learn more about 3D maths here.](#)

</div>

## Game Developer Jargon

- [A Game Studio Culture Dictionary](#)

</div>

## 3D graphics Terminology Wiki book

- [http://en.wikipedia.org/wiki/User:Jreynaga/Books/3D\\_Graphics\\_Terms](http://en.wikipedia.org/wiki/User:Jreynaga/Books/3D_Graphics_Terms)

</div>

## **title: The Scene Graph and Other jME3 Terminology**

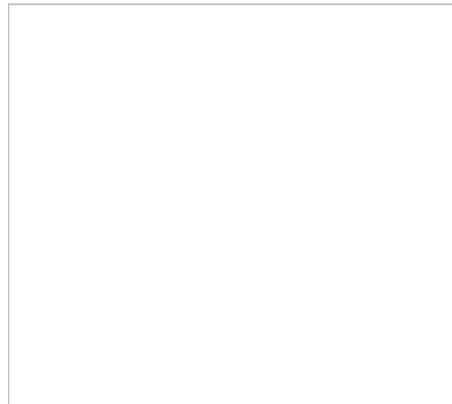
# **The Scene Graph and Other jME3 Terminology**

Before you start making games, make sure you understand general [3D Gaming terminology](#).

Second, if you are a beginner, we recommend our [Scene Graph for Dummies](#) presentation for a visual introduction to the concept of a scene graph.

Then continue learning about jME3 concepts here.

## **Coordinate System**



The jMonkeyEngine uses a right-handed coordinate system, just as OpenGL does.

The coordinate system consists of:

- The *origin*, a single central point in space.
  - The origin point is always at coordinate zero, in Java: `new Vector3f(0, 0, 0)` .
- Three *coordinate axes* that are mutually perpendicular, and meet in the origin.
  - The X axis starts left and goes right.
  - The Y axis starts below and goes up.
  - The Z axis starts away from you, and goes towards you.

Every point in 3D space is uniquely defined by its X,Y,Z coordinates. The three numeric coordinates express how many “steps” from each of the three axes a point is. The data type for all vectors in jME3 is `com.jme3.math.Vector3f`. All vectors are relative to the described coordinate system.

Example: The point `new Vector3f(3, -5, 1)` is 3 steps to the right, 5 steps down, and 1 towards you.

The unit of measurement (“one step”) in jME3 is the **world unit**, short: wu. Typically, 1 wu is considered to be one meter. As long as you are consistent throughout your game, 1 wu can be any distance you like.

For your orientation:

- The default camera's location is `Vector3f(0.0f, 0.0f, 10.0f)`.
- The default camera is looking in the direction described by the (so called) negative Z unit vector `Vector3f(0.0f, 0.0f, -1.0f)`.

This means the player's point of view is on the positive side of the Z axis, looking back, towards the origin, down the Z axis.

</div>

## How to move yourself through the 3D scene

When you play a 3D game, you typically want to navigate the 3D scene. Note that by default, the mouse pointer is invisible, and the mouse is set up to control the camera rotation!

By default, jME3 uses the following common navigation inputs

Game Inputs	Camera Motion	Player POV
Press the W and S keys	move the camera forward, and backward	you walk back and forth
Press the A and D keys	move the camera left and right	you step left or right
Press the Q and Y keys	move the camera up and down	you fly up and down
Move the mouse left-right	rotate the camera left/right	you look left or right
Move the mouse forwards-backwards	rotate up/down	you look at the sky or your feet

These default settings are called “WASD keys” and “Mouse Look”. You can customize [input handling](#) for your game. Sorry, but these settings work best on a QWERTY/QWERTZ keyboard.

## Scene Graph and rootNode

The *scene graph* represents your 3D world. Objects in the jME3 scene graph are called [Spatial](#)s. Everything attached to the parent `rootNode` is part of your scene. Your game inherits the `rootNode` object from the `SimpleApplication` class.



- *Attaching* a Spatial to the `rootNode` (or its child nodes) adds it to the scene;
- *Detaching* a Spatial from the `rootNode` (or its child nodes) removes it from the scene.

All objects in the scene graph are in a parent-child relationship. When you transform (move, rotate, scale) one parent, all its children follow.

The scene graph only manages the parent-child relationship of spatial. The actual location, rotation, or scale of an object is stored inside each Spatial.

</div>

## Spatial: Node vs Geometry

A Spatial can be transformed (in other words, it has a location, a rotation, and a scale). A Spatial can be loaded and saved as a .3jo file. There are two types of Spatial, *Nodes* and *Geometries*:

	<b>Spatial</b>	
<b>Purpose:</b>	A Spatial is an abstract data structure that stores transformations (translation, rotation, scale).	
	<b>Geometry</b>	<b>Node</b>
<b>Visibility:</b>	A visible 3-D object.	An invisible “handle” for a group of objects.
<b>Purpose:</b>	A Geometry represents the “look” of an object: Shape, color, texture, opacity/transparency.	A Node groups Geometries and other Nodes together: You transform a Node to affect all attached Nodes (parent-child relationship).
<b>Content:</b>	Transformations, mesh, material.	Transformations. No mesh, no material.
<b>Examples:</b>	A box, a sphere, player, a building, a piece of terrain, a vehicle, missiles, NPCs, etc...	The rootNode, the guiNode, an audioNode, a custom grouping node for a vehicle plus its passengers, etc.

## How to Use This Knowledge?

Before you start creating your game, you should plan your scene graph: Which Nodes and Geometries will you need? Complete the [Beginner tutorials](#) to learn how to load and create Spatial, how to lay out a scene by attaching, detaching, and transforming Spatial, and how to add interaction and effects to a game.

The [intermediate and advanced documentation](#) gives you more details on how to put all the parts together to create an awesome 3D game in Java!

## See also

- [Spatial](#) – More details about working with Nodes and Geometries
- [Traverse SceneGraph](#) – Find any Node or Geometry in the scenegraph.
- [Camera](#) – Learn more about the Camera in the scene.

[spatial](#), [node](#), [mesh](#), [geometry](#), [scenegraph](#), [rootnode](#)

</div>

---

## **title: What is updateGeometricState() and when do I need it?**

### **What is updateGeometricState() and when do I need it?**

Never, forget about this method and never use it again! Do not override, call or otherwise tamper with it.

## **title:**

This project is aimed to help beginners to understand JME basics.

You can get entire project with mercurial repository:

- <http://code.google.com/p/jme-simple-examples/source/checkout> - Simple Examples Repository.

The list of examples is here:

- [Simple Examples Code](#) - Simple examples by Mifth.
- [Asteroids Mini Game](#) - Asteroids Mini Game by Mark Schrijver.

## **title:**

[Publishing deal signed with Packt for jME3 Beginner's Book](#)

---

**title:**

**OpenCV**

**JavaCV**

**Hand gesture links:**

<http://www.movesinstitute.org/~kolsch/HandVu/HandVu.html>

## **title: WebStart (JNLP) Deployment**

# **WebStart (JNLP) Deployment**

When you [use the jMonkeyEngine SDK to deploy your application](#), you can configure the project to build files required for WebStart automatically. If you use another IDE, or work on the command line, use the following tips to set up WebStart correctly:

</div>

## **Problem Statement**

### **Problem:**

When running under WebStart, jMonkeyEngine may not have permission to extract the native libraries to the current directory.

### **Solution:**

You can instruct WebStart to load the native libraries itself using the JNLP file, and then instruct jME3 not to try to do so itself.

</div>

## **Simple way**

You can import the LWJGL JNLP extension directly into your extension, however be aware that your application will break whenever they update their jars. Simply add this line to your JNLP:

```
<extension name="lwjgl" href="http://lwjgl.org/webstart/2.7.1/exter
```

## **Reliable way**

### **Native jars**

You can download the LWJGL native jars from their site, or to ensure you're using the exact same version as bundled with your jME3 release, make your own:

```
mkdir tmp
cd tmp
jar xfv ../../jME3-lwjgl-natives.jar
cd native
for i in *; do
 cd $i
 jar cfv ../../native_$i.jar .
 cd ..
done
```

For Windows:

```
@echo off
md tmp
cd tmp
"%JDK_HOME%\bin\jar" -xfv ..\jME3-lwjgl-natives.jar
cd native
for /D %%i in ("*") do (
 cd %%i
 "%JDK_HOME%\bin\jar" -cfv ..\..\native_%%i%.jar .
 cd ..
)
cd ..
```

Remember to sign all the jar files and move them into the right place from the tmp directory.

</div>

## JNLP file

Add the following to your JNLP file:

```

<resources os="Windows">
 <j2se version="1.4+"/>
 <nativelib href="native_windows.jar"/>
 </resources>
<resources os="Linux">
 <j2se version="1.4+"/>
 <nativelib href="native_linux.jar"/>
 </resources>
<resources os="Mac OS X">
 <j2se version="1.4+"/>
 <nativelib href="native_macosx.jar"/>
 </resources>
<resources os="SunOS" arch="x86">
 <j2se version="1.4+"/>
 <nativelib href="native_solaris.jar"/>
 </resources>

```

</div>

## Set low-permissions mode

In your main() method, if running under WebStart, tell jME3 it is running in a low-permission environment so that it doesn't try to load the natives itself:

```

public static void main(String[] args)
{
 if (System.getProperty("javawebstart.version") != null) {
 JmeSystem.setLowPermissions(true);
 }

```

</div>

# SDK

## Developing for jMonkeyEngine SDK

*Note that all info is subject to change while jMonkeyEngine SDK is still in beta!*

In general, developing plugins for jMonkeyEngine SDK is not much different than creating plugins for the NetBeans Platform which in turn is not much different than creating Swing applications. You can use jMonkeyEngine SDK to develop plugins, be it for personal use or to contribute to the community.

If you feel like you want to make an addition to jMonkeyEngine SDK, don't hesitate to contact the jme team regardless of your knowledge in NetBeans platform development. For new plugins, the basic project creation and layout of the plugin can always be handled by a core developer and you can go on from there fleshing out the plugin. By using the Platform functions, your plugin feels more like a Platform application (global save button, file type support etc.).

### Creating plugins and components

- [Creating a plugin](#)
- [Creating components](#)
- [Building the jME SDK from scratch](#) (not necessary for plugin development, only for contributors)

### Extending jMonkeyEngine SDK

- [The Main Scene](#)
- [The Scene Explorer](#)
- [Projects and Assets](#)

### Recipes

- [Create a library plugin from a jar file](#)
- [Create a new or custom model filetype and loader](#)

### General Notes

- **Remember the scene runs on the render thread and most everything you do in**

**the plugin (button events etc.) runs on the AWT thread, always encapsulate calls to either side correctly via Callables/Runnables or register as an AppState to the SceneApplication to have an update() call by the render thread.**

- Although the scene can be accessed at any time via SceneApplication.getApplication() it is not recommended to modify the scene like that. Other plugins might be accessing the scene and updates will not be properly recognized. Use the sceneRequest object and the lookup of selected nodes and files to access things like the assetManager etc.
- It became a standard in jMonkeyEngine SDK to start the name of methods that execute directly on the OpenGL thread with “do” e.g “doMoveSpatial”, this makes identifying threading issues easier.
- The AssetManager of jme3 is threadsafe and can be used from any thread to load assets
- You can get access to the ProjectAssetManager via the Lookup of a JmeSpatial and other objects
- Use org.openide.filesystems.FileObject instead of java.io.File for file access, it always uses system-independent “/” path separators and has some more advanced functions that make file handling easier.
- You can get a file from a string using  
`Repository.getDefault().getDefaultFileSystem().getRoot().getFileObject("aaa/bbb/ccc/whatever");`
- You can convert a regular java File to a FileObject and vice versa using  
org.openide.filesystems.FileUtil
- If you have problems with unresolved classes, check if all needed Libraries are registered in the settings of your Project. To find out which library contains a certain class, just enter the name in the library search field.

## Teminology used here

- A “plugin” is anything you can tick in the plugin manager of the SDK. It can contain editors, simple “Java SE Libraries” that you can add to your projects as jar files and other things like project templates etc.
- A “module” is the project type that allows you to create plugins, strictly speaking all plugins are modules but there can be modules that are never shown in the plugin list and only exist as dependencies of other modules.
- A “library” is an entry for a jar file (and optionally sources and javadocs) which can be added to a SDK project to be used and distributed with it
- An “extension” is a generic name for stuff that extends the jME engine, like pathfinding algorithms or anything that can be used at the game runtime..

So if you have some cool code that others can use in their games too, you would make your extension a library by creating a module that the users can download as a plugin :)

## Handy things in jMonkeyEngine SDK Core

- com.jme3.gde.core.scene.controller
  - AbstractCameraController → A basic camera control for plugins, used by SimpleSceneComposer and “View Model”
  - SceneToolController → A basic controller for having selection, cursor etc. displayed in the scene, used by SimpleSceneComposer
- com.jme3.gde.core.scene
  - OffViewPanel → A panel that renders a 3d scene in a preview and displays it in a lightweight swing panel
- com.jme3.gde.core.util
  - DataObjectSaveNode → Allows enabling the save all button by using any file and implementing the SvaeCookie yourself.

Learn more about NetBeans Plugin Development at <http://platform.netbeans.org>

Also check out this Essential NetBeans Platform Refcard:

<http://refcardz.dzone.com/refcardz/essential-netbeans-platform>

[documentation](#), [sdk](#), [contribute](#)

</div>

## **title: Using Blender as a Intermediator Between 3dMax and the jMonkeyEngine SDK**

### **Using Blender as a Intermediator Between 3dMax and the jMonkeyEngine SDK**

The jMonkeyEngine SDK supports .blend files and can convert them to jMonkeyEngine's .j3o format. This means you can use Blender to convert, for example, a 3dMax file to .j3o format.

</div>

#### **Importing the .3ds file to Blender**

I'm using the blender 2.59 at this tutorial, but if you blender 2.49b, no problem ;). After you saved your .3ds file in 3dmax, open the blender, delete the default cube, and import your .3ds file via File—>Import—>3D Studio.

</div>

#### **Saving the .blend file**

Now save your .blend file so you can load it into the jMonkeyEngine SDK.

</div>

#### **Importing the .blend file to the SDK by using the ModelImporter and BlenderSupport plugins**

Click on Import Model button and then click on Open Model button to open the .blend file. Click next, select the checkbox to import a copy from .blend file, and click finish.



</div>

## Edit your model in SceneComposer and "VOILA"

As you can see, the .blend model was automatically converted to .j3o binary format. Now, you are able to edit it in SceneComposer ;D.



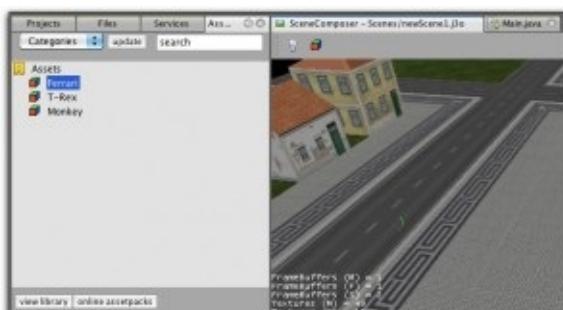
</div>

## **title: jMonkeyEngine SDK: AssetPacks and AssetPack Browser**

# **jMonkeyEngine SDK: AssetPacks and AssetPack Browser**

AssetPacks are a way to package jME3 compatible assets (like models, textures, sounds and whole scenes!) into a package that contains publisher info, license info, descriptions etc. for all of the assets. An AssetPack basically consists of an `assetpack.xml` file that describes the content and an `assets` folder that contains the content. The integrated browser in the jMonkeyEngine SDK allows you to add the assets from installed AssetPacks to any jme3 project scene you are working on.

## **The AssetPack Browser**



### **Browsing Assets**

The AssetPack browser in jMonkeyEngine SDK makes browsing the installed AssetPacks easy. Browse categories, search for tags and find the right asset for your project. When you have found it, you can add it with one click to your current scene. The AssetPack manager will automatically copy all needed textures, sounds etc. to your projects assets folder.

You can also browse a selection of online assetpacks that are available on [jMonkeyEngine.org](http://jMonkeyEngine.org) for download and install them to your jMonkeyEngine SDK's AssetPack browser.

The AssetPack browser uses a predefined directory to store the AssetPacks which is also used for new AssetPack projects. You can see and change the folder path in the AssetPack preferences (jMonkeyEngine SDK→Settings).

</div>

## Adding Assets to Your Scene

To preview a model from the browser, right-click it and select “Preview Asset”

To add a model from the AssetPack browser to a scene do the following:

1. Create a new scene (j3o) file or use an existing one
2. Open it in the SceneComposer by right-clicking it and selecting “Edit in SceneComposer”
3. Select the root node of the scene (or another node you want to add the model to)
4. Right-click a model in the AssetBrowser and select “Add to SceneComposer”

The model will be added to your scene and all needed texture files will be copied to your projects assets folder.

## Create Your Own AssetPack Project

AssetPack projects are a way to create your own AssetPacks, either for personal use or for publishing. With this project type in jMonkeyEngine SDK you can create a library of assets on your computer that you can use in any of your projects via the AssetPack browser.

Editing of asset and project info, adding of new assets and setting their properties, everything you need to create and publish your AssetPacks is there.

1. Choose “File → New Project” from the menu
2. Choose “AssetPack Project”
3. Fill in the project info and press “finish”

You can access and change the project properties by right-clicking the project and selecting “Properties”.

## Add Your Own Assets



To add new assets to your AssetPack do the following:

1. Right-Click the “Assets” node of the AssetPack project
2. Select “Add Asset..”
3. Specify the info about your asset and press Next
4. Press the “add files” button and select all files belonging to your asset
5. Select the “main” checkbox for the main model file if your asset is a model file
6. Change the asset paths and types if needed and press “finish”

The global asset type can be “model”, “scene”, “texture”, “sound”, “shader” or “other”

With the “model” or “scene” types, the AssetPack browser will try to load and add a model file from the selected assets when the user selects “Add to SceneComposer”. Specify the “load this model with material” flag for the model file that should be loaded via the AssetManager, you can also specify multiple mesh or scene files to be loaded. All texture and other needed files will be copied to the users project folder.

On the “Add Files” page you define the path of the files in the AssetPack. The importer tries to generate a proper path from the info entered on the first page. Note that for j3o binary models, the texture paths have to be exactly like they were during conversion. The given paths will also be used when copying the data to the users “assets” folder.

With the “add files” button you can open a file browser to select files from your harddisk that will be copied into the `assets/` folder of the AssetPack project. With the “add existing” button you can add a file that’s already in your AssetPack assets folder to a new asset item. This way you can reuse e.g. textures for asset items or make items for an existing collection of asset files that you copied to the projects assets folder.



You can specify a specific material to be used for the single mesh/scene files, just select it in the dropdown below the mesh or scene file. If you don't select a material file here, the first found material file is used or none if none is found.

If the material file is an Ogre material file (.material) it will be used for loading an Ogre scene or mesh file. If it is a jMonkeyEngine3 material file (.j3m), it will be applied to the mesh regardless of model type. Note that for j3o models you don't need material files as the material is stored inside the j3o file.

In your AssetPack Project, right-click each asset and select “preview asset” to see your model. Verify that it looks correctly, because then it should work for other users as well. You can change the single assets properties in the properties window after you have added them. Just select an asset and open the properties window (Windows→Properties).

Supported formats for models (main files) are:

1. OgreXML .mesh.xml / .scene
2. Wavefront .obj
3. jMonkeyEngine3 .j3o
4. Blender .blend (unpack textures)

## AssetPack Publishing



You can publish your AssetPacks either as a zip file or directly to [jmonkeyengine.org](http://jmonkeyengine.org), using your website user name and login. **This means other jMonkeyEngine SDK users can download your AssetPacks and install them to their local database right off the AssetPack online packages browser.**

To make sure you can upload, you have to be registered on [jmonkeyengine.org](http://jmonkeyengine.org), and have to enter your login info in the AssetPack preferences: jMonkeyEngine SDK→Options→Asset Packs first.

- Right-Click your AssetPack project in the SDK and select “Publish AssetPack...”
- Check the description etc. settings, and press “Next”.
- Select the checkbox for online and/or local publishing and press “finish”.

</ol>

[documentation](#), [sdk](#), [asset](#)

</div>

# **title: Blender importer for jMonkeyEngine 3**

## **Blender importer for jMonkeyEngine 3**

### **Introduction**

Importing models to any game engine is as important as using them. The quality of the models depends on the abilities of the people who create it and on the tools they use. Blender is one of the best free tools for creating 3D environments. Its high amount of features attract many model designers. So far jMonkeyEngine used Ogre mesh files to import 3D data. These files were created by the python script that exported data from blender. It was important to have always the latest version of the script that is compatible with the version of blender and to use it before importing data to jme. Now we have an opportunity to simplify the import process by loading data directly from blender binary files: \*.blend.

Before you try to import models, make sure you [created them properly](#).

</div>

### **Usage**

To use it in your game or the SDK you should follow the standard asset loading instructions. By default a BlenderModelLoader is registered with your assetManager to load blend files. This means you can load and convert .blend model files to .j3o format, just like any other supported model format.

### **Currently supported features**

1. Loading scene (only the current scene is loaded and imported as a node)
2. Loading mesh objects.
  - Meshes are split into several geometries when they have several materials applied.
  - All faces are stored as triangles (even if blender uses quads).
  - The mesh is 'Smooth' aware.
  - User defined UV coordinates are read.
  - Loading BMesh is supported.

3. Loading textures.

- Both image and generated textures are imported.
- Textures influence is supported ('Influence' tab in blender 2.5+ and 'Map to' in 2.49).
- Map input is not yet fully supported (currently working on it ; ) so please use UV-mapping for all kinds of textures.

4. Image textures.

- Textures can be loaded from: png, jpg, bmp, dds and tga.
- Both textures stored in the blender file and the outside are loaded (the outside textures need a valid path).
- Image textures are stored as Texture2D.

5. Generated textures.

- All generated textures can be loaded except: VoxelData, EnviromentMap and PointDensity.
- Feel free to use colorbands.
- Generated textures are 'baked' into 2D textures and merged to create one flat texture. They can be freely merged with image textures.
- Generated textures can be used as normal maps (but this looks poor when large amount of small triangles is used; increasing generated texture ppu in blender key might help a little)

6. Loading materials.

- Materials are loaded and attached to geometries.
- Because jMonkeyEngine supports only one material for each Mesh, if you apply several materials to one object – it will be split into several meshes (but still in one node).
- Several kinds of input mapping is supported: UV maps, Orco and Nor; all projection types for 2D textures, XYZ coordinates mapping.

7. Loading animations.

- Bone animations and object animations are supported.
- Armatures are imported as Skeleton. Constraint loading is not fully supported so use it carefully.
- Only assigning vertices to bones is at the moment supported so do not use bones' envelopes.

8. Loading modifiers.

- i. Array modifier
- ii. Mirror modifier
- iii. Armature modifier (see loading animations)
- iv. Particles modifier (see loading particles)
- More will come with time.

9. Constraints loading

- Constraints are basicly supported but they do not work the way I'd like it. So feel free to experiment with it. I will create another post when I get it to work properly.

## 10. Particles loading.

- Some features of particles loading is supported. You can use only particle emitters at the moment.
- You can choose to emit particles from vertices, faces or the geometry's convex hull (instead of volume).
- Currently Newtonian Physics is only supported.
- It was mostly tested for blender 2.49 (so I'm not 100% sure about its use in blender 2.5+).

## 11. Using sculpting.

- This should work quite well for now :).

## 12. Importing curves.

- Both bezier and NURBS curves are supproted.
- Feel free to use bevel and taper objects as well ;)

## 13. Importing surfaces

- NURBS surface and sphere can be imported.

## 14. Importing sky

- loading world's horizon color as a background color if no sky type is used
- loading sky without the texture
- loading textured sky (including both generated and normal textures)

# Planned features.

1. Full support for scale and offset in texture input mapping.
2. Full support for bone and object constraints.
3. More modifiers loaded.
4. Loading texts.
5. Loading meta objects (if jme will support it ;) ).

# Known bugs/problems.

1. RGB10 and RGB9E5 texture types are not supported in texture merging operations (which means that you can use this as a single texture on the model, but you should not combine it with other images or generated textures).
2. If an armature is attached to a mesh that has more than one material the vertices of the mesh might be strongly displaced. Hope to fix that soon.

# Using BlenderLoader instead of BlenderModelLoader

You have two loaders available.

- BlenderLoader that loads the whole scene. It returns an instance of LoadingResults that contains all the data loaded from the scene.

```
public static class LoadingResults extends Spatial {
 /** Bitwise mask of features that are to be loaded. */
 private final int featuresToLoad;
 /** The scenes from the file. */
 private List<Node> scenes;
 /** Objects from all scenes. */
 private List<Node> objects;
 /** Materials from all objects. */
 private List<Material> materials;
 /** Textures from all objects. */
 private List<Texture> textures;
 /** Animations of all objects. */
 private List<AnimData> animations;
 /** All cameras from the file. */
 private List<Camera> cameras;
 /** All lights from the file. */
 private List<Light> lights;
 /** Access Methods goes here. */
}
```

- BlenderModelLoader loads only the model node and should be used if you have a single model in a file.

To register the model do the following:

```
assetManager.registerLoader(BlenderLoader.class, "blend");
```

or

```
assetManager.registerLoader(BlenderModelLoader.class, "blend");
```

- The last thing to do is to create a proper key.

You can use com.jme3.asset.BlenderKey for that. The simplest use is to create the key with the asset's name. It has many differens settings descibing the blender file more precisely, but all of them have default values so you do not need to worry about it at the beggining. You can use ModelKey as well. This will give the same result as using default BlenderKey.

## How does it work?

BlenderLoader (as well as BlenderModelLoader) is looking for all kinds of known assets to load. It's primary use is of course to load the models within the files. Each blender object is imported as scene Node. The node should have applied textures and materials as well. If you define animations in your BlenderKey the animations will as well be imported and attached to the model.

Here is the list of how blender features are mapped into jme.

Blender	jMonkeyEngine3	Note
Scene	Node	
Object	Node	
Mesh	List<Geometry>	One mesh can have several materials so that is why a list is needed here.
Lamp	Light	
Camera	Camera	
Material	Material	
Texture	Texture	
Curve	Node	Node with Curve as its mesh
Surface	Node	The surface is transformed to the proper mesh

Using BlenderLoader can allow you to use blend file as your local assets repository. You can store your textures, materials or meshes there and simply import it when needed. Currently blender 2.49 and 2.5+ are supported (only the stable versions). Probably versions before 2.49 will work pretty well too, but I never checked that :)

## Notes

I know that the current version of loader is not yet fully functional, but belive me – Blender is a very large issue ;) Hope I will meet your expectations.

Be mindful of the result model vertices amount. The best results are achieved when the model is smooth and has no texture. Then the vertex amount is equal to the vertex amount in blender. If the model is not smooth or has a generated texture applied then the amount of vertices is 3 times larger than mesh's triangles amount. If a 2d texture is applied with UV mapping then the vertex count will vary depending on how much the UV map is fragmented.

When using polygon meshes in blender 2.5 and newer, better add and apply the triangulation modifier (if available in your version) or save the file with conversion from polygons to triangles. Even though the importer supports loading of polygons as the mesh faces, if your face isn't convex, the results might contain errors.

Not all modifiers are supported. If your model has modifiers and looks not the way you want in the jme scene - try to apply them and load again.

Cheers, Marcin Roguski (Kaelthus)

P.S. This text might be edited in a meantime if I forgot about something ;)

---

See also:

- [3DS to Blender to j3o](#)

[documentation](#), [sdk](#), [tool](#), [asset](#)

</div>

# title: Building jMonkeyEngine SDK Yourself

## Building jMonkeyEngine SDK Yourself

### In the jMonkeyEngine SDK or NetBeans

#### Plugins

Make sure all necessary plugins are installed and active:

- From the menu bar, select `Tools` → `Plugins` to open a `Plugins` dialog.
- Select the `Available Plugins` tab.
- Sort the plugin list by category.
- Check the `Install` checkbox for any plugins in the `Developing NetBeans` category.
- Click on the `Install` button and complete the wizard.
- Select the `Installed` tab.
- Activate any inactive plugins in the `Developing NetBeans` category.
- Close the `Plugins` dialog.

#### Checkout

- From the menu bar, select `Team` → `Git` → `Clone` to open a `Clone Repository` dialog.
- In the `Repository URL:` textbox, type `https://github.com/jMonkeyEngine/jmonkeyengine`.
- Clear the `User:` and `Password:` textboxes.
- Click on the `Next` button.
- Choose the remote branch, typically “master”, and click the Next button.
- Click on the `Browse...` button to the right of the `Parent directory:` textbox to open a `Browse Local Folder` dialog.
- Select an empty or non-existent directory (folder) where you want to save the jMonkeyEngine sources on your hard drive.
- Click the `Finish` button.

Project checkout may take awhile. Do not proceed before the task is complete.

#### Build

On the commandline:

1. `cd jmonkeyengine`
2. `./gradlew build`

From the SDK:

- From the menu bar, select `File` → `Open project...` to open an `Open Project` dialog.
- Highlight the free-form project folder you used for checkout and click on the `Open Project` button.
  - A new free-form project named `jME3-SDK` should appear in the SDK's `Projects` window.
- Build the free-form project by right-clicking on it and selecting `Build`.
- Wait until the `jME3-SDK (build)` task is complete. At one point, a web browser window may open.
- From the menu bar, select `File` → `Open project...` to open an `Open Project` dialog.
- Browse into the free-form project folder, highlight the `sdk` node, and click on the `Open Project` button.
  - A number of new projects should appear in the SDK's `Projects` window.

Building the main freeform project copies the jME3 libraries to the sdk, so you should run its build script each time jME3 changes.

## Run

- In the SDK's `Projects` window, right-click on the `jMonkeyEngine SDK` project and select `Run`.
  - The SDK you just built will start executing in a new window.

[documentation](#), [sdk](#), [builds](#), [project](#)

</div>

# title: jMonkeyEngine SDK: Code Editor and Palette

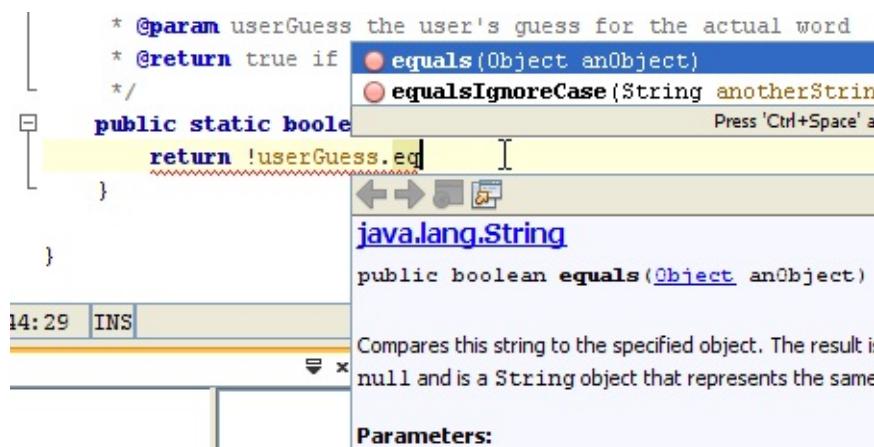
## jMonkeyEngine SDK: Code Editor and Palette

The Source Code Editor is the central part of the jMonkeyEngine SDK. This documentation shows you how to make the most of the jMonkeyEngine SDK's assistive features.

Note: Since the jMonkeyEngine SDK is based on the NetBeans Platform framework, you can learn about certain jMonkeyEngine SDK features by reading the corresponding NetBeans IDE tutorials (in the “see also links”).

## Code Completion and Code Generation

While typing Java code in the source code editor, you will see popups that help you to write more quickly by completing keywords, and generating code snippets. Additionally, they will let you see the javadoc for the classes you are working with.



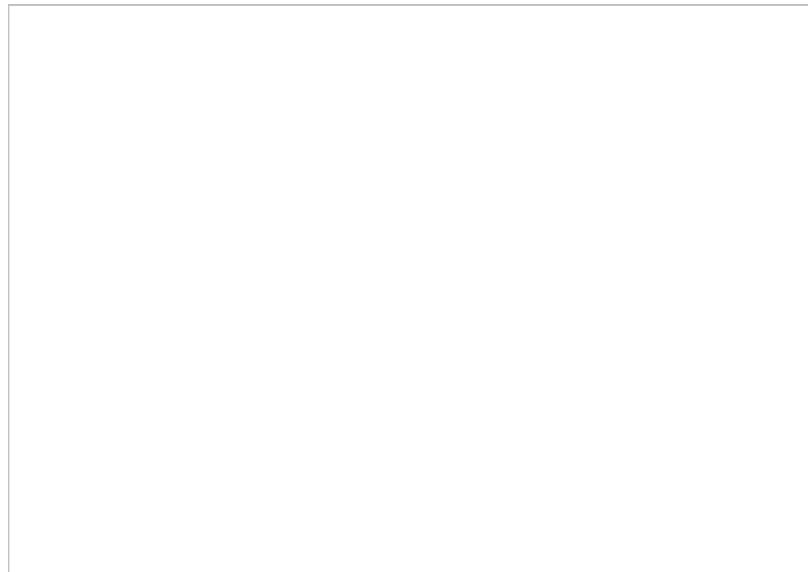
### Code Completion

- Complete keyword / method / variable: **Ctrl-Space**  
Alternatively you can also use **Ctrl-\**.
  - Customize Code Completion options: Tools > Options > Editor > Code Completion
- Show expected parameters of this method in a tooltip: **Ctrl-P**
- Complete any string (even non-Java) that has been used before: **(Shift-)Ctrl-K**

## Code Generation

- Auto-fix import statements: **Ctrl-Shift-I**
- Auto-generate getters/setters, try/catch, equals/hashCode: **Alt-Insert**
  - Customize code completion: Choose Tools > Options > Editor > Code Completion
- Auto-generate common code snippets such as loops, declarations, println, by typing the **template name + TabKey**
  - Customize code templates: Choose Tools > Options > Editor > Code Templates
- Rename, move, or introduce methods, fields, and variables, without breaking the project: **Refactoring menu**

## Semantic and Syntactic Coloring



The text color in the editor gives you important hints how the compiler will interpret what you typed, even before you compiled it.

Examples:

- Java keywords are **blue**, variables and fields are **green**, parameters are **orange**.
- **Strikethrough** means deprecated method or field.
- **Gray underline** means unused variable or method.
- Place the caret in a method or variable and all its occurrences are marked **tan**.
- Place the caret in a method's return type to highlight all exit points
- and many more...

To customize Colors and indentation:

- Tools > Options > Editor > Formatting.
- Tools > Options > Fonts and Colors.

# Editor Hints and Quick Fixes (a.k.a. Lightbulbs)

Editor hints and quick fixes show as lightbulbs along the left edge of the editor. They point out warnings and errors, and often propose useful solutions!

- Execute a quick fix: Place the caret in the line next to the lightbulb and press **Alt-Enter** (or click the lightbulb)
  - Customize hints: Choose Tools > Options > Editor > Hints.

## Javadoc

- Place the caret above a method or a class that has no Javadoc, type `/**` and press Enter: The editor generates skeleton code for a Javadoc comment.
- Right-click the project in the Projects window and choose Generate Javadoc.
- Right-click a file and choose Tools > Analyze Javadoc

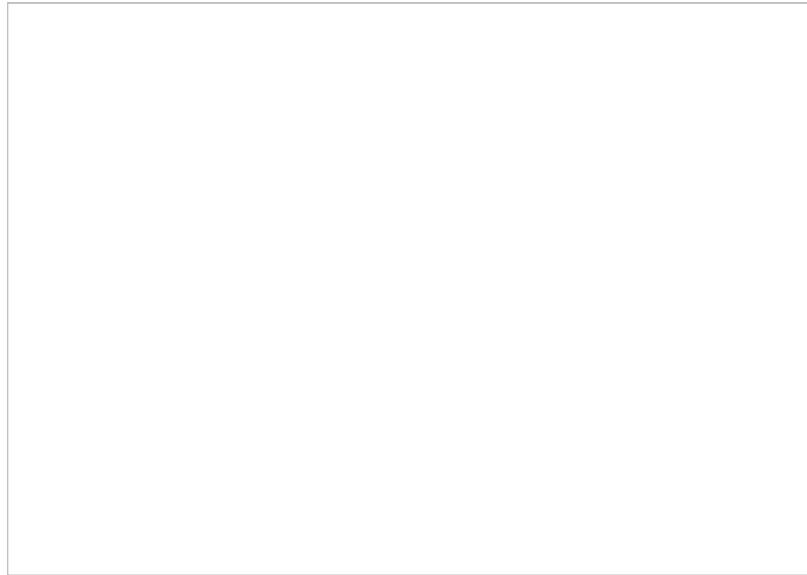
To display a javadoc popup in the editor, place the caret in a line and press **Ctrl-Space** (Alternatively use **Ctrl-l**).

- If the javadoc popup doesn't work, make certain that
  - You have the Java JDK documentation installed and set up: Tools > Java Platforms
  - You downloaded and set up javadoc for third-party libraries: Project properties > Libraries > Edit

## Navigating the jME3 Source

When the JavaDoc does not deliver enough information, you can have a look at the source of every method or object of jME3 that you use. Just right-click the variable or method, select “Navigate > Go to source..” and an editor will open showing you the source file of jME3.

## Palette



Choose Windows > Palette to open the context-sensitive Palette. The jMonkeyEngine SDK provides you with jme3 code snippets here that you can drag and drop into your source files.

- Examples: Node and Model creation code snippets.

Tip: Choose Tools > Add to Palette... from the menu to add your own code snippets to the Palette. (not available yet in beta build)

## Keyboard Shortcuts

Keyboard Shortcuts save you time when you need to repeat common actions such as Build&Run or navigation to files.

- Go to File: **Alt-Shift-O**
- Go to Type: **Ctrl-O**
- Open in Projects / Files / Favorites window: **Ctrl-Shift-1 / 2 / 3**
- Build&Run the main class of the Project: **F6**
- Run the open file: **Shift-F6**
- Switch to Editor / Projects / Files / Navigator: **Ctrl-0 / 1 / 3 / 7**
- Indent code: **Ctrl-Shift-F**

By default, jMonkeyEngine uses the same [Editor Shortcuts](#) as the NetBeans IDE, but you can also switch to an Eclipse Keymap, or create your own set.

- Customize keyboard shortcuts: Tools > Options > Keymap

## Tips and Tricks

- To browse the physical file structure of your project, use the Files window: **Ctrl-2**

- To open a file that is not part of a Java project, add it to the Favorites window: **Ctrl-3**
  - If you cannot find a particular menu item or option panel, use the IDE Search box in the top right! **Ctrl-i**
  - If a code block, class, or javadoc is quite long and you don't want to scroll over it, click the +/- signs to collapse (fold) the code block temporarily.
  - Press **F1** for Help
- 

## See also

- [Code Assistance](#)

[documentation](#), [sdk](#), [editor](#)

</div>

---

## **title: jMonkeyEngine -- The Comic Book**

# **jMonkeyEngine -- The Comic Book**

“Every good open-source project publishes documentation in comic book format.” – zathras



---

See also: [Main SDK Documentation Page](#), [Main Engine Documentation Page](#)

[documentation](#), [tool](#), [sdk](#)

</div>

## **title: jMonkeyEngine SDK: Debugging, Profiling, Testing**

# **jMonkeyEngine SDK: Debugging, Profiling, Testing**

Debugging, testing and profiling are important parts of the development cycle. This documentation shows you how to make the most of the jMonkeyEngine SDK's assistive features.

Since the jMonkeyEngine SDK is based on the NetBeans IDE and the NetBeans Platform, you can learn about certain jMonkeyEngine SDK features by reading the corresponding NetBeans IDE tutorials (in the “see also links”).

</div>

## **Testing**

The jMonkeyEngine SDK supports the JUnit testing framework. It is a good practice to write tests (assertions) for each of your classes. Each test makes certain this “unit” (e.g. method) meets its design and behaves as intended. Run your tests after each major change and you immediately see if you broke something.

### **Creating Tests**

1. Right-click a Java file in the Projects window and choose Tools > Create JUnit Tests.
2. Click OK. The jMonkeyEngine SDK creates a JUnit test skeleton in the Test Package directory.
3. The body of each generated test method is provided solely as a guide. In their place, you need to write your actual test cases!
4. You can use tests such as `assertTrue()`, `assertFalse()`, `assertEquals()` , or `assert()` .
  - The following example assertions test an addition method: `assert( add(1, 1) == 2); assertTrue( add(7, -5) == add(-5, 7) )...`
5. “Ideally”, you write a test case for every method (100% coverage).

**Tip:** Use the Navigate menu to jump from a test to its tested class, and back!

## Running Tests

1. Run one or all tests:
  - Right-click the class in the Projects window and Choose Test File, or
  - Right-click the project and select Test to run all tests.
2. Check the Test window to see successful tests (green) and failures (red).
3. If a test fails that has succeeded before, you know that your latest changes broke something!

Using unit tests regularly allows you to detect side-effects on classes that you thought were unaffected by a code change.

See also:

- [Writing JUnit Tests](#)
- <http://www.junit.org>
- [Java Assertions](#)

## Debugging

In the jMonkeyEngine SDK, you have access to a debugger to examine your application for errors such as deadlocks and NullPointerExceptions. You can set breakpoints, watch variables, and execute your code line-by-line to identify the source of a problem.

1. First, you set breakpoints and/or watches before the problematic lines of code where you suspect the bug.
  - If you want to watch a variable's value: Right-click on a variable and select New Watch from the context menu.
  - If you want to step through the execution line by line: Right-click on a line and choose Toggle Line Breakpoint; a pink box appears as a mark.
2. Choose “Debug > Debug Main Project” to start a debugger session for the whole project. Or, right-click a file and select Debug File to debug only one file.
3. The application starts running normally. If you have set a breakpoint, the execution stops in this line. Debugger windows open and print debugger output.
4. You can do many things now to track down a bug:
  - Inspect the values of local variables.
  - Use the Step buttons in the top to step into, out of, and over expressions while you watch the execution.
  - Navigate through your application's call stack. Right-click on threads to suspend or resume them.
  - Choose Debug > Evaluate Expression from the menu to evaluate an expression.
  - Move the mouse pointer over a variable to inspect its value in a tooltip.

- Inspect the classes loaded on the heap and the percentage and number of object instances. Right-click a class in the Loaded Classes window and choose Show in Instances view (JDK 6 only).
  - And more...
5. To stop debugging, choose Debug > End Debugger Session from the menu.

## Profiling

The profiler tool is used to monitor thread states, CPU performance, and memory usage of your jme3 application. It helps you detect memory leaks and bottlenecks in your game while it's running.

### Installing the Profiler

If you do not see a Profiler menu in the jMonkeyEngine SDK, you need to download the Profiler plugin first.

1. Open the Tools > Plugins menu, and go to the “Available plugins” tab
2. Find the “Java Profiler” plugin (“Java SE” category) and check the Install box.
3. Click the install button and follow the instructions.
4. When you start the profiler for the first time, you are prompted to run a calibration once. Click OK in the “Profiler integration” dialog to complete the installation process.

### Monitoring and Analyzing

1. Choose Profile Project from the Profile menu.
2. Select one of three tasks:
  - **Monitor Application** – Collect high-level information about properties of the target JVM, including thread activity and memory allocations.
  - **Analyze CPU Performance** – Collect detailed data on application performance, including the time to execute methods and the number of times the method is invoked.
  - **Analyze Memory Usage** – Collect detailed data on object allocation and garbage collection.
3. Click Run. Your application starts and runs normally.
4. Use the Profiling window to track and collect live profiling results while your application is running.

### Comparing Snapshots

Click the Take Snapshot button to capture the profiling data for later!

- You can store and view snapshots in the Profiling window.
- Choose Compare Snapshots from the profiler window to compare two selected snapshots

## Using Profiling Points

Profiling points are similar to debugger breakpoints: You place them directly in the source code and they can trigger profiling behaviour when hit.

- Open a class in the browser, right-click in a line, and select Profiling > Insert Profiling Point to add a profiling point here.
- Use Profiling points if you need a trigger to reset profiling results, take a snapshot or heap dump, record the timestamp or execution time of a code fragment, stop and start a load generator script (requires the load generator plugin).
- Open the Profiling Points window to view, modify and delete the Profiling Points in your projects.

See also:

- [Introduction to Profiling Java Applications \(netbeans.org\)](#)
- [Using Profiling Points \(netbeans.org\)](#)

[documentation](#), [sdk](#), [tool](#)

</div>

## title: Default Build Script

# Default Build Script

If you use jMonkeyEngine libraries together with the jMonkeyEngine SDK (recommended) then you benefit from the provided build script. Every new project comes with a default Ant script. The toolbar buttons and clean/build/run actions in the jMonkeyEngine SDK are already pre-configured.

## Default Targets

The build script includes targets for the following tasks:

- clean – deletes the built classes and executables in the dist directory.
- jar – compiles the classes, bundles the assets, builds the executable JAR in the dist directory, copies libraries.
- run – builds and runs the executable JAR (e.g. for the developers to test it).
- javadoc – compiles javadoc from your java files into html files.
- debug – used by the jMonkeyEngine SDK Debugger plugin.
- test – Used by the jMonkeyEngine SDK JUnit Test plugin.

You can call these targets on the command line using default Ant commands:

```
ant clean
ant jar
ant run
```

We recommended to use the user-friendly menu items, F-keys, or toolbar buttons in the jMonkeyEngine SDK to trigger clean, build, and run actions.

</div>

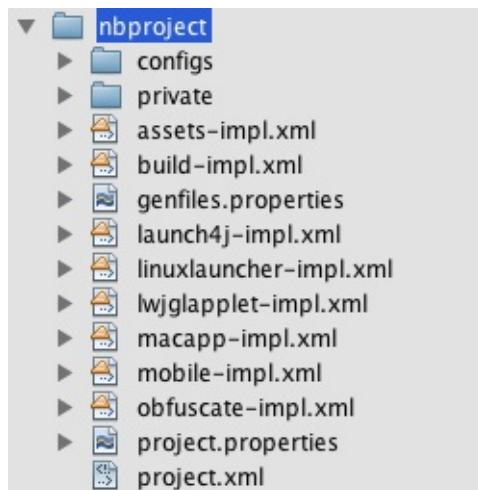
## Browsing the Build Script

To see the build script and the predefined tasks

1. Open your project in the jMonkeyEngine SDK if it isn't already open (File > Open

- Project...)
2. Open the Files window (Window > Files)
  3. Open the project node. You see `build.xml` listed.
    - i. Double-click `build.xml` to see how the jme3-specify build targets were defined.  
You typically do not need to edit the existing ones, but you can.
    - ii. Click the triangle next to `build.xml` to see all targets.
      - i. Double-click a target in the Files window, or the Navigator, to see how the target was defined.  
You will notice that the file `nbproject/build-impl.xml` opens. It contains very generic targets that you typically will never need to edit. Note that `build.xml` includes `build-impl.xml`!

## Adding Custom Targets



The build script is a non-proprietary Apache Ant script. It will work out-of-the-box, but if necessary, you can extend and customize it.

Read the comments in `build.xml`, they explain how to override targets, or extend them, to customize the build process without breaking the existing functionality.

Additionally, you can manually override the targets in the `*-impl.xml` files that are created when you change the deployment settings:

- `linuxlauncher-impl.xml`,
- `macapp-impl.xml`,
- `mobile-impl.xml`,
- `jnlp-impl.xml`, etc.

Simply copy&paste a target from these files into the main `build.xml` and that will be run instead with all modifications.

Don't edit the base `*-impl.xml` files directly, if you deactivate and reactivate a deployment setting, the SDK resets these files, so you have to copy the whole target and its dependencies, else your build script will become invalid when you disable the deployment option.

[documentation](#), [sdk](#), [builds](#), [project](#), [deployment](#)

</div>

# Developing for jMonkeyEngine SDK

*Note that all info is subject to change while jMonkeyEngine SDK is still in beta!*

In general, developing plugins for jMonkeyEngine SDK is not much different than creating plugins for the NetBeans Platform which in turn is not much different than creating Swing applications. You can use jMonkeyEngine SDK to develop plugins, be it for personal use or to contribute to the community.

If you feel like you want to make an addition to jMonkeyEngine SDK, don't hesitate to contact the jme team regardless of your knowledge in NetBeans platform development. For new plugins, the basic project creation and layout of the plugin can always be handled by a core developer and you can go on from there fleshing out the plugin. By using the Platform functions, your plugin feels more like a Platform application (global save button, file type support etc.).

</div>

## Creating plugins and components

- [Creating a plugin](#)
- [Creating components](#)
- [Building the jME SDK from scratch](#) (not necessary for plugin development, only for contributors)

## Extending jMonkeyEngine SDK

- [The Main Scene](#)
- [The Scene Explorer](#)
- [Projects and Assets](#)

## Recipes

- [Create a library plugin from a jar file](#)
- [Create a new or custom model filetype and loader](#)

## General Notes

- **Remember the scene runs on the render thread and most everything you do in the plugin (button events etc.) runs on the AWT thread, always encapsulate calls**

**to either side correctly via Callables/Runnables or register as an AppState to the SceneApplication to have an update() call by the render thread.**

- Although the scene can be accessed at any time via SceneApplication.getApplication() it is not recommended to modify the scene like that. Other plugins might be accessing the scene and updates will not be properly recognized. Use the sceneRequest object and the lookup of selected nodes and files to access things like the assetManager etc.
- It became a standard in jMonkeyEngine SDK to start the name of methods that execute directly on the OpenGL thread with “do” e.g “doMoveSpatial”, this makes identifying threading issues easier.
- The AssetManager of jme3 is threadsafe and can be used from any thread to load assets
- You can get access to the ProjectAssetManager via the Lookup of a JmeSpatial and other objects
- Use org.openide.filesystems.FileObject instead of java.io.File for file access, it always uses system-independent “/” path separators and has some more advanced functions that make file handling easier.
- You can get a file from a string using  
`Repository.getDefault().getDefaultFileSystem().getRoot().getFileObject("aaa/bbb/ccc/whatever");`
- You can convert a regular java File to a FileObject and vice versa using  
org.openide.filesystems.FileUtil
- If you have problems with unresolved classes, check if all needed Libraries are registered in the settings of your Project. To find out which library contains a certain class, just enter the name in the library search field.

## Terminology used here

- A “plugin” is anything you can tick in the plugin manager of the SDK. It can contain editors, simple “Java SE Libraries” that you can add to your projects as jar files and other things like project templates etc.
- A “module” is the project type that allows you to create plugins, strictly speaking all plugins are modules but there can be modules that are never shown in the plugin list and only exist as dependencies of other modules.
- A “library” is an entry for a jar file (and optionally sources and javadocs) which can be added to a SDK project to be used and distributed with it
- An “extension” is a generic name for stuff that extends the jME engine, like pathfinding algorithms or anything that can be used at the game runtime..

So if you have some cool code that others can use in their games too, you would make your extension a library by creating a module that the users can download as a plugin :)

## Handy things in jMonkeyEngine SDK Core

- com.jme3.gde.core.scene.controller
  - AbstractCameraController → A basic camera control for plugins, used by SimpleSceneComposer and “View Model”
  - SceneToolController → A basic controller for having selection, cursor etc. displayed in the scene, used by SimpleSceneComposer
- com.jme3.gde.core.scene
  - OffViewPanel → A panel that renders a 3d scene in a preview and displays it in a lightweight swing panel
- com.jme3.gde.core.util
  - DataObjectSaveNode → Allows enabling the save all button by using any file and implementing the SvaeCookie yourself.

Learn more about NetBeans Plugin Development at <http://platform.netbeans.org>

Also check out this Essential NetBeans Platform Refcard:

<http://refcardz.dzone.com/refcardz/essential-netbeans-platform>

[documentation](#), [sdk](#), [contribute](#)

</div>

# title: Developing for jMonkeyEngine SDK

## Developing for jMonkeyEngine SDK

*Note that all info is subject to change while jMonkeyEngine SDK is still in beta!*

In general, developing plugins for jMonkeyEngine SDK is not much different than creating plugins for the NetBeans Platform which in turn is not much different than creating Swing applications. You can use jMonkeyEngine SDK to develop plugins, be it for personal use or to contribute to the community.

If you feel like you want to make an addition to jMonkeyEngine SDK, don't hesitate to contact the jme team regardless of your knowledge in NetBeans platform development. For new plugins, the basic project creation and layout of the plugin can always be handled by a core developer and you can go on from there fleshing out the plugin. By using the Platform functions, your plugin feels more like a Platform application (global save button, file type support etc.).

### Creating plugins and components

- [Creating a plugin](#)
- [Creating components](#)
- [Building the jME SDK from scratch](#) (not necessary for plugin development, only for contributors)

### Extending jMonkeyEngine SDK

- [The Main Scene](#)
- [The Scene Explorer](#)
- [Projects and Assets](#)

### Recipes

- [Create a library plugin from a jar file](#)
- [Create a new or custom model filetype and loader](#)

### General Notes

- Remember the scene runs on the render thread and most everything you do in the plugin (button events etc.) runs on the AWT thread, always encapsulate calls to either side correctly via Callables/Runnables or register as an AppState to the SceneApplication to have an update() call by the render thread.
- Although the scene can be accessed at any time via SceneApplication.getApplication() it is not recommended to modify the scene like that. Other plugins might be accessing the scene and updates will not be properly recognized. Use the sceneRequest object and the lookup of selected nodes and files to access things like the assetManager etc.
- It became a standard in jMonkeyEngine SDK to start the name of methods that execute directly on the OpenGL thread with “do” e.g “doMoveSpatial”, this makes identifying threading issues easier.
- The AssetManager of jme3 is threadsafe and can be used from any thread to load assets
- You can get access to the ProjectAssetManager via the Lookup of a JmeSpatial and other objects
- Use org.openide.filesystems.FileObject instead of java.io.File for file access, it always uses system-independent “/” path separators and has some more advanced functions that make file handling easier.
- You can get a file from a string using  
`Repository.getDefault().getDefaultFileSystem().getRoot().getFileObject("aaa/bbb/ccc/whatever");`
- You can convert a regular java File to a FileObject and vice versa using  
`org.openide.filesystems.FileUtil`
- If you have problems with unresolved classes, check if all needed Libraries are registered in the settings of your Project. To find out which library contains a certain class, just enter the name in the library search field.

## Teminology used here

- A “plugin” is anything you can tick in the plugin manager of the SDK. It can contain editors, simple “Java SE Libraries” that you can add to your projects as jar files and other things like project templates etc.
- A “module” is the project type that allows you to create plugins, strictly speaking all plugins are modules but there can be modules that are never shown in the plugin list and only exist as dependencies of other modules.
- A “library” is an entry for a jar file (and optionally sources and javadocs) which can be added to a SDK project to be used and distributed with it
- An “extension” is a generic name for stuff that extends the jME engine, like pathfinding algorithms or anything that can be used at the game runtime..

So if you have some cool code that others can use in their games too, you would make your extension a library by creating a module that the users can download as a plugin :)

## Handy things in jMonkeyEngine SDK Core

- com.jme3.gde.core.scene.controller
  - AbstractCameraController → A basic camera control for plugins, used by SimpleSceneComposer and “View Model”
  - SceneToolController → A basic controller for having selection, cursor etc. displayed in the scene, used by SimpleSceneComposer
- com.jme3.gde.core.scene
  - OffViewPanel → A panel that renders a 3d scene in a preview and displays it in a lightweight swing panel
- com.jme3.gde.core.util
  - DataObjectSaveNode → Allows enabling the save all button by using any file and implementing the SvaeCookie yourself.

Learn more about NetBeans Plugin Development at <http://platform.netbeans.org>

Also check out this Essential NetBeans Platform Refcard:

<http://refcardz.dzone.com/refcardz/essential-netbeans-platform>

[documentation](#), [sdk](#), [contribute](#)

</div>

## title: Creating an extension library plugin

# Creating an extension library plugin

This page describes how you can wrap any jar library into a plugin that a jMonkeyEngine SDK user can download, install and then use the contained library in his own game projects.

Make sure you have your SDK set up for plugin development [as described here](#).

Creating the plugin project (in jMonkeyEngine SDK):

- Create a new Module Suite (or use an existing one)
- Open the suite, right-click the “Modules” folder and select “Add new..”
- For “Project Name” enter an all-lowercase name without spaces like `my-library`
- Make sure the “Project Location” is inside the module suite folder and press “Next”
- Enter the base java package for your plugin in “Code Name Base” like `com.mycompany.plugins.mylibrary`
- Enter a “Module Display Name” for your plugin like “My Library”
- Press Finish

Adding the library:

- Right click the Module Project and select “New→Other”
- Under “Module Development” select the “Java SE Library Descriptor” template and press “Next”
- If you dont have the external library registered in the jMonkeyEngine SDK yet, click “Manage Libraries” and do the following:
  - Click “New Library”, enter a name for the library and press OK
  - In the “Classpath” tab, press “Add JAR/Folder” and select the jar file(s) needed for the library
  - In the “JavaDoc” tab, press “Add ZIP/Folder” and add the javadoc for the library (zipped or folder)
  - In the “Sources” tab you can add a folder or jar file containing the source files of the library if available
  - Press OK
- Select the external library from the list and press “Next”
- Enter a name for the Library (used as filename for the description file)
- Enter a display name for the Library (This is the name the user later sees in his library list)

- Press OK

You will notice a new file “MyLibrary.xml” is created in the plugins base package and linked to in the layer.xml file. Additionally the jar file and sources /javadoc are copied into a “release” folder in the project root. This is basically it, you can configure a version number, license file (should be placed in Module root folder) and more via the Module Properties.

Note that the files in the release folder are **not** automatically updated when the library changes, you have to pack and replace the jar and zip files manually. See the build script extension in the link below on how you can make your module build script do that automatically.

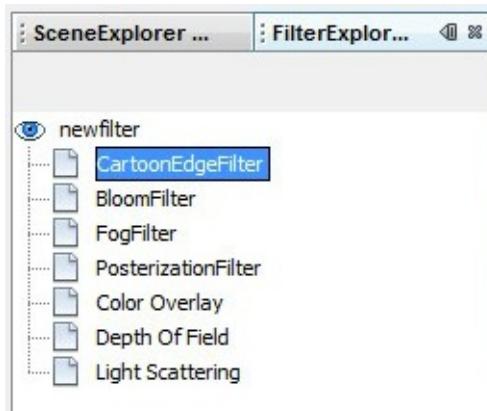
After you are done, you can [contribute the plugin in the jMonkeyEngine SDK contribution update center](#).

# **title: jMonkeyEngine SDK: Post-Processor Filters**

## **jMonkeyEngine SDK: Post-Processor Filters**

Filters are used for scene-wide effects such as glow, fog, blur. The SDK lets you create a file storing combinations of filters. You can preview the filter settings on a loaded scene in the SDK. You can load them into your application (add them to the viewPort) to activate your preconfigured set of several filters in one step.

### **Creating Filters**



To create a new filter:

- In the Projects window, right-click Assets→Effects.
- Select File→New File...
- Select Filter→Empty FilterPostProcessor File in the New File Wizard.  
An empty filter file appears in the Assets→Effects directory.
- Double-click the created file.  
The file opens in the FilterExplorer window.

### **Editing Filters**

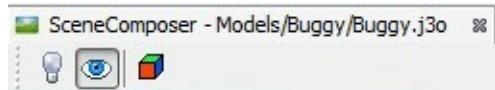
To add filters or modify existing filters

1. Double-click a j3f file to open it in the FilterExplorer window.

2. Right-click the j3f file's root node to add a filter.  
Added filters are listed under the filter's root node.
3. Open the Properties window and select a filter in the FilterExplorer. Configure filter options like intensity etc.

View the filter in the SceneComposer to see what you are doing:

## Viewing Filters



To see a loaded filter

1. Open a model or scene in the SceneComposer.
2. Double-click a j3f file to open it in the FilterExplorer window.
3. Press the “show filter” button in the OpenGL window.

## Loading filters in a game

To load a filter in a game (that is, to add it to your game's viewport), add the following lines to your game's simpleInit() method (or some other place):

```
FilterPostProcessor processor = (FilterPostProcessor) assetManager.
viewPort.addProcessor(processor);
```

[documentation](#), [sdk](#), [effect](#), [file](#)  
</div>

## **title: Font Creation**

# **Font Creation**

Most jME3 projects use Bitmap Fonts in order to display text to the user. This is because they render much faster than alternatives, although they do have a number of limitations. The main limitation being that the text will not scale in or out very well, it is designed to be displayed at a certain pixel size on the screen and will lose quality if it zooms too far in or out from that size.

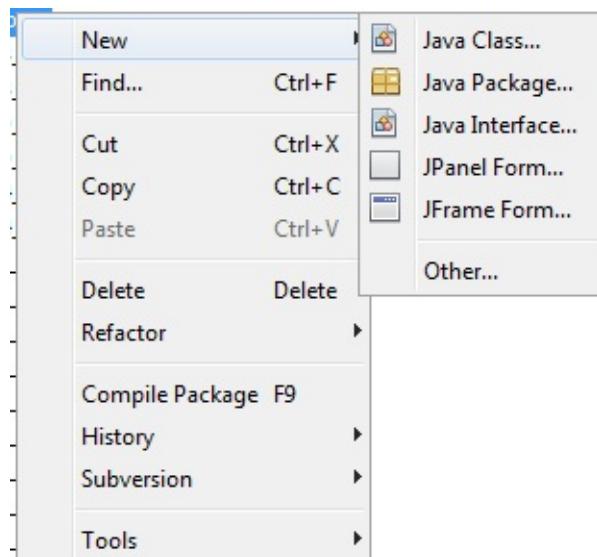
All current jME3 GUI libraries use Bitmap Fonts, as does the BitmapText class provided within jME3.

Bitmap Fonts are actually composed of two files. One is a PNG image containing the individual letters, the other is a .fnt text file containing definitions of all of the characters supported by the font and where to find them inside the PNG image.

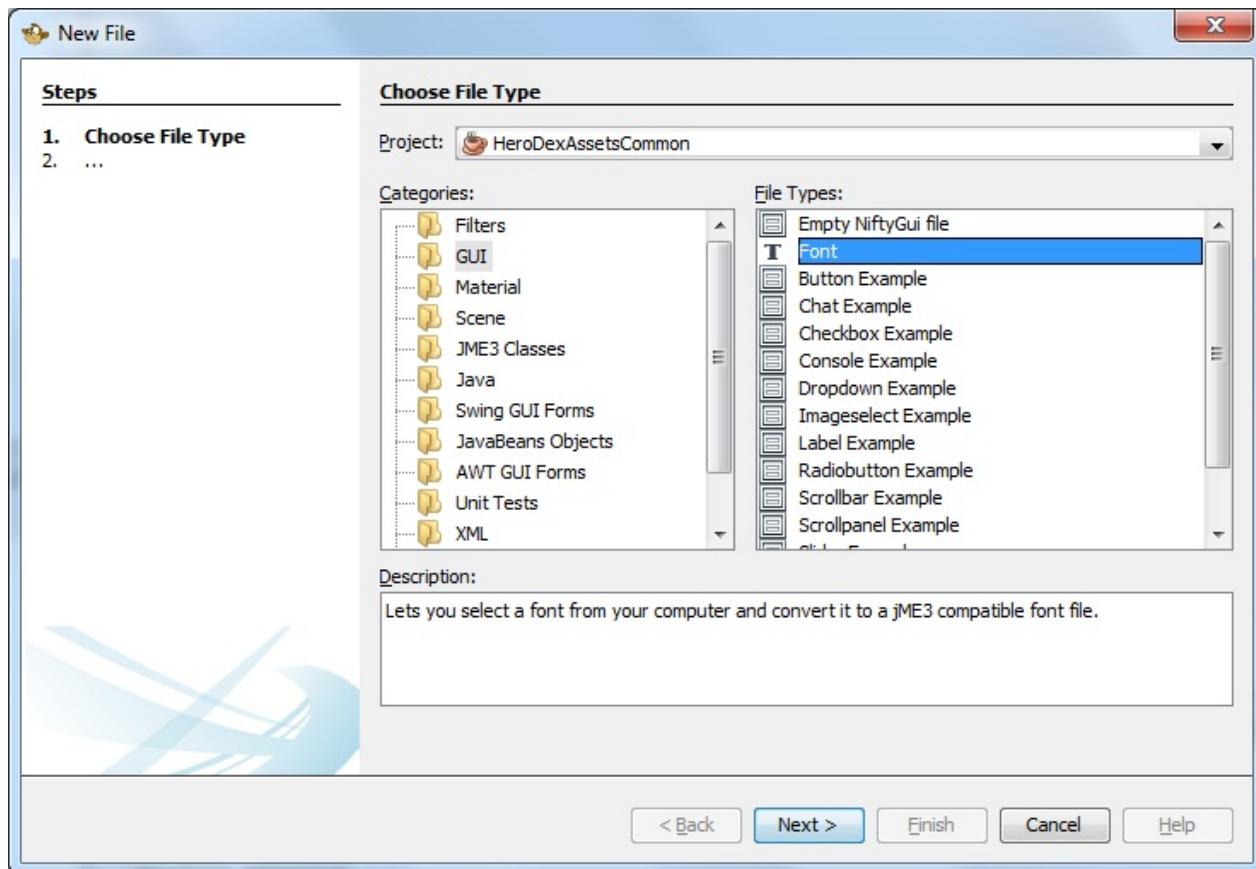
The SDK provides a tool for automatically creating a Bitmap Font from any font installed on your computer.

To create a font right click on the asset folder where you want to create the font, for example Interface/Fonts.

Open the “new” sub menu and then select “other”.

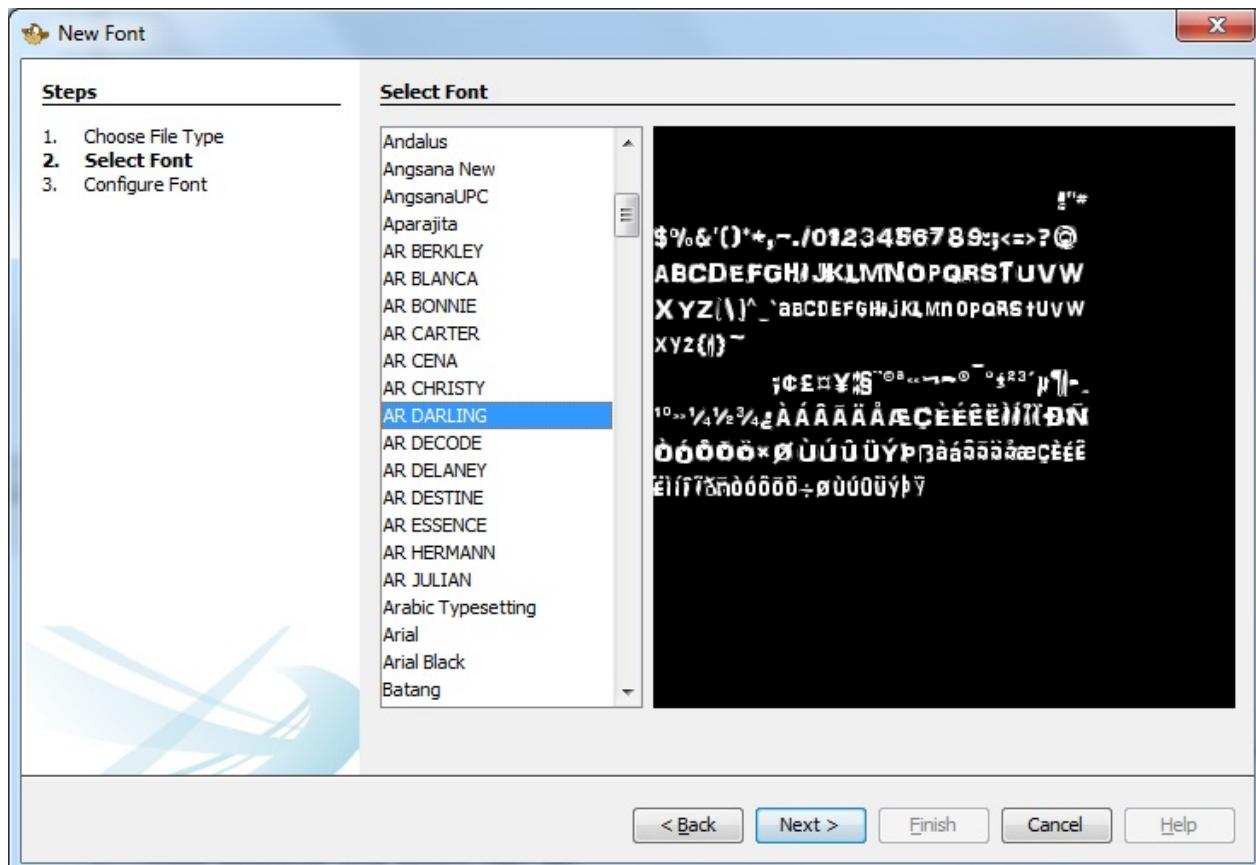


Within the window that pops up select GUI and then Font then click Next

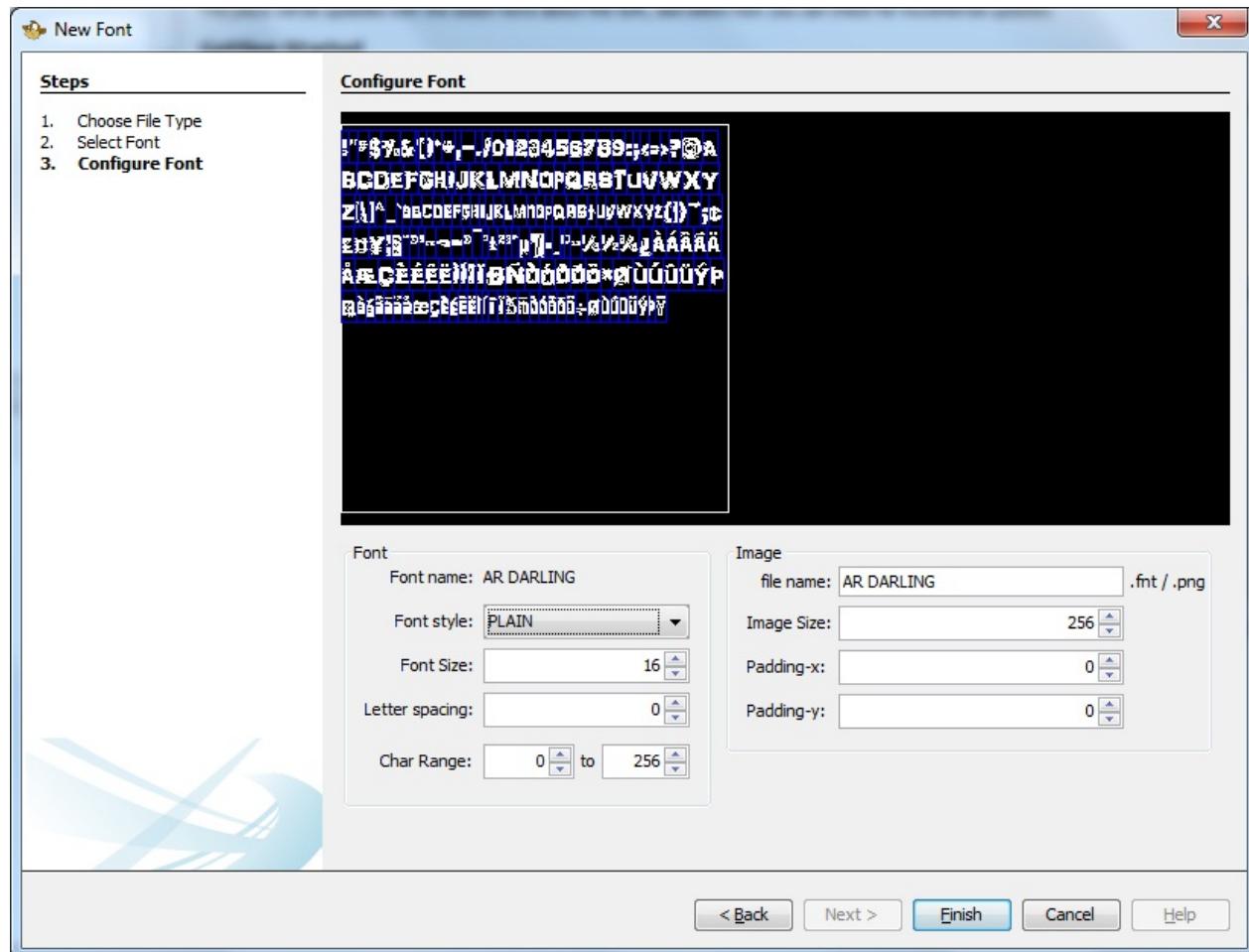


A list of every standard font installed on your system is then displayed, select the font that you would like to use to create the Bitmap Font file from and then click Next.

When selecting fonts you should check that this use does not breach the license of the font.



With your font selected you can move onto the final screen which looks as follows:



</div>

## Font Settings

The settings in the box to the left configure how the font is rendered. The Font Name is shown again to confirm your choice. Use the Back button if you want to change your mind.

The Style (For example Bold, Plain, Italic) and Font Size (in Points) can be selected as can the spacing between letters.

The Char Range selects the characters to render from the Unicode Character set. The default range of 0 to 256 renders every displayable character used in the most western languages, however the range can be extended if more characters are needed at the expense of a larger Image file.

</div>

## Image Settings

The settings in the box to the right control the generated files.

The first option specifies the file name. This should not contain a file extension as two files will be generated, both using this name but one with a .fnt extension and one with a .png extension. Note that as the generated .fnt file references the .png one then if you rename the files after generating them you will need to go into the .fnt file and manually update the reference.

The next setting shows the Image Size of the image to produce. It is recommended that you use a square power-of-two texture (256×256, 512×512, etc) for maximum performance across various graphics cards.

The final two settings are for the horizontal and vertical padding to add around each character. For most cases these can be left as 0 but sometimes it can be useful to expand them.

</div>

## Preview

The black area at the top of the window shows a preview of the generated file. You can use it to get some idea of what the characters will look like. The blue lines show the edges between the various characters that make up the font.

</div>

## Finishing

When you are ready press Finish to generate the files or Cancel to abort.

</div>

## Advanced Effects

The generated file is a standard PNG file so it can be loaded into an image editing program and effects added. For example you could add a padding onto each of the characters using the padding-x and padding-y settings and then use that to add an outer glow, drop shadow, or anything else you liked.

When rendered the Bitmap Font is multiplied with the text colour specified so if the text is set to white then any colour can be used inside the PNG file. If the PNG file just uses white then any colour can be used for the text. Furthermore the two settings can be combined if desired although this should be done with care as the multiplication will tend to make colours darker or even disappear entirely. (For example red multiplied with green gives black).

</div>

## **title: Creating plugin components**

# **Creating plugin components**

For the most common extensions like Windows, File Types, Libraries etc. there exist templates that you can add to your plugin.

1. Right-click your Module Project and select New→Other
2. Select “Module Development” to the left

You will see a list of components you can add to your project. A wizard will guide you through the creation of the component.

## **title: Increasing Heap Memory in the jMonkeyEngine SDK**

# **Increasing Heap Memory in the jMonkeyEngine SDK**

If you've been working on a large scene in the SDK, there is a good chance that you will need to increase the size of Java's heap space (which is quite small by default). [What is heap space?](#)

To increase the amount of heap space allocated to the SDK, we must navigate to the SDK's installation directory and edit the etc/jmonkeyplatform.conf file.

The installation directory for Mac and Windows is:

**OS X:**

Applications/jmonkeyplatform.app/Contents/Resources/jmonkeyplatform/jmonkeyplatform

**Windows:** C:\Program Files\jmonkeyplatform

[documentation](#), [sdk](#), [faq](#)

</div>

## title: jMonkeyEngine SDK: Log Files

# jMonkeyEngine SDK: Log Files

You find the jMonkeyEngine SDK log file in `/dev/var/log/messages.log` in the jMonkeyEngine SDK preferences folder. You can learn the location of the preferences folder in the “About” screen of the jMonkeyEngine SDK under the label **Userdir**.

- Windows: `C:\Documents and Settings\YOUR_NAME\.jmonkeyplatform\`
- Linux: `/home/YOUR_NAME/.jmonkeyplatform/`
- Mac OS: `/Users/YOUR_NAME/Library/Application Support/jmonkeyplatform/`

The message log contains all paths and **capabilities** used in your development system, and also warnings, e.g. if a plugin crashed.

## Example Log

```
>Log Session: Saturday, September 24, 2011 10:45:30 AM CEST
>System Info:
 Product Version = jMonkeyPlatform Alpha-4
 Operating System = Mac OS X version 10.6.8 running on i386
 Java; VM; Vendor = 1.6.0_26; Java HotSpot(TM) Client VM 26.1-b04
 Runtime = Java(TM) SE Runtime Environment 1.6.0_26-b05
 Java Home = /System/Library/Java/JavaVirtualMachines
 System Locale; Encoding = de_DE (jmonkeyplatform); MacRoman
 Home Directory = /Users/joemonkey
 Current Directory = /
 User Directory = /Users/joemonkey/Library/Application Su
 Installation = /Applications/jmonkeyplatform.app/Content
 ...
 ...
```

[documentation](#), [sdk](#), [file](#)

</div>

# title: jMonkeyEngine SDK: Material Editor

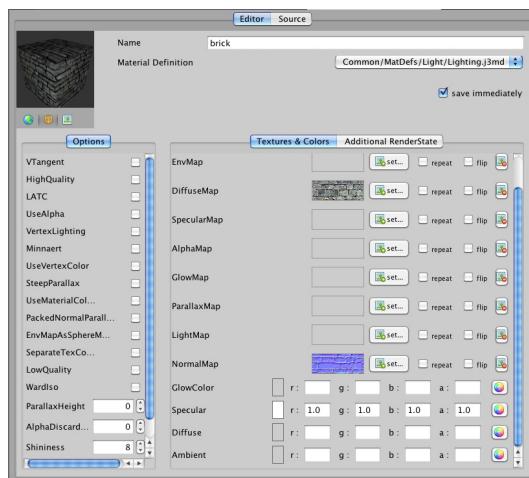
## jMonkeyEngine SDK: Material Editor

If you are looking for background information, read about [Material Definitions](#) and [j3M Material Files](#). You can [write .j3m files in a text editor](#), or [use the jMonkeyEngine SDK to generate them](#) for you as described in this article.

## Materials

The jMonkeyEngine uses a special Material format, which comes in .j3m files. You use .j3m files to store sets of material properties that you use repeatedly. This enables you write one short line of code that simply loads the presets from a custom .j3m file. Without a .j3m file you need to write several lines of material property setters every time when you want to use a non-default material.

## Creating .j3m Materials



To create new .j3m files in the jMonkeyEngine SDK,

1. Right-click the `assets/Materials` directory and choose New... > Other.
2. In the New File Wizard, choose Material > Empty Material File, and click Next.
3. Give the file a name, for example `mat_wall` for a wall material.
4. A new file `mat_wall.j3m` is created in the Materials directory and opens in the Material Editor.

You can edit the source of the material, or use the user-friendly visual editor to set the properties of the material. Set the properties to the same values as you would otherwise specify with setters on a Material object in Java code:

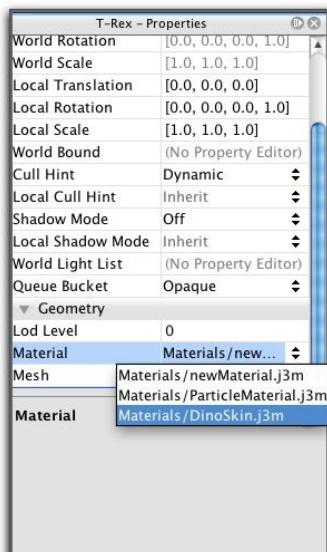
```
Material mat_wall = new Material(
 assetManager, "Common/MatDefs/Light/Lighting.j3md");
mat_wall.setTexture("DiffuseMap",
 assetManager.loadTexture("Textures/wall_diffuse.png"));
mat_wall.setTexture("NormalMap",
 assetManager.loadTexture("Textures/wall_normals.png"));
mat_wall.setFloat("Shininess", 5f);
```

This Java code corresponds to the following .j3m file:

```
Material my brick wall : Common/MatDefs/Light/Lighting.j3md {
 MaterialParameters {
 DiffuseMap: Repeat Textures/wall_diffuse.png
 NormalMap: Repeat Textures/wall_normals.png
 Shininess: 5.0
 }
}
```

You can modify the source code of the j3m file in the “source” tab of the Material Editor.

## Using .j3m Materials



When the material is ready and saved into your projects assets directory, you can assign the .j3m to a Geometry.

### In the jMonkeyEngine SDK

1. Right-click the j3o file and select “Edit in SceneComposer”
2. Open the SceneExplorer window
3. In the SceneExplorer, click the geometry to which you want to assign the material.
4. Open the Properties window
5. Assign the .j3m material to the .j3o in the Properties>Geometry>Material section
6. Save the j3o and load it into your game.

Or in your Java code

- Use a loader and a setter to assign the material to a Geometry

```
mywall.setMaterial(assetManager.loadMaterial("Materials/mat_wall.j
```

---

### See also:

- [Developer specification of the jME3 material system \(.j3md,.j3m\)](#)
- [Hello Material](#)
- [Materials Overview](#)
- [Material Definitions](#)
- [j3M Material Files](#)
- [Neotexture \(Procedural textures\)](#)

[documentation](#), [sdk](#), [material](#), [file](#), [texture](#)

</div>

## **title: jMonkeyEngine SDK: Importing and Viewing Models**

# **jMonkeyEngine SDK: Importing and Viewing Models**

The jMonkeyEngine SDK imports models from your project and stores them in the assets folder. The imported models are converted to a jME3 compatible binary format called .j3o. Double-click .j3o files in the jMonkeyEngine SDK to display them in the SceneViewer, or load them in-game using the AssetManager.

Presently, [Blender 3D](#) is the preferred modelling tool for jME3 as it is also Open-Source Software and an exporter for OgreXML files exists. There is a direct .blend file importer available in the SDK and you can directly import or store blend files in your project to convert them. If for some reason your version of blender is not compatible, you can use the default OgreXML format. Note that the OgreXML exporter is not compatible with Blender 2.49 or before!

Also, see this [demo video](#) on importing models.

## **Installing the OgreXML Exporter in Blender**

The jMonkeyEngine SDK includes a tool to install the correct exporter tools in Blender to export OgreXML files. To install the exporter, do the following:

1. Select “Tools”→“OgreXML”→“Install Blender OgreXML” in the jMonkeyEngine SDK Menu
2. If you are presented a filechooser, select the folder where your blender scripts reside
3. Press “Install” in the window that opens

Also check out [how to create compatible models in blender](#) and [how to organize your assets](#).

## **Importing and Viewing a Model**

### **Using the Model Importer Tool**

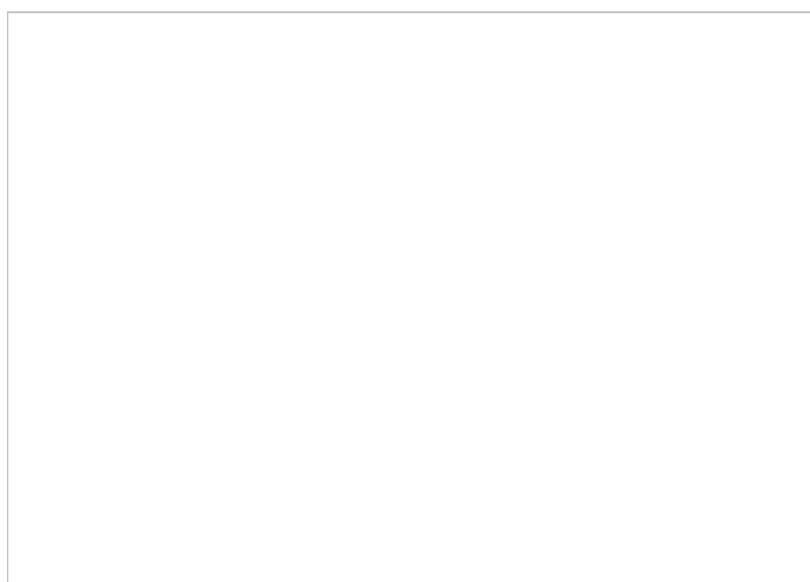
jMonkeyEngine SDK includes a model importer tool that allows you to preview and import supported models into your jMonkeyEngine3 project.



1. Select the project you want to import your model to
2. Click the “Import Model” button in the toolbar
3. Click “open model..” and select the main model file
4. Check the preview and file list and press “Next”
5. Check and change the import path if necessary
6. If you want to copy the original model files as well, check the checkbox
7. Press “finish”

The model is converted to j3o and all necessary files are copied to the project assets folder. If you have specified the corresponding option, all original model files will also be copied to your assets folder.

## Using the model files directly



1. Create a separate folder for each model in the `assets` folder of your project.
2. Save the model created in Blender (.blend) **in the asset folder of your project**,
3. Make sure that no textures are packed in the .blend file.
4. Make sure all textures used in the .blend file are in the assets folder of the project.
5. In the Projects Explorer Assets node, select the model that you want to import.
6. Double-click the model or right-click and select “Convert to JME binary” from the context-menu.
7. In the Projects Explorer Assets node you should see your model j3o file.
8. Double-click it to view it in the SceneViewer.
9. Click on the lightbulb to turn on the light if you cannot see your model.

Note: It is important that you copy the model file and its textures to the correct assets folder before creating the j3o file because the paths for textures (and possibly other things) will be stored as absolute (to the assets folder root) when you convert that model. This means the texture location should not change after the import.

Note: If the SceneViewer doesn't work refer to [Troubleshooting jMonkeyEngine3 SDK](#).

## Working With a Model

- Open Windows>SceneExplorer to view sub-nodes of the model
- Open Windows>Properties to view the properties of the model's nodes.
- Click the cube button in the SceneViewer to toggle between Wireframe mode and Textured mode.
- Click the lightbulb to view Materials that require a light source

## Notes About Model Assets

The original OgreXML `.mesh.xml`, `.scene`, `.material`, `.skeleton.xml` and `.blend` model files **will not be included** in the distribution `assets.jar` file of your distributed game, they are only available in the assets folder so you are able to recreate the `.j3o` file from the original if you ever come to change it in blender and have to export it again.

## About the SceneViewer and SceneExplorer window

The SceneViewer and SceneExplorer windows are shared among plugins to save system resources. This means that you will have to keep an eye on what plugin is using the scene right now and what you are actually modifying in these windows.

Most plugins will deliver their own UI elements to modify the scene so the SceneExplorer is more of a global tool. The simple SceneComposer however heavily relies on its functions as other plugins might too in the future.

## About the projects AssetManager

Each jMonkeyEngine SDK project has its own internal AssetManager that has the projects assets folder registered as a FileLocator. When the project is run, the assets folder is compressed into a jar file and added to the projects main jar classpath. This way the editors in jMonkeyEngine SDK and the running game have the same asset folder structure.

You might wonder why jMonkeyEngine SDK requires you to copy the model that is to be converted to j3o into the assets folder before. The Model Import Tool also copies the model and associated files to the project directory before converting. To load the model it needs to be in a folder (or jar etc..) that belongs to the projects AssetManager root. To load a model from some other folder of the filesystem, that folder would have to be added to the AssetManager root. If every folder that contains a model was in the root of the AssetManager, all textures named “hull.jpg” for example would be the same for the AssetManager and it would only deliver the texture of the first model folder that was added.

To have a valid jME3 object, the paths to textures and other assets belonging to the model have to be read with the correct, final path that can then be stored in the j3o object. The j3o object will use those paths when it is loaded with the AssetManager and it requires the AssetManager to deliver the assets on those paths, this is why the folder structure while converting has to be the same as when loading.

[documentation](#), [sdk](#), [tool](#), [asset](#), [scene](#)

</div>

## **title: jMonkeyEngine SDK: Creating a model importer**

# **jMonkeyEngine SDK: Creating a model importer**

You can create custom model importers for the jMonkeyEngine SDK. The SDK supports NBM plugins.

1. [Create an NBM plugin](#)
2. Add importer jar file (wrap jar file)
3. Add filetype (Template)
4. Change DataObject to extend SpatialAssetDataObject
5. Implement getAssetKey(): if(!assetKey instanceof MyKeyType){assetKey = new MyKeyType(oldKey);} return key;
6. Maybe implement loadAsset method in DataObject (if necessary, most model formats should load normally via the loader)
7. Create AssetManagerConfigurator

See also:

- [Projects and Assets](#)
- <http://platform.netbeans.org/tutorials/nbm-filetype.html>

[documentation](#), [sdk](#), [tool](#)

</div>

## **title: jMonkeyEngine SDK: Procedural Textures with NeoTexture**

# **jMonkeyEngine SDK: Procedural Textures with NeoTexture**

The NeoTextureEditor allows creating tiled textures procedurally using a simple node interface for generating images, blending them, creating normal maps and much more. You can directly load the .tgr files as a material or export the generated images as .png files and use them in a jMonkeyEngine-based game.

Textures usually make up most of the size of a game distribution. This is why NeoTexture is not only an editor, but also a library that generates textures from .tgr files. Use the library to load .tgr files directly in jME3 as a material, without having to export the textures before. This means high-quality textures for your models, but just tiny description files in your distribution.

## **Creating and Editing a NeoTexture file**



1. Right-click the `assets/Textures` directory and choose New... > Other.
2. In the New File Wizard, choose NeoTexture > Empty NeoTextureFile and click Next.
3. Give the file a name, for example `neoMaterial`.
4. A new file `neoMaterial.tgr` is created in the Textures directory and opens in the NeoTexture Editor.

Edit the .tgr file and create your procedural texture: ([Learn more about creating Procedural Textures here](#))

1. Drag any pattern from the left bar to the editor area.
2. Right-click the editor area and paste a NormalMap filter node.
3. Connect the green output mark of the pattern with the red input mark of the filter.  
This generates a Normal Map that can be used as bump map.

## **Adding the NeoTexture libraries to your project**

To use NeoTexture tgr files directly in your application, you have to add the NeoTexture libraries to your project:

1. Right-click your project and select “Properties”
2. Go to the “Libraries” section of the properties window
3. Press “Add Library..”
4. Select “NeoTexture-Libraries” and press “add”

## Loading tgr files as material



We want to use the procedural texture as a material based on `Lighting.j3md` (the default). We know that `Lighting.j3md` supports `DiffuseMap`, `NormalMap`, `SpecularMap`, and `ParallaxMap`.

1. Click the Normal Map filter to select it. We want to use it as the Normal Map of the texture.
  - i. In the NeoTextures Properties window under `Export name`, enter the name of the texture layer as it would appear in a `.j3m` file, e.g. “`NormalMap`”.
2. Click the pattern to select it. We want to (re)use it as Diffuse Map of the texture.
  - i. In the NeoTextures Properties window under `Export name`, enter the name of the texture layer as it would appear in a `.j3m` file, e.g. “`DiffuseMap`”.
3. Click the Save button in the NeoTextures Editor. The `.tgr` file is saved with two layers. (We could add `SpecularMap` and `ParallaxMap` the same way, if we wanted.)

Let's assign the newly created texture to a mesh to see what it looks like.

1. Open your `Main.java` file.
2. Choose `Window→Palette` to open the Palette. You will find two NeoTexture code snippets, one for adding the NeoTexture loader to the `assetManager`, and one for loading a NeoTexture material.
3. Drag&Drop one of each into the `simpleInitApp()` method of your application.
4. Adjust the `.tgr` file path names to match the file that we just created: “`Materials/neoMaterial.tgr`”.

Clean, build and run. You're ready to load your procedural material!

```

assetManager.registerLoader(com.jme3.material.plugins.NeoTextureMat
NeoTextureMaterialKey key = new NeoTextureMaterialKey("Materials/ne

Material mat = assetManager.loadAsset(key);
mat.setFloat("Shininess",12); /* Lighting.j3md has non-map parameter
thing.setMaterial(mat);

```



The default Material Definition for NeoTextures is `Lighting.j3md`, and it's probably the one you will use most often. In case that you want to use additional texture parameters, other than DiffuseMap, SpecularMap, ParallaxMap, and NormalMap, you can switch to another Material Definition using `setMaterialDef()`, for instance:

```
key.setMaterialDef("Commons/MatDefs/Misc/ColoredTextured.j3md");
```

Remember that the layer names in the `.tgr` file must match the ones declared in the `.j3md` file. In our example of `ColoredTextured.j3md`, the `.tgr` file must contain a `ColorMap` and you need to set a RGBAColor.

## Using tgr files like normal textures in j3m files

You can use the `.tgr` files like normal textures in `j3m` files, with a syntax like this:

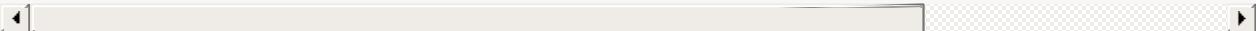
`Materials/neoMaterial?DiffuseMap.tgr` The part between the `?` and the suffix is the name of the node you want to load as a texture. If you dont supply a name, `texture` is used.

To be able to load these textures, you have to register a Locator and a Loader in the AssetManager, note that you can only register one loader per extension so you cannot load `.tgr` files as materials and as textures with the same assetManager.

```

assetManager.registerLocator("/", com.jme3.texture.plugins.NeoTextureLoc
assetManager.registerLoader(com.jme3.texture.plugins.NeoTextureLoad

```



[documentation](#), [sdk](#), [tool](#), [texture](#), [material](#)

</div>

---

## **title: Platform Development**

# **Platform Development**

[Please go here](#)

# title: jMonkeyEngine SDK: Creating Projects

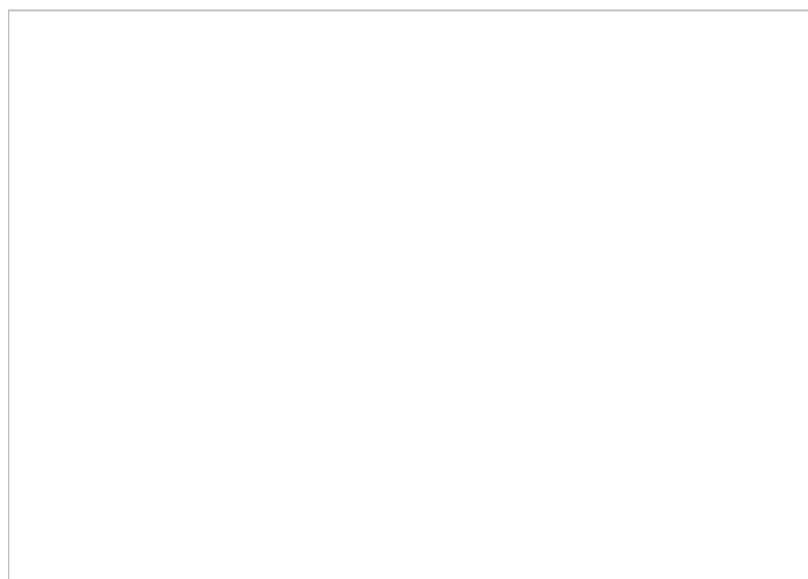
## jMonkeyEngine SDK: Creating Projects

The jMonkeyEngine SDK makes it easy to get started with developing 3-D games based on the jMonkeyEngine.

### Creating a New jMonkeyEngine Project

1. Choose File > New Project from the main menu.
2. In the New Project Wizard, select the template JME3 > Basic Game
3. Click next to specify a project name, and the path where to store your new project.
4. Click Finish. A skeleton application is created and opens in the Project Explorer.
  - This basic jme3 application is based on the SimpleApplication class to allow an easy start with jme3.
  - You can click the run button to run it: You will see a jMonkey cube.

### Project Structure



Let's have a look at the abstract project structure in the Project Explorer (ctrl-1).

- **Project Assets node:** These directories have been created for you to store your games assets, such as fonts, materials, models, shaders, sounds, and textures. For a newly created project, these directories are empty.

- **Source Packages node:** This is where you manage your packages and classes. For a newly created project, it contains one package and one class, `Main.java`. Double click `Main.java` to open it in the editor.
- **Libraries node:** An overview of all libraries on your game's classpath. The classpath is already set-up for the jme3 framework (including LWJGL, Bullet, Nifty GUI, etc).

## Directory Structure

Now let's have a look at the project's file structure in the File Explorer (ctrl-2). This explorer shows the physical directory structure on your hard drive.

- **assets** – This directory corresponds to the Project Assets node. It is needed for the assetManager. This is the recommended internal structure:
  - `assets/Interface`
  - `assets/MatDefs`
  - `assets/Materials`
  - `assets/Models`
  - `assets/Scenes`
  - `assets/Shaders`
  - `assets/Sounds`
  - `assets/Textures`
- **src** – This directory corresponds to the Source Packages node. Your sources code goes here.
- **nbproject** – This is meta data used by the jMonkeyEngine SDK (don't edit).
- **build.xml** – This is an Ant build script that is hooked up to the clean/build/run/test actions in the jMonkeyEngine SDK. It loads a default build script, and allows you to further customize the build process. The Ant script also assures that you are able to clean/build/run/test your application outside of the jMonkeyEngine SDK – e.g. from the command line.
- **build** – This directory contains the compiled classes. (Will be generated by the jMonkeyEngine SDK when you build the project.)
- **dist** – This directory contains the executable JAR files. (Will be generated by the jMonkeyEngine SDK when you build the project.)
- **test** – The jMonkeyEngine SDK will store JUnit tests here if you create any. (Optional.)

### Project Configuration

Right-Click the project to open the Project properties.

In the Run section, specify the main class of your project. (Pressing F6 runs this main class.)  
In the Run section, you can optionally configure JVM options and command line parameters  
**(in most cases set the-Xms VMOption [NUMBER] m for the memory usage. for**

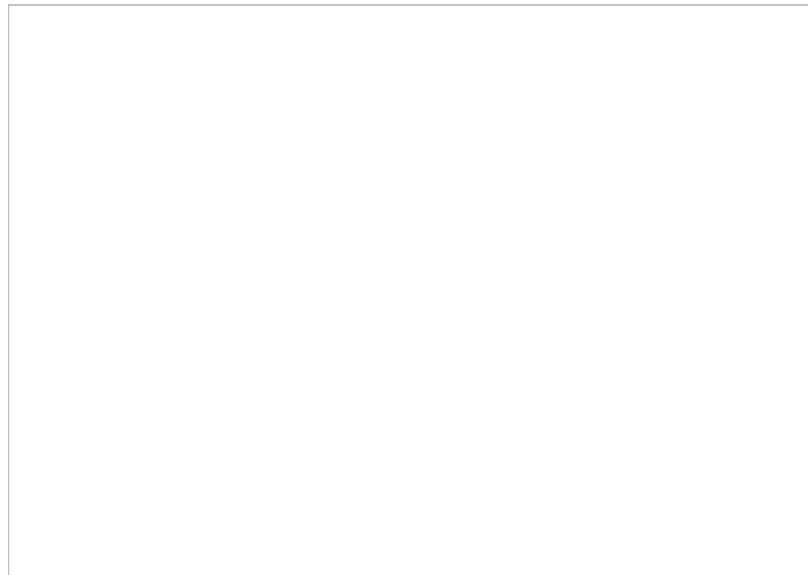
**example (-Xms500m).** see <http://performance.netbeans.org/howto/jvmswitches/>). In the Application section, specify the game title (by default the game will be named BasicGame). In the Application section, specify the vendor name (your name), a short description, your project's homepage, and a splash screen.

## Project Configuration

Right-Click the project to open the Project properties.

- In the Run section, specify the main class of your project. (Pressing F6 runs this main class.)
- In the Run section, you can optionally configure JVM options and command line parameters.
- In the Application section, specify the game title (by default the game will be named `BasicGame` ).
- In the Application section, specify the vendor name (your name), a short description, your project's homepage, and a splash screen.

## Clean, Build and Run Cycle



Pressing **F6** builds & runs the *main* class of the *main* project. If there are several classes, or several projects, you have to specify which one you want F6 to run. Right-click a project and choose Set As Main Project, then right-click the project again and choose Properties > Run and choose a Main Class.

To build and run the main() of *any file that is open in the editor*, press **Shift-F6** !

- Right-Click the project and use the context-menu to clean all generated classes and JARs.
- Right-Click individual files with a main method to build and run them. (Shift-F6)

- Press the Run button (green arrow in the toolbar) to build and run the project. (F6)

**More than one project open?** The toolbar buttons and the F-keys are bound to the main project, which is shown in bold in the Project Explorer. Right-click a project and select Set As Main Project to make it respond to the toolbar buttons and F-keys.

**Worried About Proprietary Lock-in?** You are never locked into the jMonkeyEngine SDK: At any time, you can change into your project directory on the command line, and clean, build, and run your project, using non-proprietary Apache Ant commands:

```
ant clean; ant jar; ant run;
```

## Development Process

- **Creating new files and packages:** Select the Source Packages node (or any of its subnodes), and press ctrl-N (File→New File): Use the New File wizard to create new Java classes, Java packages, Java beans, Swing forms, JUnit files, j3m Materials, j3o scenes, j3f filters, and many more.
- **Editing files:** Open the Projects Explorer and double-click a Java file from the Source Packages to open it in the Editor. The [jMonkeyEngine SDK Code Editor](#) assists you in many ways, including syntactic and semantic code coloring, code completion, and javadoc. ([More...](#))
- **Adding Assets:**
  - You can [import models, scenes, and materials](#) as assets into your project.
  - To add sound files and images, use your operating system's file explorer and copy the files into your project's asset directory.
- **ToDo List:** The tasks window automatically lists all lines containing errors and warnings, and all lines that you have marked with the comment keywords `FIXME`, `@todo`, or `TODO`.
- **Integrated tools:** [Debugging](#), [Testing](#), [Profiling](#).

## Adding external jar libraries

You may want to use external Java libraries in your jME project, for example content generators or artificial intelligence implementations.

Add the library to the global library list:

- Select Tools→Libraries in the main menu.
- Click “New Library”, enter a name for the library, and press OK
- In the “Classpath” tab, press “Add JAR/Folder” and select the jar file(s) needed for the library

- (Optional) In the “JavaDoc” tab, press “Add ZIP/Folder” and select the javadoc for the library, as zip file or folder.
- (Optional) In the “Sources” tab you can select a folder or jar file containing the source files of the library.
- Press OK

Add the library to a project:

- Right-Click your project and select “Properties”
- Select “Libaries” on the left and then press “Add Library”
- Select the library from the list and press OK

That's it, your project can now use the external library. If you also linked the javadoc and sources, the SDK will assist you with javadoc popups, code completion (ctrl-space) and source navigation (ctrl-click).

</div>

## Application Deployment

- You can [deploy](#) your game as desktop application (JAR), browser applet, WebStart (JNLP), or on the Android platform. ([More...](#))

## Running Sample Projects

The SDK contains [Sample Code](#) (read more).

Open the Source Packages node of the JmeTests project.

- Right-click the `JME3Tests` project and choose Run.  
Choose samples from the TestChooser and try out the included demos.
- Browse a demo's source code in the SDK's Project window to learn how a feature is implemented and used.
- Feel free to modify the code samples and experiment! If you break something, you can always recreate the packaged samples from the `JME3 Tests` template.

[documentation](#), [project](#), [deployment](#), [sdk](#)

</div>

## title: Projects and Assets

# Projects and Assets

The SDK heavily uses the systems provided by the base platform for the handling of assets and projects and extends the system with jME3 specific features.

</div>

## ProjectAssetManager

All AssetDataObjects and SceneExplorerNodes allow access to the ProjectAssetManager of the project they were loaded from.

```
ProjectAssetManager pm = node.getLookup().lookup(ProjectAssetManager.class)
```

The ProjectAssetManager is basically a normal DesktopAssetManager for each project with some added functionality:

- Access to the FileObject of the assets folder of the project to load and save data
- Convert absolute file paths to relative asset paths and vice versa
- Get lists of all textures, materials etc. in the project
- more convenient stuff.. :)

</div>

## AssetDataObject

Most “files” that you encounter in the SDK come in the form of AssetDataObjects. All Nodes that you encounter contain the AssetDataObject they were loaded from. It provides not just access to the FileObject of the specific file but also an AssetData object that allows access to jME specific properties and data. The AssetData object also allows loading the object via the jME3 assetManager. It is accessible via the lookup of the Node or AssetDataObject:

```
assetDataObject.getLookup().lookup(AssetData.class)
```

&lt;/div&gt;

## New Asset File Types

When you add a new file type for a model format or other asset file that can be loaded in jME3 you can start by using new file type template (New File→Module Development→File Type). Change the DataObject to extend AssetDataObject (general), SpatialAssetDataObject (some type of model) or BinaryModelDataObject (basically a j3o savable file). And possibly override the loadAsset and saveAsset methods which are used by the AssetData object to return the correct AssetKey type (needed for import properties to work).

```
public class BlenderDataObject extends SpatialAssetDataObject {
 public BlenderDataObject(FileObject pf, MultiFileLoader loader)
 super(pf, loader);
 }
}
```

[◀] [▶]

An AssetManagerConfigurator class can be created to configure the assetManager of the projects and model importer to use the new asset type:

```
@org.openide.util.lookup.ServiceProvider(service = AssetManagerConf
public class BlenderAssetManagerConfigurator implements AssetManage
 public void prepareManager(AssetManager manager) {
 manager.registerLoader(com.jme3.scene.plugins.blender.Blend
 }
}
```

[◀] [▶]

&lt;/div&gt;

## title: jMonkeyEngine SDK: Sample Code

# jMonkeyEngine SDK: Sample Code

You can run example code by opening an included JME3Tests project and included assets.

You can also search the built-in documentation or drag and drop code snippets from the Palette in the SDK to get short code sample.

## Code Palette and Samples in the SDK

- Type a keyword into the search box in the SDK or press F1 to search the built-in help for sample code.
- [SDK code editor and palette](#)

## The JME3Tests Project Template

The jMonkeyEngine SDK contains a Test Project with lots of sample code and assets. The Test Project is all set up and ready to run, and it's easy to use for beginners (no need to mess with classpaths or libraries).

1. Install and Open the jMonkeyEngine [SDK](#)
2. Go to File→New Project
3. In the New Project Wizard, select `JME3 Tests` from the `JME3` category. Click Next.
4. Specify a location, e.g. create a `jMonkeyProjects` directory in your home directory. Click Finish.

This default project template creates a project called `JmeTests`. It contains all test classes and examples from the jme3 source repository. Feel free to modify the code samples and experiment! In the unlikely event that you should break the project,  you can always recreate all packaged samples by creating another project from the New Project wizard's `JME3 Tests` template.

Press Shift-F6 to run a class that is open in the editor, or right-click a class in the Project window and choose Run File.

</div>

# JME3TestData Assets

You may want access to some sample assets, such as 3D models or textures, in one of your projects. A common situation for this would be while going through the [beginner tutorials](#).

1. Right-click an existing jME3 project in the [SDK](#), and select Properties.
2. In the Properties window, select the Libraries section. Go to the Compile tab, it contains compile-time libraries.
3. Click the Add Library... button and select the pre-defined `jme3-test-data`. Click OK.
4. Click OK to save and close the Properties.

The project's assetManager now has access to sample files from the `jme3-test-data.jar` file. This JAR library contains

- Ogre XML Models: Ninja.mesh.xml, Oto.mesh.xml, HoverTank.mesh.xml, Sinbad.mesh.xml, and many more
- Blender models
- Materials and Textures
  - Terrain, sky, rock, brick, pond...
  - Particle effect textures
- Sounds
- And more...

## AssetPacks

If you

need sample 3D models, don't miss the opportunity to download our community-provided [Asset Packs!](#)

In the SDK:

- Open the AssetPackBrowser from the Windows menu
- In the AssetPackBrowser, click the Online AssetPacks button
- Click install on the AssetPack of your choice. The SDK downloads it and makes the assets accessible in your AssetPack Library.
- Click the View Library button and open the Assets node.
- Right-click an asset to
  - Preview it
  - Add it to the SceneComposer
  - Add it to a game project's assets directory

Read more about [Asset Packs](#) and how you can share your own collection with the community.

[documentation](#), [sdk](#), [asset](#), [project](#)

</div>

## **title: jMonkeyEngine SDK: Scene Composer**

# **jMonkeyEngine SDK: Scene Composer**

SceneComposer allows you to edit scenes stored in j3o files and add content or modify existing content. Please note that the Scene Composer and Explorer are a work in progress and will provide more powerful functions in the future. Also other plugins will allow creation of more specific game type scenes in jMonkeyEngine SDK.

Most buttons in the SceneComposer have tooltips that appear when you hover the mouse over the button for a short while.

## **Mouse/Cursor Controls**

- Left-click and drag to rotate the camera around the cam center
- Right-click and drag to move the cam center
- Scroll the mouse wheel to zoom in/out the cam center
- Left-click a geometry to select it
- Right-click a geometry to place the cursor

In the SceneComposer toolbar are buttons to snap the camera to the cursor, snap the cursor to the selection and etc.

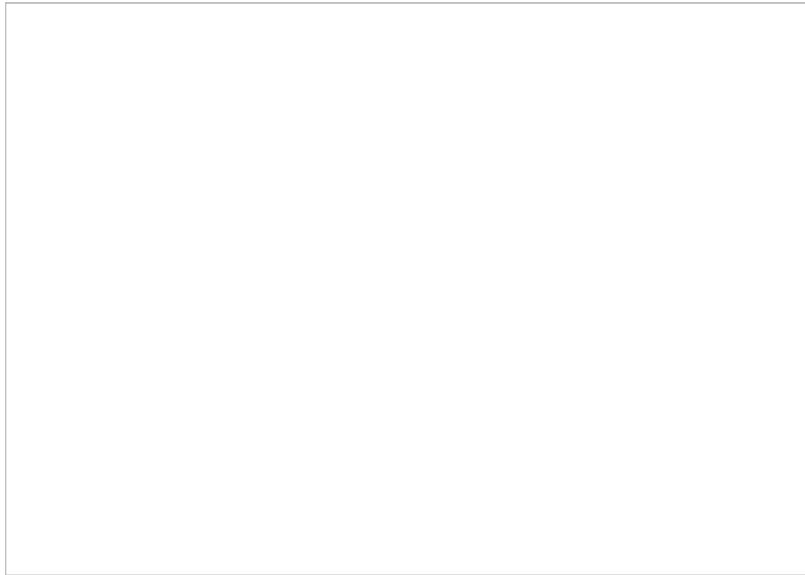
## **Creating a scene file**

The jMonkeyEngine SDK stores the scene in a j3o file, this binary file contains the whole scenegraph including all settings for spatial, materials, physics, effects etc. Textures are not stored in the j3o file but as absolute locators to the textures.

To create a blank scene file do the following:

1. Right click the “Scenes” folder in your Project Assets and select “New→Other”
2. Select “Scene” to the left then select “Empty jME3 Scene” and press “Next”
3. Enter a file name for your scene like “MyScene” and press “OK”

## **Loading the scene**



To open a scene

1. In the Project Explorer, right-click the \*.j3o file of the scene
2. Choose “Open in SceneComposer”

Now the SceneComposer window opens at the bottom and displays the scene in the SceneViewer. The SceneExplorer displays the contained scene graph as a tree and when selecting a node, you can edit the properties of the corresponding scene graph object in the Properties window.

For now, you only see the cursor in the SceneViewer and a single node (the root node of the scene) in the SceneExplorer.

## **Adding light to the scene**

1. Select the root node in the SceneExplorer
2. Select “Directional Light” in the SceneComposer window
3. Press the “+” button in the SceneComposer window

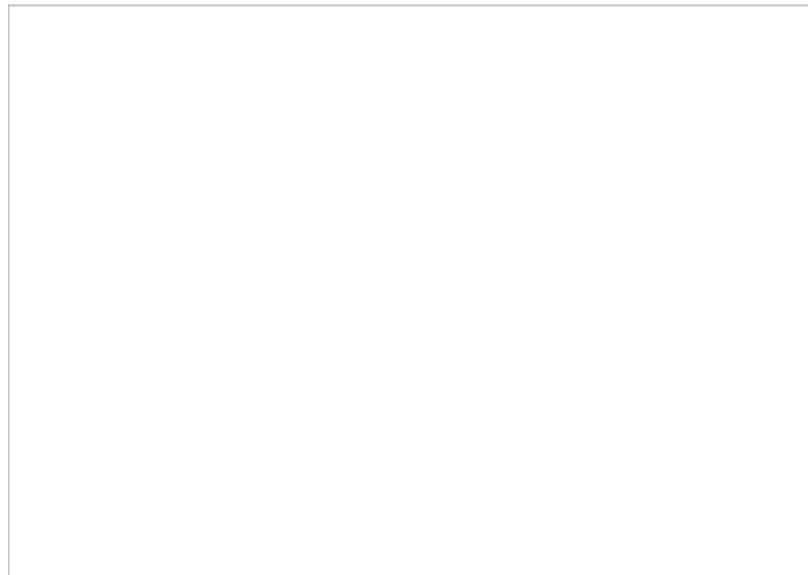
A directional light has been added to your scene, you can see it in the SceneExplorer.

## **Adding effects etc. to the scene**

You can add a variety of special objects with the SceneComposer, including lights, effects, audio etc.

1. Select root Node in the SceneExplorer
2. Select the object type in the list displayed in the SceneComposer window
3. Press the “+ cursor” button in the SceneComposer window

## Adding Models to the scene



You can directly import 3d models to your scene so that they will be part of your scene file. To be able to import for example an OgreXML file, first export it from your 3D editor to a separate folder in the assets folder of your project (e.g. assets/Models/MyModel/).

1. Place the SceneComposer cursor where you want the model to be
2. Select the parent Node for the model in the SceneExplorer
3. In the Project Explorer right-click the model file you want to import
4. Choose “Add to SceneComposer”

Note that when importing a model the texture paths are stored absolute, so the folder you import the model from will later only be a textures folder because the original model file is not included in the release.

Also note that when adding models this way, changes in the original model file will not be reflected in the scene file as its a complete copy of the original file. If you change the original model, delete the models node from the scene and import it again.

## Linking Models to the scene

You can also link models/objects into your scene, this way they are reloaded dynamically from the other/original file.

1. Place the SceneComposer cursor where you want the model to be
2. Select the parent Node for the model in the SceneExplorer
3. In the Project Explorer right-click the model file you want to link
4. Choose “Link in SceneComposer”

Note that when linking objects this way, you cannot edit them as part of the scene. To change the model you have to change the original j3o file.

Also note that although it's possible to directly link external model files (OgreXML, OBJ etc.), this is not recommended. Convert the original file to a j3o file by right-clicking it and selecting “Convert to jME Binary” before linking it. This is required because the original model files are not included in the release version of the application.

## Saving the Scene

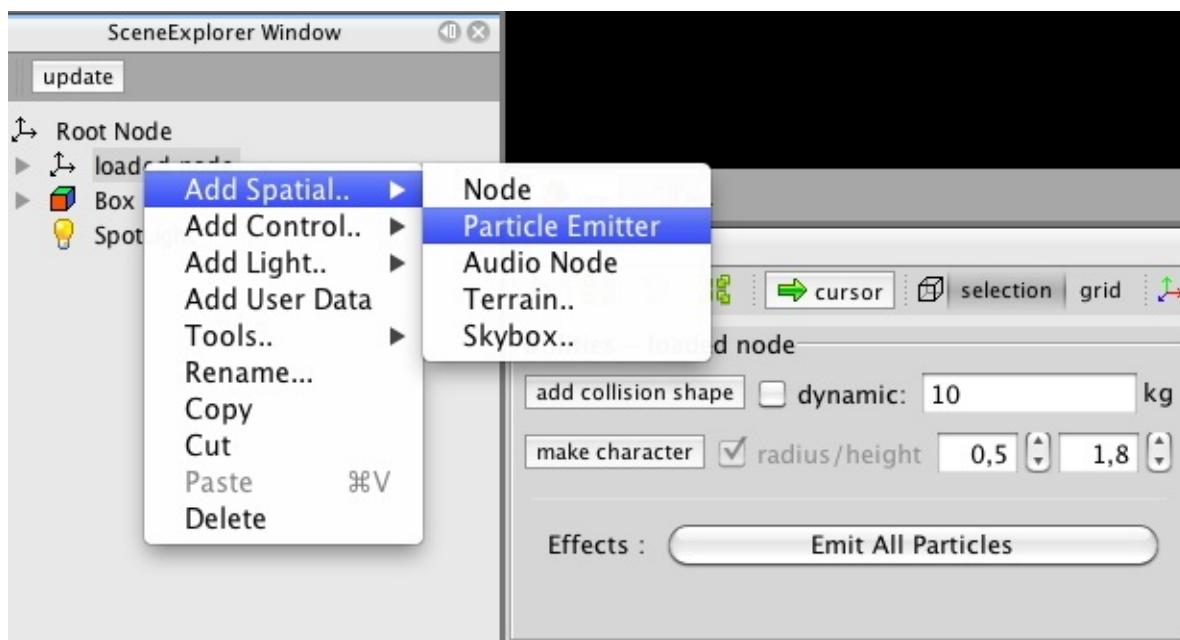
1. When a scene has been changed, press the “save” button in the main toolbar or press [Ctrl-S] / [Apple-S] to save it.

[documentation](#), [sdk](#), [scene](#), [node](#), [asset](#), [light](#), [effect](#)

</div>

## title: jMonkeyEngine SDK: Scene Explorer

# jMonkeyEngine SDK: Scene Explorer



## About the SceneExplorer window

The SceneExplorer gives you a structural overview of the currently edited scene and is active among all plugins

Most plugins will deliver their own UI elements to modify the scene so the SceneExplorer is more of a global tool. The simple SceneComposer however heavily relies on its functions as other plugins might too in the future.

## Using the SceneExplorer

The SceneExplorer displays Nodes in a tree that represents the tree of Spatial in your scene. Spatial controls, lights and geometry meshes are also displayed in the tree.

SceneExplorer works in conjunction with SceneComposer, the default editor for J3O files in the jMonkeyEngine IDE. If SceneExplorer doesn't appear when you select "Edit in SceneComposer", choose Window → SceneExplorer from the menu bar to reveal the

window.

## Editing Objects in the scene

1. Select a node in the SceneExplorer window (Open via Window→SceneExplorer if not open)
2. Edit the node in the Properties window (Open via Window→Properties if not open)
3. You can rename a Spatial by right clicking it or by slowly double-clicking the node

## Reorganizing Objects in the scene

1. You can cut, copy and paste Nodes in the SceneExplorer with the normal keyboard commands or the right-click menu of the Nodes
2. You can move single object within the SceneExplorer tree by dragging&dropping them

## Adding Objects to the scene

Right-click a Spatial or Node in the SceneExplorer to add other Spatials like ParticleEmitters or Lights, you can also add UserData to a Spatial that can be read during runtime.

[documentation](#), [sdk](#), [tool](#), [scene](#), [node](#)

</div>

## **title: The SceneExplorer**

# **The SceneExplorer**

## **Adding Node types to SceneExplorer**

If your plugin brings in its own SceneGraph objects you can still have them work like any other SceneExplorer item, including its special properties.

If you want to support special properties of your objects that are not exposed by the SDK automatically, you will have to create your own class that extends org.openide.nodes.Node and implement the interface

com.jme3.gde.core.sceneexplorer.nodes.AbstractSceneExplorerNode. Then you register that class by adding

```
@org.openide.util.lookup.ServiceProvider(service=SceneExplorerNode .
```

above the body of your class. Thats all, your Spatial type will automatically be used and displayed in the SceneExplorer. Make sure you register a jar with the used classes in the plugin preferences under “wrapped libraries”, otherwise the IDE cannot access those classes.

AbstractSceneExplorerNode brings some other useful features you might want to include like automatic creation of properly threaded properties etc. JmeSpatial for example bases on it. A simple SceneExplorerNode example for an object extending Spatial would be JmeGeometry (see below). Editors for special variable types can be added using the SceneExplorerPropertyEditor interface, which can be registered as a ServiceProvider as well.

The SceneExplorerNode can be used for Spatial and Control type objects.

- Add the “Nodes API” and “Lookup API” libraries to your project when you want to use this

## **Spatial Example**

```
@org.openide.util.lookup.ServiceProvider(service=SceneExplorerNode.
public class JmeGeometry extends JmeSpatial {

 private static Image smallImage =
 ImageUtilities.loadImage("com/jme3/gde/core/sceneexplor
private Geometry geom;

public JmeGeometry() {
}

public JmeGeometry(Geometry spatial, SceneExplorerChildren chil
 super(spatial, children);
 getLookupContents().add(spatial);
 this.geom = spatial;
 setName(spatial.getName());
}

@Override
public Image getIcon(int type) {
 return smallImage;
}

@Override
public Image getOpenedIcon(int type) {
 return smallImage;
}

@Override
protected Sheet createSheet() {
 Sheet sheet = super.createSheet();
 Sheet.Set set = Sheet.createPropertiesSet();
 set.displayName("Geometry");
 set.setName(Geometry.class.getName());
 Geometry obj = geom;//getLookup().lookup(Geometry.class);
 if (obj == null) {
 return sheet;
 }

 set.put(makeProperty(obj, int.class, "getLodLevel", "setLoc
 set.put(makeProperty(obj, Material.class, "getMaterial", "s
```

```

 set.put(makeProperty(obj, Mesh.class, "getMesh", "Mesh"));

 sheet.put(set);
 return sheet;

 }

 public Class getExplorerObjectClass() {
 return Geometry.class;
 }

 public Class getExplorerNodeClass() {
 return JmeGeometry.class;
 }

 public org.openide.nodes.Node[] createNodes(Object key, Object
 SceneExplorerChildren children= new SceneExplorerChildren((c
 children.setReadOnly(readOnly);
 return new org.openide.nodes.Node[]{new JmeGeometry((Geomet
 }

}

```



## Control Example

```

@org.openide.util.lookup.ServiceProvider(service=SceneExplorerNode.
public class JmeGhostControl extends AbstractSceneExplorerNode {

 private static Image smallImage =
 ImageUtilities.loadImage("com/jme3/gde/core/sceneexplor
 private GhostControl control;

 public JmeGhostControl() {
 }

 public JmeGhostControl(GhostControl control, DataObject dataObj
 super(dataObject);
 getLookupContents().add(this);
 getLookupContents().add(control);
 }
}

```

```
 this.control = control;
 setName("GhostControl");
 }

@Override
public Image getIcon(int type) {
 return smallImage;
}

@Override
public Image getOpenedIcon(int type) {
 return smallImage;
}

protected SystemAction[] createActions() {
 return new SystemAction[]{
 // SystemAction.get(CopyAction.class)
 // SystemAction.get(CutAction.class)
 // SystemAction.get(PasteAction.class)
 SystemAction.get(DeleteAction.class)
 };
}

@Override
public boolean canDestroy() {
 return !readOnly;
}

@Override
public void destroy() throws IOException {
 super.destroy();
 final Spatial spat=parentNode().getLookup().lookup(Spatial.class);
 try {
 SceneApplication.getApplication().enqueue(new Callable<Void>() {
 public Void call() throws Exception {
 spat.removeControl(control);
 return null;
 }
 }).get();
 }
}
```

```
 ((AbstractSceneExplorerNode)getParentNode()).refresh(tr
 } catch (InterruptedException ex) {
 Exceptions.printStackTrace(ex);
 } catch (ExecutionException ex) {
 Exceptions.printStackTrace(ex);
 }
}

@Override
protected Sheet createSheet() {
 Sheet sheet = super.createSheet();
 Sheet.Set set = Sheet.createPropertiesSet();
 set.displayName("GhostControl");
 set.setName(GhostControl.class.getName());
 GhostControl obj = control;//getLookup().lookup(Spatial.clas
 if (obj == null) {
 return sheet;
 }

 set.put(makeProperty(obj, Vector3f.class, "getPhysicsLocati
 set.put(makeProperty(obj, Quaternion.class, "getPhysicsRotat
 set.put(makeProperty(obj, CollisionShape.class, "getCollisi
 set.put(makeProperty(obj, int.class, "getCollisionGroup", "
 set.put(makeProperty(obj, int.class, "getCollideWithGroups"
 sheet.put(set);
 return sheet;
}

public Class getExplorerObjectClass() {
 return GhostControl.class;
}

public Class getExplorerNodeClass() {
 return JmeGhostControl.class;
}

public org.openide.nodes.Node[] createNodes(Object key, DataObj
```

```
 return new org.openide.nodes.Node[] {new JmeGhostControl((Gr
 }
}
```

◀

▶

## Adding items to the add and tools menus

For adding Spatial, Controls and for general tools there's pre-made abstract classes that you can use to extend the options. Undo/Redo is handled by the abstract class.

**AbstractNewSpatialWizardAction** allows you to show an AWT wizard before creating the Spatial. You can also just implement the base ServiceProvider class and return any kind of action (such as a wizard), in this case you have to handle the threading yourself!

Note that the classes you create are singletons which are used across multiple nodes and you should not store any data in local variables!

To add a new Tool, create a new AbstractToolAction:

```
@org.openide.util.lookup.ServiceProvider(service = ToolAction.class
public class GenerateTangentsTool extends AbstractToolAction {

 public GenerateTangentsTool() {
 name = "Generate Tangents";
 }

 @Override
 protected Object doApplyTool(AbstractSceneExplorerNode rootNode
 Geometry geom = rootNode.getLookup().lookup(Geometry.class)
 Mesh mesh = geom.getMesh();
 if (mesh != null) {
 TangentBinormalGenerator.generate(mesh);
 }
 return geom;
 }

 @Override
 protected void doUndoTool(AbstractSceneExplorerNode rootNode, C
 Geometry geom = rootNode.getLookup().lookup(Geometry.class)
 Mesh mesh = geom.getMesh();
 if (mesh != null) {
 mesh.clearBuffer(Type.Tangent);
 }
 }

 public Class<?> getNodeClass() {
 return JmeGeometry.class;
 }

}
```

For a new Spatial or Control, use AbstractNewSpatialAction

```
@org.openide.util.lookup.ServiceProvider(service = NewSpatialAction)
public class NewSpecialSpatialAction extends AbstractNewSpatialAction {

 public NewSpecialSpatialAction() {
 name = "Spatial";
 }

 @Override
 protected Spatial doCreateSpatial(Node parent) {
 Spatial spatial=new Node();
 return spatial;
 }
}
```

or AbstractNewControlAction:

```

@org.openide.util.lookup.ServiceProvider(service = NewControlAction)
public class NewRigidBodyAction extends AbstractNewControlAction {

 public NewRigidBodyAction() {
 name = "Static RigidBody";
 }

 @Override
 protected Control doCreateControl(Spatial spatial) {
 RigidBodyControl control = spatial.getControl(RigidBodyControl.class);
 if (control != null) {
 spatial.removeControl(control);
 }
 Node parent = spatial.getParent();
 spatial.removeFromParent();
 control = new RigidBodyControl(0);
 if (parent != null) {
 parent.attachChild(spatial);
 }
 return control;
 }
}

```

## Adding using a Wizard

You can create a new wizard using the wizard template in the SDK (New File→Module Development→Wizard). The Action that the template creates can easily be changed to one for adding a Control or Spatial or for applying a Tool. Note that we extend `AbstractNewSpatialWizardAction` here.

A good example is the “Add SkyBox” Wizard:

```

@org.openide.util.lookup.ServiceProvider(service = NewSpatialAction)
public class AddSkyboxAction extends AbstractNewSpatialWizardAction {

 private WizardDescriptor.Panel[] panels;

 public AddSkyboxAction() {
 name = "Skybox..";
 }
}

```

```
}

@Override
protected Object showWizard(org.openide.nodes.Node node) {
 WizardDescriptor wizardDescriptor = new WizardDescriptor(getDescriptor());
 wizardDescriptor.setTitleFormat(new MessageFormat("{0}"));
 wizardDescriptor.setTitle("Skybox Wizard");
 Dialog dialog = DialogDisplayer.getDefault().createDialog(wizardDescriptor);
 dialog.setVisible(true);
 dialog.toFront();
 boolean cancelled = wizardDescriptor.getValue() != WizardDescriptor.CANCELLED;
 if (!cancelled) {
 return wizardDescriptor;
 }
 return null;
}

@Override
protected Spatial doCreateSpatial(Node parent, Object properties) {
 if (properties != null) {
 return generateSkybox((WizardDescriptor) properties);
 }
 return null;
}

private Spatial generateSkybox(WizardDescriptor wiz) {
 if ((Boolean) wiz.getProperty("multipleTextures")) {
 Texture south = (Texture) wiz.getProperty("textureSouth");
 Texture north = (Texture) wiz.getProperty("textureNorth");
 Texture east = (Texture) wiz.getProperty("textureEast");
 Texture west = (Texture) wiz.getProperty("textureWest");
 Texture top = (Texture) wiz.getProperty("textureTop");
 Texture bottom = (Texture) wiz.getProperty("textureBottom");
 Vector3f normalScale = (Vector3f) wiz.getProperty("normalScale");
 return SkyFactory.createSky(pm, west, east, north, south, top, bottom, normalScale);
 } else {
 Texture textureSingle = (Texture) wiz.getProperty("textureSingle");
 Vector3f normalScale = (Vector3f) wiz.getProperty("normalScale");
 boolean useSpheremap = (Boolean) wiz.getProperty("useSpheremap");
 return SkyFactory.createSky(pm, textureSingle, normalScale, useSpheremap);
 }
}
```

```
 }

 }

 /**
 * Initialize panels representing individual wizard's steps and
 * various properties for them influencing wizard appearance.
 */
private WizardDescriptor.Panel[] getPanels() {
 if (panels == null) {
 panels = new WizardDescriptor.Panel[]{
 new SkyboxWizardPanel1(),
 new SkyboxWizardPanel2()
 };
 String[] steps = new String[panels.length];
 for (int i = 0; i < panels.length; i++) {
 Component c = panels[i].getComponent();
 // Default step name to component name of panel. Ma
 // for getting the name of the target chooser to ap
 // list of steps.
 steps[i] = c.getName();
 if (c instanceof JComponent) { // assume Swing comp
 JComponent jc = (JComponent) c;
 // Sets step number of a component
 // TODO if using org.openide.dialogs >= 7.8, ca
 jc.putClientProperty("WizardPanel_contentSelect"
 // Sets steps names for a panel
 jc.putClientProperty("WizardPanel_contentData",
 // Turn on subtitle creation on each step
 jc.putClientProperty("WizardPanel_autoWizardSty
 // Show steps on the left side with the image c
 jc.putClientProperty("WizardPanel_contentDispla
 // Turn on numbering of all steps
 jc.putClientProperty("WizardPanel_contentNumber
 }
 }
 }
 return panels;
}
}
```

The abstract spatial and control actions implement undo/redo automatically, for the ToolActions you have to implement it yourself.

</div>

## **title: Creating a jMonkeyEngine SDK plugin**

# **Creating a jMonkeyEngine SDK plugin**

Note that the creation of a Module Suite is only necessary if you want to upload your plugin to the contribution update center.

## **Using jMonkeyEngine SDK for development**

- Install the “Netbeans Plugin Development” and “NetBeans API Documentation” plugins via Tools→Plugins
- Create a new “Module Suite” project (can be any name, this will be your local “collection” of plugins that you create)
- If no platform is listed, add one by selecting the SDK application folder
  - Mac users have to right-click the jmonkeyplatform application and select “show contents” and then select the jmonkeyplatform folder under Contents/Resources/
- Open the suite, right-click the “Modules” folder and select “Add new..”
- For “Project Name” enter an all-lowercase name without spaces like `my-library`
- Make sure the “Project Location” is inside the module suite folder and press “Next”
- Enter the base java package for your plugin in “Code Name Base” like `com.mycompany.plugins.mylibrary`
- Enter a “Module Display Name” for your plugin like “My Library”
- Press Finish
- To use core SDK or jME3 functions, add “SDK Core” and “SDK Engine” via “Module Properties→Library→Add Dependency”
- Write your plugin (e.g. [create a new editor](#) or [wrap a jar library](#))

## **jMonkeyEngine SDK Contributions Update Center**

If you want your plugin to appear in the “jMonkeyEngine SDK Contributions Update Center” so users can download, install and update it easily via the plugin manager, you can host your plugin in the contributions update center svn repository. The contributions update center is based on a googlecode project for contributed plugins which will be automatically compiled, version-labeled and added to the contributions update center like the core jMonkeyEngine SDK plugins.

Effectively its one large module suite with multiple modules which each represent one plugin, extension library.

## Adding your plugin to the repository

To add your plugin to the repository, do the following:

- Make sure the plugin is part of a “Module Suite” and that its located in the folder of the suite (this saves you from problems with the svn and local version not being configured the same)
- In “Module Properties→Sources”
  - Set the “Source Level” to 1.5 if possible (jMonkeyEngine SDK is compatible to Java 1.5)
- In “Module Properties→API Versioning”
  - Set a specification version for your plugin (like 0.8.1)
  - Set the “implementation version” to “0” and select “append implementation versions automatically”
- In “Module Properties→Display”
  - Enter a purposeful description of your plugin and one of the following categories:
    - For a library plugin: “jME3 - Library”
    - For a SDK plugin: “jME3 - SDK Plugin”
    - For a model loader plugin: “jME3 - Loader”
- In “Module Properties→Build→Packaging”
  - Add your name
  - Add a link to your forum post / home page relating to the plugin
  - Add a license, you can use `../license-jme.txt` to insert the default jME BSD license or use a text file you store in the project folder
- Ask the managers or developers for access to the gc project
- Commit **only the module project** to trunk:
  - Right click the Module Project and select “Versioning → Import into Subversion Repository”
  - Enter `https://jmonkeyplatform-contributions.googlecode.com/svn/trunk` in the URL field
  - Enter your googlecode username and commit password (different than login pass, you can find your password [here!](#)) and press “Next”
  - Check that the “Repository Folder” is `trunk/mypluginfolder` and enter an import message
  - Press “Finish”

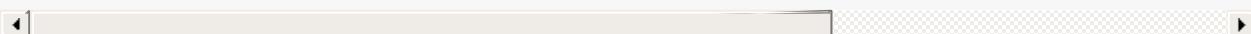
And that's it, from now on each time you commit changes to your module it will be built and added to the contributions center automatically and the version number will be extended by the svn revision number (e.g. 0.8.1.1234)

## Building wrapped library jar files on the server

You can just build your library locally and update and commit the jar file and javadoc/sources zip files to the `release/libs` folder of your plugin in the contrib repo. The users plugins will automatically be updated with the new jar files. You can however also build the library project on the server.

As normally only the module project is being built on the server, any projects that create the actual jar files for library plugins ("normal" projects from the SDK/NetBeans) have to be built from the module build file. To do that simply add the following ant targets to the module build file (adapt to your project file and folder names):

```
<target name="init" depends="basic-init,files-init,build-init,-java
<target name="-build-subproject">
 <ant dir=".//AI" inheritall="false" inheritrefs="false" target="
 <ant dir=".//AI" inheritall="false" inheritrefs="false" target="
 <ant dir=".//AI" inheritall="false" inheritrefs="false" target="
 <zip basedir=".//AI/dist/javadoc" file="release/libs/jME3-ai-jav
 <zip basedir=".//AI/src" file="release/libs/jME3-ai-sources.zip"
 <copy file=".//AI/dist/jME3-ai.jar" todir="release/libs"/>
 </target>
```



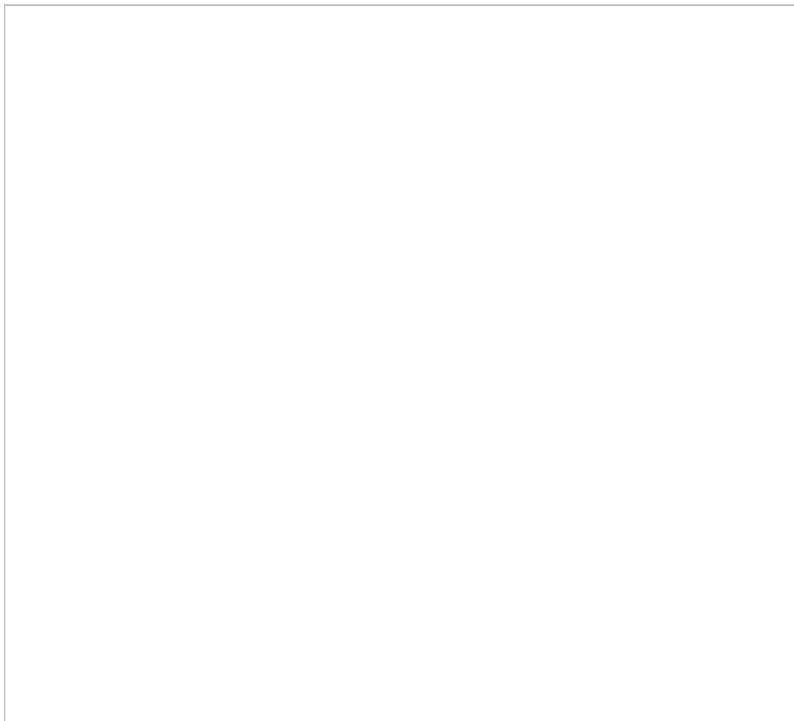
**Note that for the module version number to increase automatically on a commit to the library project, the library project has to be a subfolder of the main module project.**

```
</div>
```

## **title: jMonkeyEngine SDK: Terrain Editor**

# **jMonkeyEngine SDK: Terrain Editor**

The terrain editor lets you create, modify, and paint terrain.

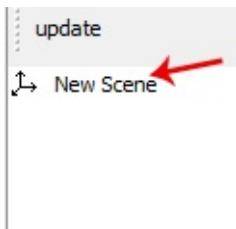


## **Controls**

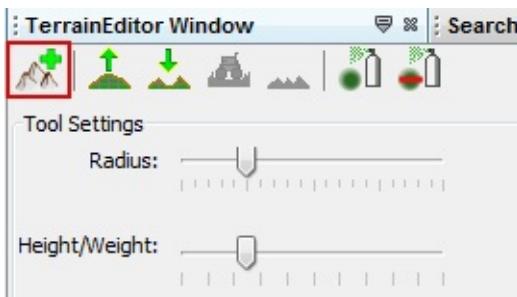
Terrain controls are the same as the Scene Composer, you rotate the camera with the left mouse button and pan the camera with the right mouse button. Until you select one of the terrain tools in the toolbar, then the controls change for that tool. Then left mouse button will use that tool: painting, raising/lowering terrain, etc. The right mouse button might do something, depending on the tool.

## **Creating Terrain**

To create terrain, first select a node (probably your root node) in your scene.



Then click the add terrain button.



This will pop up the Create Terrain wizard that will walk you through the steps for creating terrain. Make sure you decide now how large you want your terrain to be and how detailed you want the textures to be as you cannot change it later on!

In order to see the terrain, you will need to add light to your scene. To do this, right-click the root node in the SceneExplorer window and select “Add Light→Directional Light”

## Step 1: Terrain Size

Here you determine the size of your terrain, the total size and the patch size. You should probably leave the patch size alone. But the total size should be defined; this is how big your terrain is.

## Step 2: Heightmap

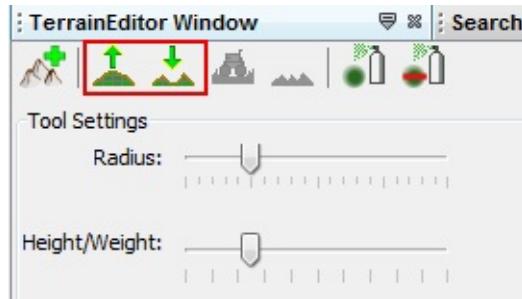
Here you can select a heightmap generation technique to give you a default/random look of the terrain. You can also select a heightmap image to pre-define how your terrain looks. By default, it will give you a flat terrain.

## Step 3: Alpha Image Detail

This step determines how large the alpha blend images are for your terrain. The smaller the alpha image, the less detailed you can paint the terrain. Play around with this to see how big you need it to be. Remember that terrain does not have to be that detailed, and is often covered by vegetation, rocks, and other items. So having a really detailed texture is not always necessary.

# Modifying Terrain

Right now there are two terrain modification tools: raise and lower terrain.



There are two sliders that affect how these tools operate:

- Radius: how big the brush is
- Weight/Height: how much impact the brush has

Once a tool is selected, you will see the tool marker (now an ugly wire sphere) on the map where your mouse is. Click and drag on the terrain to see the tool change the height of the terrain.

# Painting Terrain

Your terrain comes with one diffuse default texture, and you can have up to 12 total diffuse textures. It also allows you to add normal maps to each texture layer. There can be a maximum of 13 textures, including Diffuse and Normal (3 textures are taken up by alpha maps).

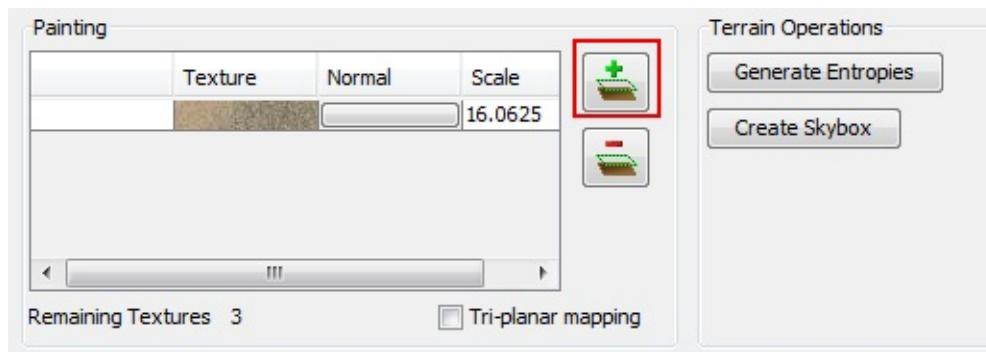
All of the textures can be controlled in the Texture Layer table by clicking on the textures.

There are two sliders that affect how the paint tool operates:

- Radius: how big the brush is
- Weight/Height: how much impact the brush has.

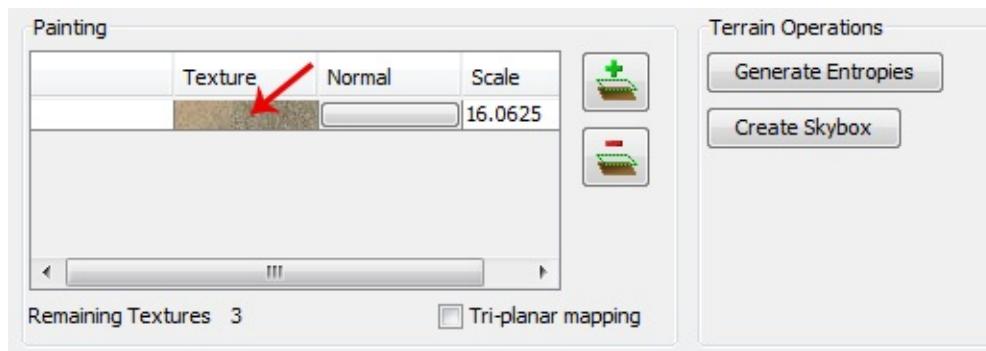
## Adding a new texture layer

Adds a new texture layer to the terrain. The will be drawn over top of the other texture layers listed above it in the list.



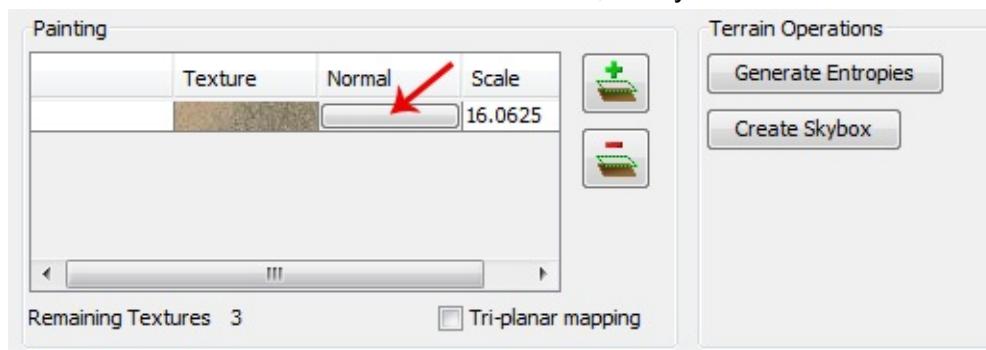
## Changing the diffuse texture

Click on the diffuse texture image in the texture table. This will pop up a window with the available textures in your assets directory.



## Adding a normal map to the texture layer

When you add a texture layer, by default it does not add a normal map for you. To add one, click on the button next to the diffuse texture for that texture layer. This will pop up the texture browser. Select a texture and hit Ok, and you are done.

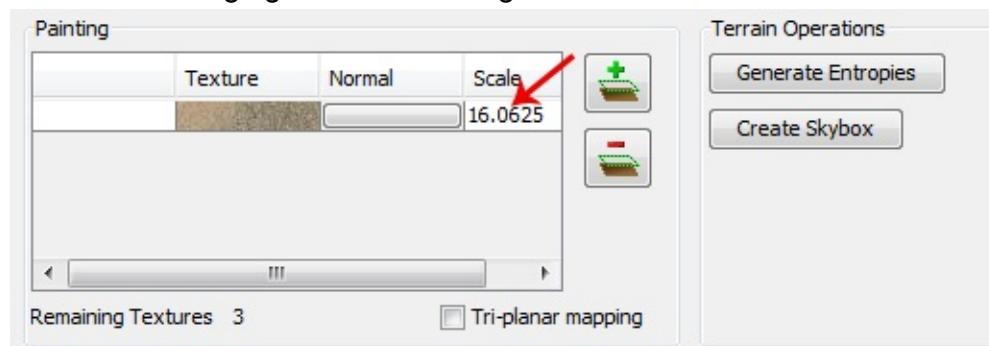


## Removing a normal map from the texture layer

To remove a normal map from the texture layer, hit the normal map button for that texture layer again, and deselect the texture. Then hit Ok and the texture should be removed.

## Changing the texture scale

The field in the table to the right of the diffuse and normal textures for your texture layer is the scale. Changing that value changes the scale of the texture.



You will notice that the scale changes when you switch between Tri-Planar and normal texture mapping. Tri-planar mapping does not use the texture coordinates of the geometry, but real world coordinates. And because of this, in order for the texture to look the same when you switch between the two texture mapping methods, the terrain editor will automatically convert the scales for you. Essentially if your scale in normal texture coordinates is 16, then for tri-planar gets converted like this:  $1/\text{terrainSize}/16$

## Tri-planar texture mapping

Tri-planar texture mapping is recommended if you have lots of near-vertical terrain. With normal texture mapping the textures can look stretched because it is rendered on the one plane: X-Z. Tri-planar mapping renders the textures on three planes: X-Z, X-Y, Z-Y; and blends them together based on what plane the normal of the triangle is facing most on. This makes the terrain look much better, but it does have a performance hit! Here is an article on tri-planar mapping: [http://http.developer.nvidia.com/GPUGems3/gpugems3\\_ch01.html](http://http.developer.nvidia.com/GPUGems3/gpugems3_ch01.html)

## Total texture count

Terrain will support a maximum of 12 diffuse texture. And a combined total of 13 diffuse and normal maps. Most video cards are limited to 16 texture units (textures), and 3 are used behind the scenes of the terrain material for alpha blending of the textures, so you are left with a maximum of 13 textures.

## Generating Terrain Entropies for LOD

If you are using the recommended PerspectiveLodCalculator for calculating LOD levels of the terrain, then you will want to pre-generate the entropy levels for the terrain. This is a slow process. If they are not pre-generated, the LOD control will generate them for you, but this will lag the user when they load the scene, and the terrain will flicker. Use the 'Generate Entropies' button to pre-generate the entropies for the terrain, they will be saved with it. Note

that whenever you modify the height of the terrain, you should re-generate the entropies. Of course, don't do this every time, but maybe just before you are ready to send the map out for testing.

## Loading Terrain Into Your Game

There are a few things your code needs to do to load the terrain.

- You must first use the asset manager to load the scene, see the [hello asset tutorial](#).
- The terrain (as you can see on the left in the editor) is a sub-node of the scene, so you have to write code to investigate the child nodes of the scene until you find the node that is the terrain, see [this tutorial for scene graph concepts](#).
- You also have to set the camera on the LOD control in order for it to work correctly:

```
TerrainLodControl lodControl = ((Node)terrain).getControl(TerrainLodControl.class);
if (lodControl != null)
 lodControl.setCamera(getCamera());
```



[documentation](#), [sdk](#), [tool](#), [terrain](#), [asset](#), [texture](#)

</div>

# **title: Troubleshooting jMonkeyEngine3 SDK**

## **Troubleshooting jMonkeyEngine3 SDK**

### **Graphics Card Driver**

**On Windows and Linux make sure you have the latest driver installed. Make sure its the one supplied by the card manufacturer and not just the OS-default one.** On OSX, make sure you have the latest update for your MacOS.

### **Stability / Graphics issues**

On some Linux and Windows systems, the SDK might perform unstable and quit with native VM crashes or “x errors”. There are a few things one can try to remedy those issues.

### **Heavyweight Canvas**

First of all theres the new “OpenGL” settings page in the SDK global settings where you can enable the “heavyweight” canvas, which solved some issues for some people. The settings panel can be found under Tools→Options on Windows and Linux and in the main menu (or by pressing Apple-Comma) for MacOSX.

If you cannot start the SDK, edit the file `config/Preferences/com/jme3/gde/core.properties` in the SDK settings folder (see above). If it doesn't exist, create the file including all folders. Add the line `use_lwjgl_canvas=true`. To try OpenGL1 compatibility mode (works for both canvas settings) add `use_opengl_1=true`.

### **Look and Feel**

The OS-built-in look and feel might cause issues, you can change the LAF by using the appropriate command line switch (or add it to the [app folder]/etc/jmonkeyplatform.conf file, without the “- -” prefix).

```
--laf javax.swing.plaf.nimbus.NimbusLookAndFeel
```

or alternatively

```
--laf javax.swing.plaf.metal.MetalLookAndFeel
```

## Compiz

Compiz on Linux might cause issues, if you set its rendering quality to “medium” these should go away. \* Appearance→Set Special effects to → “Medium”

## Updating problems

If you have problems updating the SDK, try deleting all files from

`jmonkeyplatform/update/download` and/or `[settings folder]/update/download` depending on your system (see below for the settings folder location).

If you are on Linux, check if the user you run the SDK with has access to the files in

`jmonkeyplatform/jdk/bin` and that they are executable.

## Freezing / Performance problems

If the SDK starts to become sluggish and / or slow or you get unexpected freezes of the application, you can try deleting the cache folder at `var/cache` in the settings folder (see below for the location of the settings folder). Do this while the SDK is not running, then restart the SDK.

## Preferences and Settings

To completely remove and/or reinstall the SDK it is vital that the settings folder is deleted too. The location can be seen through the “about” menu and is as following for the different OS's:

- Windows: `C:\Users\<username>\AppData\Roaming\.jmonkeyplatform`
- Windows (alt): `C:\Users\<username>\.jmonkeyplatform\`
- MacOSX: `/Users/<username>/Library/Application Support/jmonkeyplatform`
- Ubuntu: `/home/<username>/ .jmonkeyplatform`

## Log

To see or post the error output of the SDK in the forum, you can find the log of the application in the settings folder above too, the file is called `var/log/messages.log`

## Getting error messages and reporting issues

When an exception happens in the SDK, a small warning sign appears in the lower right corner of the main window. Double-click it to open a window that allows you to see the exception stack trace. When posting about issues in the forum, always post the stack trace along with a description of what happens and how it can be reproduced.

## Specifying the JDK location

You can install another JDK for use with the jMonkey SDK. You then have to specify the location manually.

1. Go to your jMonkeyEngine SDK installation directory.  
Mac users right-click `jMonkeyApplication.app` (which actually is a directory) in the Finder and select "Show package contents".
2. Navigate to the `etc` directory.  
Mac users navigate to `contents/Resources/jmonkeyplatform/etc/`.
3. Open the file `jmonkeyplatform.conf` in a text editor.
4. Change the following line and enter the path to the JDK:

```
jdkhome="/path/to/jdk"
```

## Freezing at startup

If you're behind a proxy or special network settings, try :

1. disable your network connection
2. launch jme sdk (may wait 30s/1min for timeout)
3. go into tools->options->general
4. setup "manual proxy settings" (for some reason the "Use System F

[discussion](#)

## Known Issues

For a list of known issues and possible workarounds see the following link: [List of known issues on googlecode](#)

[documentation](#), [tool](#), [sdk](#), [faq](#)

</div>

# **title: Automatically Updating jMonkeyEngine SDK**

## **Automatically Updating jMonkeyEngine SDK**

### **Getting stable updates**

The jMonkeyEngine SDK includes an automatic web update feature. To run an update, simply go to Help→Check for Updates and you will get the most current stable update of the SDK and the engine. By default the IDE will check every week for new updates and inform you about them with a symbol in the lower right.

### **Testing the nightly version**

You can test the nightly version with all the latest untested features to give feedback to the developers. Be warned however, the changes in the nightly versions might break your current game project if heavy changes have been committed. To make sure that you do not break your current development environment it is recommended to use a separate application and settings directory for the nightly version.

- Copy the whole application (folder) to a new name like jmonkeyplatform\_nightly.
- Edit the file jmonkeyplatform.conf in the etc directory of the folder. Mac users have to right-click the application and select “Show package contents” and then navigate to Contents/Resources/jmonkeyplatform.
- Change the default\_userdir or default\_mac\_userdir from “\${HOME}/.\${APPNAME}/version” to something like “\${HOME}/.\${APPNAME}/nightly”.

Then start the new application and have your SDK being updated to the most current nightly version:

- Go to Tools→Plugins
- Select the “Settings” tab
- Select the checkbox for “jMonkeyEngine SDK nightly svn”
- Make sure the “force install to shared directories” checkbox is selected
- Select the “Updates” tab
- Press “Reload Catalog”
- Press “Update”

[documentation](#), [sdk](#), [builds](#), [update](#)

</div>

## **title: How to integrate your own jME3 compile in jMonkeyEngine SDK projects**

### **How to integrate your own jME3 compile in jMonkeyEngine SDK projects**

1. [Download jme3 project from svn](#)
2. Make your changes
3. Compile jme3 project
4. Go to Tools → Libraries
5. Press “New Library”
6. Name it “jme3-modified”
7. Press “Add Jar/Folder”
8. Select all JAR files from the `dist` dir of the compiled jme3 version
9. Add the `src` folder of the jme3 project in the “sources” tab
10. Optionally javadoc in the “javadoc” tab
11. Press “OK”
12. Right-click your project and select “Properties”
13. Select “Libraries” to the left
14. Remove the “jme3” library
15. Press “Add Library” and select the “jme3-modified” library

[documentation](#), [sdk](#), [project](#), [builds](#)

</div>

## **title: jMonkeyEngine SDK: Vehicle Creator**

# **jMonkeyEngine SDK: Vehicle Creator**

Best results when car is facing in z direction (towards you).

## **Usage**

1. Select a j3o that contains a vehicle model and press the vehicle button (or right-click → edit vehicle)
2. The VehicleCreator automatically adds a PhysicsVehicleControl to the rootNode of the model if there is none
3. Select the Geometry or Node that contains the chassis in the SceneExplorer and press “create hull shape from selected”
4. Select a Geometry that contains a wheel and press “make selected spatial wheel”, select the “front wheel” checkboxes for front wheels
5. Do so for all wheels

New wheels will get the current suspension settings, you can edit single wheels via the SceneExplorer (update VehicleControl if wheels dont show up) or apply settings created with the settings generator to wheel groups.

Press the “test vehicle” button to drive the vehicle, use WASD to control, Enter to reset.

## **Known Issues**

Don't save while testing the vehicle, you will save the location and acceleration info in the j3o.

## **Code Sample**

Code Example to load vehicle:

```
//load vehicle and access VehicleControl
Spatial car=assetManager.loadModel("Models/MyCar.j3o");
VehicleControl control=car.getControl(VehicleControl.class);
rootNode.attachChild(car);
physicsSpace.add(control);

//then use the control to control the vehicle:
control.setPhysicsLocation(new Vector3f(10,2,10));
control.accelerate(100);
```

[documentation](#), [sdk](#), [tool](#), [asset](#), [editor](#), [physics](#)

</div>

## **title: jMonkeyEngine SDK: Version Control**

# **jMonkeyEngine SDK: Version Control**

Whether you work in a development team or alone: File versioning is a handy method to keep your code consistent, compare files line-by-line, and even roll back unwanted changes. This documentation shows you how to make the most of the SDK's integrated version control features for Subversion, Mercurial, and Git.

For every team member, the file versioning workflow is as follows:

1. Once: Download a working copy of the project from the repository (“checkout”).
2. Regularly: Upload your own changes to the repository (“commit”).
3. Regularly: Download updates by others from the repository (“update”).

Note: Since the jMonkeyEngine SDK is based on the NetBeans Platform framework, you can learn about certain jMonkeyEngine SDK features by reading the corresponding NetBeans IDE tutorials (in the “see also links”).

## **Version Control Systems**

The jMonkeyEngine SDK supports various Version Control Systems such as Subversion, Mercurial, and Git. No matter which of them you use, they all share a common user interface.

You can use file versioning alone or in a team. The advantages are that you can keep track of changes (who did it, when, and why), you can compare versions line-by-line, you can revert changes to files and lines, merge multiple changes to one file, and you can even undelete files.

## **Creating a Repository (Upload)**

Requirements:

- You must have a project that you want to version.
- You must have version control software installed (Subversion, Mercurial, or Git) and have initialized a repository.

- Tip: For Subversion, for example, the init command looks like this example:  

```
svnadmin create /home/joe/jMonkeyProjects/MyGame
```
- The computer where the repository is to be hosted must be available in your team's network, or you will only be able to use your repo locally.
  - Tip: Hosts such as SourceForge, GoogleCode, dev.java.net offer free version control services for open-source projects.

Now you create a repository to store your project's files.

1. In the jMonkeyEngine SDK, right-click the project in the Projects window and choose Versioning > Import Into Subversion Repository (or initialize Mercurial Project, etc, respectively).
  - Tip: If you haven't evaluated yet which system to choose, start with Subversion for now.
2. Go through the wizard and fill in the fields to set up the repository.

## Checking Out a Repository (Download)

You and your team mates check out (download) the repository to their individual workstations.

1. Go to the Team menu and choose Subversion > Checkout (or Git or Mercurial respectively)
2. Fill in your repo data into the wizard and click Finish.
  - A typical repository URL looks like this example:  

```
http://jmonkeyengine.googlecode.com/svn/trunk/engine
```
  - If you want to be able to submit changes, you must have a username and password to this repository. Otherwise leave these fields blank.
3. The repository is downloaded and stored in the location you chose.
4. Use the File > Open Project menu to open the checkout as project and start working.
  - If the checkout is not recognized you need to choose File > New Project from Existing Sources

Of course you can also check out existing repositories and access code from other open-source projects (e.g. SourceForge, GoogleCode, dev.java.net).

## Updating and Committing Changes (Send and Receive)

Receiving the latest changes from the team's repository is referred to as `updating`. Sending your changes to the team's repository is referred to as `committing`.

1. Before making changes, right-click the project and select Subversion > Update to make sure you have the latest revision.
  - Get in the habit of updating regularly, always before you edit a version controlled file. It will spare you much grief.
2. After making changes to the project, make certain your change did not break anything.
  - i. Update, build, run, test.
  - ii. Look at the red/green/blue marks in the editor to review what you have deleted/added/changed. Click the marks to review all differences in a file.
  - iii. Choose Subversion > Show Changes to see all files that were recently changed – by you and other team members.
  - iv. *Update again* in case your team mates made changes while you were reviewing.
3. If there are no conflicts and you want to commit your changes, choose Subversion > Commit from the menu.
4. Write a commit message describing what you changed.
  - Remember, you are writing “a message to your future self”. Never write redundant stuff like “I changed something”.

## Comparing and Reverting Changes

- If you and another committer edited the same line, there is a conflict, and the jMonkeyEngine SDK will show an error message.
  - Right-click a file Choose Subversion > Resolve Conflict
    1. Compare the conflicting versions. Press the buttons to accept or reject each change individually.
    2. After the resolver shows green, save the resolution.
    3. Build and test the resolution, update, and commit.
- Right-click a file and choose Subversion > Search History.
  - You can inspect a file's history and see who changed what, why, and when.
  - You can roll back a file to a historic version if necessary.
- In general, you can choose Subversion > Diff for any file to see two versions of a file next to each other.

## No Version Control? Local History!

If you do not use any version control, you can still track changes in projects to a certain degree.

- Right-click a file or directory and choose Local History to show or revert changes, or undelete files.

- You can also select any two files in the Project window and choose Tools > Diff to compare them.
- Local History only works for files edited in jMonkeyEngine SDK Projects (It does not work for other files, e.g. in the Favorites window.)

See also:

- [Source Code Management with Subversion](#)

[documentation](#), [sdk](#), [editor](#), [tool](#)

</div>

## title: Why not Eclipse?

# Why not Eclipse?

The jMonkeyEngine's default IDE ([Integrated Development Environment](#)) bases on the [NetBeans RCP \(Rich Client Platform\)](#). There is a RCP for Eclipse too, so why don't we use Eclipse as IDE for jMonkey? Well, there's several reasons:

1. Eclipse uses a proprietary [GUI](#) system (SWT), NetBeans uses the Java-default AWT implementation for which a high-performance canvas display exists in LWJGL/jME3. AWT compatibility allowed e.g. integration of plugins like NeoTexture.
2. Eclipse projects are proprietary and they can not be opened without Eclipse. The NetBeans Platform uses the ANT standard which works outside the IDE as well, and can be extended by other build processes that use ANT (e.g. Android deployment). Furthermore, Eclipse can open projects generated by the NetBeans-based jMonkeyEngine SDK if needed.
3. There is no way to extend Eclipse projects properly as there is no global concept of a "Project" in the RCP.
4. Eclipse RCP does not offer a Nodes [API](#) that allows easy wrapping of the SceneGraph into a visual representation.
5. Eclipse RCP only has commercial [GUI](#) editors, NetBeans comes with a free AWT [GUI](#) editor for designing plugins.
6. The two Platforms are the same feature-wise.
7. The core jME3 developers use NetBeans IDE.

If you come from another IDE and want to try jMonkeyEngine SDK, you can easily set up the keyboard shortcut mappings to resemble the configuration you are used to. Profiles exist for Eclipse, IntelliJ and others. Just go to [Settings → Keymap](#) and select one of the existing profiles.

</div>

# Frequently Asked Questions

</div>

## I want to create and configure a jME3 Application

### How do I start writing a preconfigured jME game?

Write a Java class that extends [com.jme3.app.SimpleApplication](#).

**Learn more:** [Hello SimpleApplication](#), [TestAppStateLifeCycle](#).

### How do I change the background color?

```
viewPort.setBackgroundColor(ColorRGBA.Blue);
```

### Can I customize the SimpleApplication class?

Yes! Actually, you MUST customize it! For your own games, you always create a custom base class that extends [com.jme3.app.SimpleApplication](#) class. From now on it's no longer a "simple application" – it's now your game. Configure your [application settings](#), implement methods, and customize away!

**Learn more:** [SimpleApplication](#), [AppSettings](#).

### How can I switch between screens or states?

You should break down your application logic into components by spreading it out over individual AppStates. AppStates can be attached to and detached from the game. AppStates have access to all objects (rootNode, PhysicsSpace, inputManager, etc) and methods in your main application. So each AppState can bring its own subset of input handlers, [GUI](#) nodes, spatial nodes, and even its own subset of game mechanics in the update() loop.

**Learn more:** [Application States](#).

### How do I pause/unpause a game?

You split up your application into several AppStates and implement the `setEnabled()` methods for each state. Then you create, for example, a `GameRunningAppState` and a `GamePausedAppState`. `GamePausedAppState`'s job is to attach all your AppStates that contain the logic and GUI of the pause screen, and to detach all the AppStates that contain logic and GUI of the running game. `GameRunningAppState` does the opposite. By attaching one or the other to the game, you switch between the paused and unpause states.

**Learn more:** [Application States](#).

## How do I disable logger output to the console?

During development, you can switch the severity level of the default logger to no longer print FINE warnings, but only WARNINGS.

```
java.util.logging.Logger.getLogger("").setLevel(Level.WARNING);
```

For the release, switch the severity level of the default logger to print only SEVERE errors.

```
java.util.logging.Logger.getLogger("").setLevel(Level.SEVERE);
```

**Learn more:** [Logging](#).

## Why does the executable crash with "Cannot locate resource"?

Make sure to only load() models converted to .j3o binary format, not the original Ogre or Wavefront formats. If you load assets from zip files, make sure to amend the build script to copy them to the build directory.

**Learn more:** [Asset Manager](#)

## What is java.lang.LinkageError: Version mismatch?

This rare exception shows a message similar to the following: `Exception in thread "LWJGL Renderer Thread" java.lang.LinkageError: Version mismatch: jar version is (number), native library version is (another number)`. jME3 needs native libraries (.dll, .jnilib, lib\*.so files) to run LWJGL and jBullet. The correct versions of these libraries are included when you install the SDK or download the binaries. However there are circumstances where jME3 cannot determine which copy of the native library it should use:

If you install another application that needs a different version of a native library, and this app globally installs its version over jME3's; or if an old copy of a native library is in your project directory, your home directory, or Java library path, or in the classpath; or if you

permanently linked an old copy in your IDE's settings; then Java assumes you prefer these native libraries over the bundled ones, and your jME3 application ends up running with the wrong version.

To fix this, search for .dll (Windows), .jnilib (Mac), and .so (Linux) files for jBullet and LWJGL on your harddrive and in your path and IDE settings, and verify they don't interfere. (If you have other jME versions installed and linked somehow, the outdated natives may also be in a lwjgl.jar or jbullet.jar file!)

## I want to load my scene

### How do I make objects appear / disappear in the 3D scene?

To make a spatial appear in the scene, you attach it to the rootNode (or to a node that is attached to the rootnode). To remove a spatial, you detach it from its parent node.

```
rootNode.attachChild(spatial); // appear in scene
```

```
rootNode.detachChild(spatial); // remove from scene
```

**Learn more:** [The Scene Graph](#), [Hello Node](#), [Hello Asset](#), [Spatial](#), [com.jme3.scene.Node](#) and [com.jme3.scene.Geometry](#).

### Why do I get AssetNotFoundException when loading X ?

First check whether the file path of the asset is correct. By default it is relative to your project's assets directory:

```
// To loadjMonkeyProjects/MyGame/assets/Models/Ninja/Ninja.j3o
Spatial ninja = assetManager.loadModel("Models/Ninja/Ninja.j3o");
```

If you are not using the default `assets` directory, verify that you have registered a locator to the AssetManager. [Different Locator types](#) are available.

```
this.assetManager.registerLocator("assets/", FileLocator.class); //
this.assetManager.registerLocator("c:/jme3User/JMEisSoCool/myAwesomeModel/", ZipLocator.class);
this.assetManager.registerLocator("town.zip", ZipLocator.class);
```

Note that you should not register every single folder containing a texture as the assetmanager will not be able to discern between images with the same name anymore.

**Learn more:** [Asset Manager](#)

## How do I Create 3-D models, textures, sounds?

Follow our best practices for the [multi-media asset pipeline](#).

You create 3-D models in a 3-D mesh editor, for example Blender, and export it in Ogre Mesh XML (animated objects, scenes) or Wavefront OBJ format (static objects, scenes). You create textures in a graphic editor, for example Gimp, and export them as PNG or JPG. You create sounds in an audio editor, for example, Audacity, and export them as WAVE or OGG.  
**Learn more:** [3D Models](#), [Multi-Media Asset Pipeline](#), [JME3's blend-to-j3o importer](#); [Download Blender](#), [Blender intro tutorial](#), [Blender-to-Ogre plugin](#), [Comparison of 3D graphic software features \(Wikipedia\)](#).

## How do I load a 3-D model into the scene?

Use the jMonkeyEngine SDK to convert models from Ogre XML or Wavefront OBJ formats to .j3o binary format. Load the .j3o file using the AssetManager.

```
// To load .../jMonkeyProjects/MyGame/assets/Models/Ninja/Ninja.j3o
Spatial ninja = assetManager.loadModel("Models/Ninja/Ninja.j3o");
```

**Learn more:** [Hello Asset](#), [Asset Manager](#), [com.jme3.assets.AssetManager](#), [com.jme3.scene.Geometry](#), [jMonkeyEngine SDK j3o converter](#),  
**Code sample:** [TestOgreLoading.java](#), [TestOgreConvert.java](#).

## How do initialize the scene?

Use the simpleInitApp() method in SimpleApplication (or initApp() in Application).

**Learn more:** [Hello SimpleApplication](#), [SimpleApplication.java](#).

## I want to transform objects in the scene

### How do I move or turn or resize a spatial?

To move or turn or resize a spatial you use transformations. You can concatenate transformations (e.g. perform rotations around several axes in one step using a Quaternion with `slerp()` or a com.jme3.math.Transform with `interpolateTransforms()`).

```
spatial.setLocalTranslation(1, -3, 2.5f); spatial.rotate(0, 3.14f, 0);
```

**Learn more:** [Hello Node](#), [Spatial](#), [math\\_for\\_dummies](#).

## How do I make a spatial move by itself?

Change the geometry's translation (position) live in the update loop using `setLocalTranslation()` for non-physical and `applyForce()` or `setWalkDirection()` for physical objects. You can also define and remote-control a spatial's motion using [Cinematics](#), e.g. to record cutscenes, or to implement mobile platforms, elevators, airships, etc.

**Learn more:** [Hello Loop](#), [Update Loop](#), [Custom Controls](#), [Cinematics](#)

**Code sample:** [TestBumpModel.java](#), [TestOgreLoading.java](#)

## How do I access a named sub-mesh in Model?

```
Geometry submesh = (Geometry) model.getChild("door 12");
```

**Learn more:** [Spatial](#)

## How do I make procedural or custom shapes?

You can programmatically create `com.jme3.scene.Mesh`'es.

**Learn more:** [Custom Meshes](#)

## I want to change the surface of objects in the scene

### Why is my UV wrapping / texture appearance all wrong?

The most likely reason is the flipping of textures. You may be using the following default method:

```
material.setTexture("ColorMap", assetManager.loadTexture("myTextureName"));
```

You can set the boolean value in the constructor of `TextureKey` to flipped or not flipped. Toggle the boolean to see if it fixes your UV wrapping/texture problem:

```
material.setTexture("ColorMap", this.assetManager.loadTexture(new
```

## How do I scale, mirror, or wrap a texture?

You cannot scale a texture, but you scale the texture coordinates of the mesh the texture is applied to:

```
mesh.scaleTextureCoordinates(new Vector2f(2,2));
```

You can choose among various `com.jme3.texture.Texture.WrapMode`s for individual texture maps of a material: BorderClamp, EdgeClamp, Clamp; MirrorBorderClamp, MirrorEdgeClamp, MirrorClamp; Repeat, MirroredRepeat.

```
material.getTextureParam("DiffuseMap").getTextureValue().setWrap(Wr
```

## How do I change color or shininess of an material?

Use the AssetManager to load Materials, and change material settings.

**Learn more:** [Hello Material](#), [How To Use Materials](#), [Materials Overview](#), [Asset Manager](#).

**Code sample:** [TestNormalMapping.java](#), [TestSphere.java](#).

## How do I make a surface wood, stone, metal, etc?

Create Textures as image files. Use the AssetManager to load a Material and use texture mapping for improved looks.

**Learn more:** [Hello Material](#), [How To Use Materials](#), [Materials Overview](#), [Asset Manager](#),

`com.jme3.assets.AssetManager`, [Blender: Creating Bump Maps and Normal Maps](#)

**Code sample:** [TestSimpleBumps.java](#)

## Why are materials too bright, too dark, or flickering?

If you use a lit material (based on Lighting.j3md) then you must attach a light source to the rootNode, otherwise you see nothing. If you use lit material colors, make sure you have specified an Ambient color (can be the same as the Diffuse color) if you use an AmbientLight. If you see objects, but they are gray or too dark, set the light color to white, or make it brighter (you can multiply the color value with a scalar), or add a global white light source (AmbientLight). Similarly, if everything is too white, tune down the lights. If materials

flicker under a directional light, change the light direction vector. Change the background color (which is independent of light sources) to get a better contrast while debugging a light problem.

## How do I make geometries cast a shadow?

Use `com.jme3.shadow.BasicShadowRenderer` together with `com.jme3.light.DirectionalLight`, and `setShadowMode()`.

**Learn more:** [Light and Shadow](#)

**Code sample:** [TestEverything.java](#), [TestShadow.java](#)

## How do I make materials transparent?

Assign a texture with an alpha channel to a Material and set the Material's blend mode to alpha. Use this to create transparent or translucent materials such as glass, window panes, water, tree leaves, etc.

```
material.getAdditionalRenderState().setBlendMode(BlendMode.Alpha);
```

**Learn more:** [Hello Material](#), [How To Use Materials](#),

## How do I force or disable culling?

While debugging custom meshes, you can switch the

```
com.jme3.material.RenderState.FaceCullMode off to see the inside and outside of the mesh.
```

```
someMaterial.getAdditionalRenderState().setFaceCullMode(FaceCullMode.Off);
```

You can also deactivate the `com.jme3.scene.Spatial.CullHint` of a whole spatial to force jme to calculate it even if it is behind the camera and outside of view.

```
someNode.setCullHint(CullHint.Never);
```

**Learn more:** [Spatial](#)

## Can I draw only an outline of the scene?

Add a renders state to the material's and activate `Wireframe`.

```
material.getAdditionalRenderState().setWireframe(true);
```

Learn more: [Debugging](#).

## I want to control the camera

The default camera `cam` is an instance of the `Camera` class. Learn more: [com.jme3.renderer.Camera](#)

### How do I keep the camera from moving?

- SimpleApplication activates `flyCam` (an instance of `FlyByCamera`) by default. flyCam causes the camera to move with the mouse and the WASD keys. You can disable flyCam as follows:

```
flyCam.setEnabled(false);
```

### How do I switch between third-person and first-person view?

- You can activate the FlyBy Cam as a first-person camera.

Learn more: [Hello Collision](#).

Code sample: [com.jme3.input.FlyByCamera](#)

```
flyCam.setEnabled(true);
```

- You can also create a third-person chase cam.

Learn more: [com.jme3.input.ChaseCamera](#)

Code sample: [jme3test/input/TestChaseCamera.java](#).

```
flyCam.setEnabled(false);
chaseCam = new ChaseCamera(cam, spatial, inputManager);
```

### How do I increase camera speed?

```
flyCam.setMoveSpeed(50f);
```

# Actions, Interactions, Physics

## How do I implement game logic / game mechanics?

Use Controls to define the behaviour of types of Spatial. Use Application States to implement global behaviour, to group subsets of input handlers or [GUI](#) screens, etc. Use the `simpleUpdate()` and `update()` loops for tests and interactions. Use Cinematics to remote-control objects in scenes.

**Learn more:** [Hello Loop](#), [Update Loop](#), [Custom Controls](#), [Application States](#), [Cinematics](#)

## How do I let players interact via keyboard?

Use `com.jme3.input.KeyInput` and a Input Listener.

**Learn more:** [Hello Input](#), [Input Handling](#)

## How do I let players interact by clicking?

Players typically click the mouse to pick up objects, to open doors, to shoot a weapon, etc. Use an Input Listener to respond to mouse clicks, then cast a ray from the player; if it intersects with the bounding volume of a spatial, this is the selected target. The links below contain code samples for both “fixed crosshair” picking and “free mouse pointer” picking.

**Learn more:** [Hello Picking](#), [Mouse Picking](#), [Collision and Intersection](#), [Input Handling](#), `com.jme3.bounding.*`, `com.jme3.math.Ray`, `com.jme3.collision.CollisionResults`.

**Code sample:** [TestRayCollision.java](#)

## How do I animate characters?

Create an animated OgreMesh model with bones in a 3-D mesh editor (e.g. Blender).

**Learn more:** `com.jme3.animation.*`, [Hello Animation](#), [Animation](#), [Blender animation tutorial](#)

**Code sample:** [animation](#)

## How do I keep players from falling through walls and floors?

Use collision detection. The most common solution is to use jme's physics integration, jBullet.

**Learn more:** [Hello Collision](#), [Physics](#), `com.jme3.bullet.*`, `CapsuleCollisionShape` versus `CompoundCollisionShape`, `CharacterControl` versus `RigidBodyControl`.

## How do I make balls/wheels/etc bounce and roll?

Add physics controls to Spatial objects and give them spherical or cylindrical bounding volumes.

**Learn more:** [Hello Physics](#), [Physics](#), com.jme3.bounding.\* , com.jme3.bullet.collisions, com.jme3.bullet.controls.RigidBodyControl,

**Code sample:** [TestSimplePhysics.java](#), [more physics samples](#)

## How do I debug weird Physics behaviour?

Maybe your collision shapes overlap – or they are not where you think they are. Make the collision shapes visible by adding the following line after the bulletAppState initialization:

```
bulletAppState.getPhysicsSpace().enableDebug(assetManager);
```

## How do I make a walking character?

You can use jBullet's CharacterControl that locks a physical object upright, so it does not tip over when moving/walking (as tall physical objects are typically wanted to).

**Learn more:** CharacterControl

Code samples: [TestQ3.java](#) (first-person), [TestPhysicsCharacter.java](#) (third-person)

## How do I steer vehicles?

Use a VehicleControl that supports suspension behavior.

**Learn more:** [Vehicles](#), com.jme3.bullet.\* , VehicleControl

Code samples: [TestFancyCar.java](#), (Press HUJK keys to steer, spacebar to jump.)

## Can objects swing like a pendulums, chains, ropebridges?

Use a PhysicsControl's hinges and joints.

**Learn more:** [Hinges and Joints](#), com.jme3.bullet.joints.PhysicsHingeJoint,

[TestPhysicsHingeJoint.java](#) (Press HK keys to turn, spacebar to swing.)

## Default GUI Display

## What are these FPS/Objects/Vertices/Triangles statistics?

At the bottom left of every default SimpleGame, you see the [StatsView](#) and the FPS (frames per seconds) view. These views provide you with extra information during the development phase. For example, if you notice the object count is increasing and the FPS is decreasing,

then you know that your code attaches too many objects and does not detach enough of them again (maybe a loop gone wild?).

**Learn more:** [StatsView](#)

## How do I get rid of the FPS/Objects statistics?

In the application's simpleInitApp() method, call:

```
setDisplayFps(false); // to hide the FPS
setDisplayStatView(false); // to hide the statistics
```

## How do I display score, health, mini-maps, status icons?

Attach text and pictures to the orthogonal `guiNode` to create a heads-up display ([HUD](#)).

**Learn more:** [HUD](#), com.jme3.font.\* , com.jme3.ui.Picture, guiNode.attachChild()

**Code sample:** [TestOrtho.java](#), [TestBitmapFont.java](#) |

## How do I display buttons and UI controls?

You may want to display buttons to let the player switch between the game, settings screen, and score screens. For buttons and other more advanced UI controls, jME supports the Nifty [GUI](#) library.

**Learn more:** [Nifty GUI](#)

Sample Code: [TestNiftyGui.java](#)

## How do i display a loading screen?

Instead of having a frozen frame while your games loads, you can have a loading screen while it loads.

**Learn more:** [Loading screen](#)

## Nifty GUI

### I get NoSuchElementException when adding controls (buttons etc)!

Verify that you include a controls definition file link in your XML: This is the default:

```
<useControls filename="nifty-default-controls.xml"/>
```

## Where can I find example code of Nifty GUI's XML and Java classes?

<http://nifty-gui.svn.sourceforge.net/viewvc/nifty-gui/nifty-examples/trunk/src/main/>

## Is there Java Doc for Nifty GUI?

[Nifty GUI 1.3 Java docs](#)

## I want to create an environment with sounds, effects, and landscapes

### How do I play sounds and noises?

Use AudioRenderer, Listener, and AudioNode from com.jme3.audio.\*.

**Learn more:** [Hello Audio](#), [Audio](#)

**Code sample:** [audio](#)

### How do I make fire, smoke, explosions, swarms, magic spells?

For swarm like effects you use particle emitters.

**Learn more:** [Hello Effects](#), [Particle Emitters](#), [Bloom and Glow](#), [Effects Overview](#),

com.jme3.effect.EmitterSphereShape, com.jme3.effect.ParticleEmitter

**Code sample:** [TestExplosionEffect.java](#), [TestMovingParticle.java](#), [TestSoftParticle.java](#)

### How do I make water, waves, reflections?

Use a special post-processor renderer from com.jme3.water.\*.

**Learn more:** [Water](#), [Post-Processor Water](#)

**Code sample:** [TestSimpleWater.java](#), [TestSceneWater.java](#), [TestPostWaterLake.java](#),

[TestPostWater.java](#)

### How do I make fog, bloom, blur, light scattering?

Use special post-processor renderers from com.jme3.post.\*.

**Learn more:** [effects\\_overview](#)

### How do I generate a terrain?

Use com.jme3.terrain.\*. The JMonkeyEngine also provides you with a Terrain Editor plugin.

**Learn more:** [Hello Terrain](#), [Terrain](#), [Terrain Editor](#)

**Code sample:** [TerrainTest.java](#)

## How do I make a sky?

**Code sample:** [TestSkyLoading.java](#)

```
rootNode.attachChild(SkyFactory.createSky(assetManager,
 "Textures/Sky/Bright/BrightSky.dds", false));
skyGeo.setQueueBucket(Bucket.Sky)
```

**Learn more:** [Sky](#)

## I want to access to back-end properties

### How do I read out graphic card capabilities?

If your game is heavily using features that older cards do not support, you can [Read Graphic Card Capabilities](#) in the beginning before starting the app, and then decide how to proceed.

```
Collection<com.jme3.renderer.Caps> caps = renderer.getCaps();
Logger.getLogger(HelloJME3.class.getName()).log(Level.INFO, "Capabi
```

## How do I Run jMonkeyEngine 3 with OpenGL1?

In your game, add

```
settings.setRenderer(AppSettings.LWJGL_OPENGL1)
```

to the [AppSettings](#) (see details there).

For the jMonkeyEngine SDK itself, choose Options > OpenGL, and check OpenGL1.

## How do I optimize the heck out of the Scene Graph?

You can batch all Geometries in a scene (or a subnode) that remains static.

```
jme3tools.optimize.GeometryBatchFactory.optimize(rootNode);
```

Batching means that all Geometries with the same Material are combined into one mesh. This optimization only has an effect if you use only few (roughly up to 32) Materials total. The pay-off is that batching takes extra time when the game is initialized.

## How do I prevent users from unzipping my JAR?

Add an [obfuscator to the Ant script](#). The SDK comes with a basic obfuscation script that you can enable in the project settings.

## I want to do maths

### What does addLocal() / multLocal() etc mean?

Many maths functions (mult(), add(), subtract(), etc) come as local and a non-local variant (multLocal(), addLocal(), subtractLocal(), etc).

1. Non-local means a new independent object is created (similar to clone()) as a return value. Use non-local methods if you want to keep using the old value of the object calling the method.
  - Example 1: `Quaternion q1 = q2.mult(q3);`
    - Returns the result as a new Quaternion q1.
    - The involved objects q2 and q3 stay as they are and can be reused.
  - Example 2: `v.mult(b).add(b);`
    - **Watch out:** This calculates the expected result, but unless you actually use the return value, it is discarded!
2. Local means that no new objects are created, instead, the calling object is modified. Use this if you are sure you no longer need the old value of the calling object.
  - Example 1: `q2.multLocal(q3)`
    - Calculates  $q2 * q3$  without creating temp objects.
    - The result is stored in the calling object q2. The old value of q2 is gone.
    - Object q3 stays as it was.
  - Example 2: `v.multLocal(a).addLocal(b);`
    - Calculates the expected result without creating temp objects.
    - The result is stored in the calling object v. The old value of v is gone.
    - The objects a and b stay as they were.

## What is the difference between World and Local coordinates?

World coordinates of a Spatial are its absolute coordinates in the 3D scene (this is like giving GPS coordinates). Local coordinates are relative to the Spatial's parent Spatial (this is like saying, "I'm ten meters left of the entrance").

## How do I convert Degrees to Radians?

Multiply degree value by FastMath.DEG\_TO\_RAD to convert it to radians.

[documentation](#), [faq](#)

</div>

## **title: How to report bugs against the jMonkeyEngine**

New issues should be reported against the [GitHub issue tracker](#). The entire jMonkeyEngine repository on [Google Code](#) is deprecated as of March 2014.

## **How to report bugs against the jMonkeyEngine**

Even if you cannot contribute patches and plugins, you can still help a lot by reporting bugs and helping us improve existing features.

To file a bug report, you must have a [google account](#). If that's something you are strongly opposed to, you may file a report by posting in our [contribution depot for jME3](#), or [here for jME2](#). We do highly prefer the GoogleCode issue tracker though.

</div>

## **Filing a new issue to GoogleCode**

1. Navigate to [jMonkeyEngine's issues list](#).
2. Carefully search through existing jME3 issues. Is your issue not listed? Please continue.
3. Press the `new issue` button in the top left corner, or use this direct link to [file a new issue](#).
4. Simply follow the instructions. Do not apply any new labels unless you know what you're doing.

For more information, see: <http://code.google.com/p/support/wiki/IssueTracker>

## **Filing a new issue in the ‘contribution depot’**

You need a registered account on our forum to post a new thread.

- [jME3 Contribution Depot](#)
- [jME2 Contribution Depot](#)

Please make a descriptive title for your post, and use the template below.

## Bug Report Template

- What did you intend to do, what was the expected outcome?
- What outcome do you experience instead?
- What steps will reproduce the problem?
  1. ...
  2. ...
  3. ...
- What version of the product(s) are you using? **Please copy build details from jMonkeyEngine's About Screen!**
- On what operating system?
- Please provide any additional information. (e.g. your hardware specs and Java version).

[contributor](#)

</div>

## **title: The jMonkeyEngine Logo**

# **The jMonkeyEngine Logo**

You may use this logo for the purposes of education and when identifying the use of jMonkeyEngine. This page comes about due to the reoccurring question of the acquisition and use of the jME logo. It is important to note the distinction in using the logo to represent the use of jMonkeyEngine (allowed and very much appreciated!) as opposed to the endorsement of a product by the jMonkeyEngine community or core team (not allowed).

Original discussion in [this thread](#).

</div>

## **Examples**

</div>

### **Good**



</div>

**Bad**



</div>

## Logos



</div>

# Sample Chapter

Here's some content for the sample chapter.

# Sample page

Sample content in the sample page.

# **second-sample-page**

Just content on a page. Indeed.