# Tesla Engine Material (TEM) Scripting

**Version: 1.1**

## Introduction

Tesla Engine Material (TEM) scripts allow a developer to completely define how an object is rendered in the engine in the form of a easy to read and modify text file. A typical material script will contain a shader effect, parameter configurations, and a set of render states for single-pass or multi-pass rendering techniques. The advantage of separating this logic from your C# code should be obvious, as you can modify how your geometry is rendered without recompiling your application. This document will provide a comprehensive overview of material scripting syntax, a description of available features, and examples.

It is assumed the reader has an understanding of the graphics pipeline and the basic layout of Tesla Engine's graphics/render systems. A great deal of features that a typical material script employs will correspond to these C# objects, data types, and enumerations.

## General Syntax

A material script is very similar to curly-brace programming languages, such as C#, where you have blocks containing statements and potentially other blocks nested inside of them. Statements are each placed on their own line, or separated by a semi-colon if on the same line. Comments are permitted in the script and are ignored by the parser. Only "//" style single-line comments are allowed, the "/* */" comment style is not supported. A comment continues from the "//" prefix to the end of the line.

Additionally, keywords and syntax are **not** case sensitive. Identifiers and some value types (e.g. file paths), however, are case sensitive. Case sensitivity will be explicitly specified in this document where appropiate. Also, there is no defined order of sub blocks, as they can be declared out of order of their usage. For example, parameter declaration blocks can come before an effect declaration.

*Example:*

```
Root {
	//This is a comment
	SubBlock BlockName {
		Statement // So is this
	}

	//Brace placement is unimportant
	SubBlock
	{
		NestedBlock {
			Statement
		}
	}

	SubBlock {
		Statement
		//Two statements separated by a semicolon
		Statement ; Statement
	}
}
```

The following is a broad overview of a typical material script, illustrating all the valid sub blocks. The next section goes into the composition and expected syntax, along with detailed examples of each sub block type. If you understand how the runtime Material object is constructed, then writing material scripts should come easily, as it is a natural extension of creating a material in C# code.

```
Material <name> : <parent_material> {
	Effect {
		….
	}

	MaterialParameters {
		….
	}

	EngineParameters {
		….
	}

	RenderStates {
		….
	}
```

```
        Technique <name> {
            Pass <name> {
                ….
            }
            ….
        }
}
```

## Material Composition

A material consists of the following components:

### 1. A name and optionally, parent material

All materials are required to have a name identifying that material. However, these names do not need to be unique. Every material may have a parent, which means that the child material inherits everything the parent declares. This can be **very** useful in creating **template** TEM files that declare effects, engine parameters, and render states while leaving specific parameter setting to child materials. A child material will start off with everything the parent has and either override existing values or add new ones. This allows you to quickly put together new materials that may only change several parameters, such as a texture, without having to re-write the material in its entirety. This is also recommended as the engine provides materials that can be inherited in its shader library.

Multiple materials can be contained in a single file. Material names need not be unique, but each subsequent material must be separated by at least a white space or placed on a new line. In order to support multiple materials, there are some special considerations when loading TEM files at runtime, as specified in the “Loading Materials” section.

*Syntax:*

*Material <name> : <parent_material> {*

    *…*
*}*

*Examples:*

```
Material MySkybox : Materials/Skybox.tem {

    ...
```

```
}

Material Phong {

      ...
}
```

### 2. A shader effect and a technique to use

Shaders are defined inside an effect, which are contained in a single file such as a HLSL Fx, CgFx, or compiled TEBO file (and so on). An effect is a complete rendering description, it contains a set of shader uniforms, vertex/pixel shaders, and one or more techniques, each with one or more passes. A pass consists of a vertex/pixel shader pair and a number of render states.

*Example:*

```
Effect {
      File : Shaders/MyEffect.tebo
      Technique : MyTechniqueToUse
}
```

There should only ever be one effect block in your material script, as a material can only contain a single effect. The technique specified here has to exist in the effect, although more techniques can exist.

### 3. Material parameters

Parameters reside in a *MaterialParameters* block, their names must match with existing shader uniforms in your effect. Parameters that do not exist in the shader are simply ignored.

*Syntax:*

*MaterialParameters {*
      *<type> <identifier> : <value>*
*}*

*Example:*

```
MaterialParameters {
      float Shininess : 2.0f
}
```

Parameter types match up with C# runtime types almost exactly. Data types are **not** case sensitive. Numerical values that have more than one component have each component separated by a white space. The table below shows all the valid types and expected values.

| Data Type | Value | Comments |
|---|---|---|
| float | 0.5f or 0.5 | The “f” suffix is valid or can be dropped. |
| int | 124 | |
| bool | true / false | Not case sensitive. |
| Vector2 | 25.5 32 | Format: X Y |
| Vector3 | 100.5f 25.02f 19.5f | Format: X Y Z |
| Vector4 | 64.2 0.24 2.5 43.9 | Format: X Y Z W |
| Color | 128 255 64 255 | Clamped to 0-255 range, Format: R G B A |
| Quaternion | Same as vector4 | |
| Matrix | M11 M12 M13 M14<br>M21 M22 M23 M24<br>M31 M32 M33 M34<br>M41 M42 M43 M44 | Matrices are row-major and as thus, their rows are laid out in sequential order |
| Texture1D | Textures/MyTexture.jpg | Case sensitive, as the path/file must exist. |
| Texture2D | Same as Texture1D | |
| Texture3D | Same as Texture1D | |
| TextureCube | Same as Texture1D | |

## 4. Engine parameters

Engine parameters are very similar to material parameters, however they do not have a value associated with them. Instead a shader uniform is bound to an engine value enumeration. This means when the material is used in rendering, the engine will update the shader uniform dynamically with a a value generated by the engine at runtime. A typical engine value is a WorldViewProjection matrix used in transforming vertices from model space to clip space. Engine values tend to be per-frame computations that can only be known at runtime, hence we only bind them to the shader uniform, rather than provide a value.

*Syntax:*

*EngineParameters {*
*<engine_value> : <identifier>*
*}*

*Example:*

```
EngineParameters {
     WorldViewProjection : WVP
     CameraPosition : CamPos
}
```

Engine values are **not** case sensitive, but their spelling corresponds to the **EngineValue** enumeration in the engine, the complete list as follows.

| **EngineValue** | **Comments** |
|---|---|
| WorldMatrix | |
| ViewMatrix | |
| ProjectionMatrix | |
| WorldViewmatrix | |
| WorldViewProjection | |
| WorldInverseTranspose | Used to correctly transform normals of geometry with non-uniform scaling for lighting. |
| WorldMatrixInverse | |
| ViewMatrixInverse | |
| ProjectionMatrixInverse | |
| Viewport | A Vector4 representing the currently active viewport. X = Viewport.X, Y = Viewport.Width,<br>Z = Viewport.Width, W = Viewport.Height |
| Resolution | A Vector2 where X = Viewport.Width, Y = Viewport.Height |
| FrustumNearFar | A Vector2 where X = near, Y = Far |
| AspectRatio | Float equal to Width / Height |
| CameraPosition | Vector3 of the camera's position. |
| CameraDirection | Vector3 (normalized) |
| CameraRight | Vector3 (normalized) |

| CameraUp | Vector3 (normalized) |
|---|---|
| Time | Total time in seconds since the app was started (float) |
| TimePerFrame | Time in seconds since the last frame was rendered (float) |
| FrameRate | Frames per second (float) |
| RandomFloat | Next random float from the engine's random value stream |
| RandomInt | Next random int from the engine's random value stream |

## 5. Render states

Render states are immutable objects that configure the rendering pipeline. There are four types of states: **BlendState**, **RasterizerState**, **DepthStencilState**, and **SamplerState.** Sampler states are defined once per material, while the other three states may be different for each pass in a technique. Therefore, declaring sampler states take a slightly different approach. All render states are declared in a *RenderStates* block.

Blend, rasterizer, and depth stencil states must be declared with an identifier, similar to how you declare parameters. Since render states are immutable once they are created and bound to the pipeline, you can easily re-use them (in fact, it's encouraged). So for a complex material with multiple techniques and passes, you can declare a single render state with a certain configuration and set it in multiple techniques or passes, which will re-use the same object instance at runtime.

There are two options when declaring any of these three states:

1. **Use a pre-defined engine render state.** These are the static properties in each render state which provides different render state configurations. This method uses parameter declaration syntax, where the value is the name of the pre-defined state. The name is **not** case sensitive, but its spelling corresponds to the C# static properties.

2. **Define your own configuration**, which is contained in a nested curly brace block, where each statement corresponds to a property assignment. Property names are **not** case sensitive, but their spelling corresponds to the C# properties of each state object. The syntax is similar to setting engine values. Like with engine values, if the value is an enum, the enum name should be omitted. Not all properties need to be specified, as each omitted property has a default value.

*Syntax:*

```
RenderStates {
	<render_state> <identifier> : <predefined_state>
	<render_state> <identifier> {
		<property_name> : <value>
	}
}
```

*Examples:*

```
RenderStates {
	RasterizerState rastState1 : CullNoneWireframe

	RasterizerState rastState2 {
		Cull : None //Corresponds to CullMode.None
		Fill : Wireframe //FillMode.Wireframe
	}
}

RenderStates {
	BlendState blendOff : Opaque

	DepthStencilState dss {
		DepthEnable : true
		DepthFunction : GreaterEqual //CompareFunction.GreaterEqual
	}

	BlendState blend {
		ColorSourceBlend : SourceAlpha //Blend.SourceAlpha
		ColorDestinationBlend : One //Blend.One
	}
}
```

Every render state is tracked by its identifier. In circumstances where there are duplicate state names of the same state type, the most recently declared state is used. So if a child material defines a render state with an identifier that is also defined in the parent script, the state declared in the child will be used for any techniques or passes defined in the child instead of the one declared in the parent script. An internal cache in the parser maintains unique render states between instances, in order to maximize reusability. If two materials declare the same render state, then only one state is created and shared between the two instances. The predefined states provided by the engine are **always** shared due how they are acquired.

Sampler states on the other hand are not declared with an identifier, instead they use a prefix identifying which pipeline stage and a suffix identifying the sampler index. The maximum sampler index for each stage is defined at runtime by the render system that creates the sampler state graphics objects. Sampler states relate to textures used in your shaders, as they are used to sample textures, which is why they use a different declaration syntax. Like the other render states, you can declare predefined sampler states or create a custom declaration.

*Syntax:*

```
RenderStates {
	<Vertex | Pixel>SamplerState[index] : <predefined_state>
	<Vertex | Pixel>SamplerState[index] {
		<property_name> : <value>
	}
}
```

*Example:*

```
RenderStates {
	PixelSamplerState[0] : PointWrap

	SamplerState[1] : LinearWrap //No prefix is the same as Pixel

	VertexSamplerState[0]  {
		Filter : Linear     //TextureFilter.Linear
		AddressU : Wrap     //TextureAddressMode.Wrap
		AddressV : Wrap     //TextureAddressMode.Wrap
	}
}
```

A complete listing of all predefined render states, their properties, and corresponding values are as follows. Default values for properties are specified in parentheses. Predefined render states, render state properties, and enumeration values are **not** case sensitive.

**BlendState**

| Predefined BlendStates | Comments |
|---|---|
| AdditiveBlend | Alpha/Color SourceBlend set to SourceAlpha,<br>Alpha/Color DestinationBlend set to One |
| Opaque | Default (blending disabled) |
| AlphaBlendNonPremultiplied | Alpha/Color SourceBlend set to SourceAlpha, |

| | |
|---|---|
| | Alpha/Color DestinationBlend set to InverseSourceAlpha |
| AlphaBlendPremultiplied | Alpha/Color SourceBlend set to One,<br>Alpha/Color DestinationBlend set to InverseSourceAlpha |

| Property | Value (default) |
|---|---|
| AlphaBlendFunction | BlendFunction enum (Add) |
| AlphaSourceBlend | Blend enum (One) |
| AlphaDestinationBlend | Blend enum (Zero) |
| ColorBlendFunction | BlendFunction enum (Add) |
| ColorSourceBlend | Blend enum (One) |
| ColorDestinationBlend | Blend enum (Zero) |
| BlendFactor | Vector4 (1.0 1.0 1.0 1.0) |
| MultiSampleMask | Integer (int.MaxValue) |
| BlendEnable[index] | Boolean (index 0 true, rest false) |
| ColorWriteChannels[index] | ColorWriteChannels bitflag \| ColorWriteChannels bitflag (default is All) E.g.:<br>Red \| Green \| Blue |

**RasterizerState**

| Predefined RasterizerStates | Comments |
|---|---|
| CullNone | Cull set to None, VertexWinding set to Clockwise |
| CullBackClockwiseFront | Cull set to Back, VertexWinding set to Clockwise |
| CullBackCounterClockwiseFront | Cull set to Back, VertexWinding set to CounterClockwise |
| CullNoneWireframe | Cull set to None, Fill set to Wireframe, VertexWinding set to Clockwise |

| Property | Value (default) |
|---|---|
| Cull | CullMode enum (Back) |
| VertexWinding | VertexWinding enum (Clockwise) |
| DepthBias | Integer (0) |
| Fill | FillMode enum (Solid) |
| EnableMultiSampleAntiAlias | Boolean (true) |
| EnableScissorTest | Boolean (false) |
| SlopeScaledDepthBias | Float (0.0) |

**DepthStencilState**

| Predefined DepthStencilStates | Comments |
|---|---|
| None | DepthEnable and DepthWriteEnable set to false |
| Default | DepthEnable and DepthWriteEnable set to true |
| DepthWriteOff | DepthEnable set to true, DepthWriteEnable set to false |

| Property | Value (default) |
|---|---|
| DepthEnable | Boolean (true) |
| DepthWriteEnable | Boolean (true) |
| DepthFunction | ComparisonFunction (LessEqual) |
| StencilEnable | Boolean (false) |
| ReferenceStencil | Integer (0) |
| CCWStencilFunction or CounterClockwiseStencilFunction | ComparisonFunction (Always) |
| CCWStencilDepthFail or CounterClockwiseDepthFail | StencilOperation (Keep) |
| CCWStencilFail or CounterClockwiseStencilFail | StencilOperation (Keep) |
| CCWStencilPass or CounterClockwiseStencilPass | StencilFunction (Keep) |
| StencilFunction | ComparisonFunction (Always) |
| StencilDepthFail | StencilOperation (Keep) |
| StencilFail | StencilOperation (Keep) |
| StencilPass | StencilOperation (Keep) |
| TwoSidedStencilEnable | Integer (false) |
| StencilReadMask | Integer (int.MaxValue) |
| StencilWriteMask | Integer (int.MaxValue) |

**SamplerStates**

| Predefined SamplerStates | Comments |
|---|---|
| PointWrap | Filter set to Point, AddressU/V/W set to Wrap |
| PointClamp | Filter set to Point, AddressU/V/W set to Clamp |
| LinearWrap | Filter set to Linear, AddressU/V/W set to Wrap |

| | |
|---|---|
| LinearClamp | Filter set to Linear, AddressU/V/W set to Clamp |
| AnisotropicWrap | Filter set to Anisotropic, AddressU/V/W set to Wrap |
| AnisotropicClamp | Filter set to Anisotropic, AddressU/V/W set to Clamp |

| **Property** | **Value (default)** |
|---|---|
| AddressU | TextureAddressMode enum (Clamp) |
| AddressV | TextureAddressMode enum (Clamp) |
| AddressW | TextureAddressMode enum (Clamp) |
| Filter | TextureFilter enum (Point) |
| MaxAnisotropy | Integer (4) |
| MipMapLevelOfDetailBias or MipMapLoDBias | Float (0.0) |

## 6. Techniques and passes

A technique consists of a number of passes, where each pass describes how to render an object (e.g. a single draw call), which contains the following:

- A set of render states (Blend, DepthStencil, and Rasterizer)
- A vertex and pixel shader (defined in the effect)

In a material script, of these two you can only specify which render states are set per-pass. The identifiers for both techniques and passes **must** match those defined in the effect file. Not all passes need to be specified in the material file; only those that you intend to modify render states for. Passes that are omitted will use the default render states. Likewise, you do not have to specify all three render states in a pass, as the ones you omit will automatically be set to the default render state.

If a technique has multiple passes and you intend to set the same render state for each pass, you can abbreviate the declarations by setting the render states inside the technique block and omitting any nested pass blocks. Render states set at the technique level are used by **all** passes in that effect technique, unless if explicitly overridden.

Additionally, exceptions will be thrown if the sampler state keywords are used in the technique/ pass blocks or if you set a render state that was not declared in a *RenderStates* block.

*Syntax:*

*Technique <name> {*
*<render_state> : <identifier>*
*Pass <name> {*
*<render_state> : <identifier>*
*}*
*}*

*Example:*

```
//Names are defined in the effect, and the render state identifiers
//were declared elsewhere
Technique MyTechnique {
     //BlendState will be used for both Pass0 and Pass1
     BlendState : blend

     //Pass0 uses the default depth stencil state , and the "blend"
     //BlendState specified above
     Pass Pass0 {
          //RasterizerState used only for Pass0
          RasterizerState : rast
     }

     //Pass1 uses the default rasterizer and depth stencil states
     Pass Pass1 {
          BlendState : blendOff //Pass1 uses this blend state instead
     }

     //The technique may contain a "Pass2" (in the effect)
     //that is omitted here, that  //pass would use the "blend"
     //BlendState and the default  //rasterizer and depthstencil
     //states.
}
```

## Loading Materials

The engine's material parser supports loading a TEM file into two different engine objects:

1. A Material object, e.g.

*Material mat = ContentManager.Load<Material>("Materials/MyMaterial.tem");*

This loads the very first material in a file, regardless of how many materials are defined.

2. A MaterialCollection object, e.g.

*MaterialCollection mat = ContentManager.Load<MaterialCollection>("Materials/MyMaterial.tem");*

When dealing with single material scripts, the two methods are nearly identical, as the material collection will only contain a single material. When multiple materials are involved, the first method only loads the first material and caches it in the content manager. The second approach loads all materials as a collection, which is cached. However, the individual materials are **not** cached in the content manager.

## More Examples

Here are some examples putting all of these aspects together to create a complete material:

***A Complete Example:***

```
//All materials must begin with the root material block, and must
//have a name
Material MyMaterial {

     //The technique listed here will be the technique that is used f
     //or rendering
     Effect {
          File : Shaders/MyEffect.tebo
          Technique: MyTechniqueToUse
     }

     MaterialParameters {
          float Shininess : 5.0f
          Vector3 MatDiffuse : 0.25 0.75 1.0
          Texture2D DiffuseMap : Textures/MyTexture.jpg
     }

     //The identifier WVP is a shader uniform, which is a 4x4 matrix.
     //Every frame, the WorldViewProjection matrix is computed and
     //sent to the shader automatically by the engine.
     EngineParameters {
          WorldViewProjection : WVP
     }

     //Render states are declared using an identifier, allowing you
     //to create a state and re-use it for
     //multiple techniques. Once created render states are immutable.
     RenderStates {
          RasterizerState rastState : CullBackCounterClockwiseFront

          PixelSamplerState[0] {
               Filter : Linear
               AddressU : Clamp
               AddressV : Clamp
          }
     }

     //Technique and pass names must match those in the shader effect
     Technique MyTechniqueToUse {
```

```
        //Each pass is allowed a single BlendState,
        //RasterizerState, or DepthStencilState
        //If no state is set, the default ones are used
        Pass pass1 {
            RasterizerState : rastState
        }
    }
}
```

***Example from the Samples (Planet sample):***

```
Material PlanetMaterial {

    Effect {
        File : Shaders/PlanetEffect.tebo
        Technique : Planet
    }

    MaterialParameters {
        Texture2D DiffuseMap : Textures/earth_diffuse.dds
        Texture2D CloudMap : Textures/earth_clouds.dds
        Texture2D Normalmap : Textures/earth_normal.dds
        Texture2D SpecularMap : Textures/earth_specular.jpg
        Texture2D LightsMap : Textures/earth_lights.dds
        bool UseSpecularMap : true
        float SkyScale : 2.5
        float CloudScale : 1.0
        float CloudRotateDirection : - 1.0
        Vector3 IntWaveLength : 5.602 9.478 19.646
        float InnerRadius : 200.0
        float OuterRadius : 205.0
        float InnerRadiusSquared : 40000.0
        float OuterRadiusSquared : 42025.0
        float KrESun : 0.0375
        float KmESun : 0.0225
        float Kr4PI : 0.0314
        float Km4PI : 0.0188
        float Scale : 0.2
        float ScaleOverScaleDepth : 0.8
        float ScaleDepth : 0.25
        float InvScaleDepth : 0.4
        float G : -0.95
        float Gsquared : 0.9025
    }

    EngineParameters {
        WorldViewProjection : WorldViewProj
```

```
        WorldMatrix : World
        CameraPosition : CamPos
    }

    RenderStates {
        RasterizerState rs : CullbackCounterClockwiseFront
        BlendState bs : AdditiveBlend
    }



    Technique Planet {
        Pass SkyFromSpace {
            RasterizerState : rs
            BlendState : bs
        }

        Pass CloudsFromSpace {
            BlendState : bs
        }
    }
}
```

***Parent-Child Example:***

This is a useful example, as you can derive from any of the built-in material scripts that are located in the engine's shader library. Each engine render system defines default content, which is searched if the content manager cannot locate a file. If in your project, you do not redefine "SkyBox.tem" in the root content path, the child material below will load the material by that name from the default content manager. Alternatively, you can access the render system's default content manager yourself.

This is the easiest and quickest way to create materials where you only want to change a color or a texture of one of the built-int materials, and not have to bother with anything else.

*(Parent, defined in Tesla's shader library)*

```
Material SkyBox {

    EngineParameters {
        CameraPosition : CamPos
        ViewMatrix : View
        ProjectionMatrix : Proj
```

```
    }

    Effect {
        File : SkyBoxEffect.tebo
        Technique : Skybox
    }

    RenderStates {
        RasterizerState rs : CullNone
        DepthStencilState dss : DepthWriteOff
    }

    Technique Skybox {
        RasterizerState : rs
        DepthStencilState : dss
    }
}
```

*(Child, defined in the Samples)*

```
Material SpaceSkybox : SkyBox.tem {

    MaterialParameters {
        TextureCube DiffuseMap : Textures/purplenebula.dds
    }
}
```