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 "//", 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.

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 its 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 provide a fall back drawing for GPUs that do not support shaders.

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.

Syntax of a J3MD file

All J3MD files begin with "MaterialDef" as the root block, following that, is the name of the material def. 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, are specified the parameters. 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 m_TexParam
    Color     m_ColorParam
    Vector3   m_VectorParam
```

```
    Boolean   m_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.`

The Future:

For many reasons, it has been considered to allow to specify a default value for material parameters, inside material definitions, in the case that no value is specified in the material instance. This is especially important for supporting GPUs without shaders and certain other GPUs, where uniform values may "leak" onto other shaders.

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.

The Future:

At some point, it will be permitted to specify .geom files as well, with the GeometryShader statement.

When the material is applied to an object, the shader will have its uniforms set based on

the material parameter values specified in the material instance.
For example, assuming the parameter m_Shininess is defined in the MaterialParameters block like so:

```
MaterialParameters {
  Float m_Shininess
}
```

The value of that parameter will map into the same named uniform in the GLSL shader:
`uniform float m_Shininess;`

The m_ prefix is currently optional, but it has been considered to be made a requirement, so that it is not needed to be specified in the material and will still be named as such in the shader. The letter "m" in the prefix stands for material.

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.

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
```

```
}
```

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:

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

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 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 TextureKey
Texture tex =
assetManager.loadTexture(new TextureKey("Textures/GrassTex.jpg", true));

// Set the parameters
mat.setTexture("m_MyTex", tex);
mat.setFloat("m_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.