A material in TSE is a data structure that describes a surface. It contains many different types of information such as sound, physics and rendering properties. At this early release of the TSE, most of the information in materials focuses on rendering.

The TSE generates shaders from Material definitions. The shaders are compiled at runtime and output into the example/shaders directory. Any errors or warnings generated from compiling the procedurally generated shaders are output to the console as well as the output window in the Visual C IDE. If a Material shader fails to compile, notify GarageGames at tsesupport@garagegames.com. Make sure to include the Material definition, and follow bug submission best practices.

Stages

There are many different rendering properties that can be defined in materials and most of them are specific to a particular material stage. Each stage is essentially a different pass that is rendered on a surface. Stages are indicated by the array index of the material parameter. See the example Materials at the end of this chapter. Stages are useful for multi-layer effects.

For instance, if you wanted a special effect where one layer would scroll in one direction and another would scroll on top of it in another direction and they are blended together, stages would be the way to do it.

Another example would be if you wanted the specular component of a material to glow, but not the rest of the material, then you could specify a second stage that had specular and glow on it. The second stage would then be additively blended on top of the first.

Rendering Properties

Base Texture

This is the base diffuse texture.

```
Script reference: baseTex = filename
```

Detail Texture

Detail textures are repeating, high frequency textures that are tiled and modulated over a material. They can give the impression that the material is of a higher resolution than it actually is.

```
Script reference: detailTex = filename
```

Bumpmap Texture

Bumpmaps give a material a more 3D appearance. The TSE supports bumpmaps through the use of normalmaps. There are many free utilities available to create normalmaps. nVIDIA has a nice PhotoShop plugin for this purpose available at developer.nvidia.com.

```
Script reference: bumpTex = filename
```

Environment Map Texture

Not currently supported. See cubemap.

Specular

This value indicates what color the specular highlights on a material should be. Usually this is the same color as the light source that is lighting the material. Specular is additively applied to materials, so this value can also be used to control the intensity of the specular highlight. The darker the color, the less intense the highlight.

```
Script reference: specular = color
```

Specular Power

This is the "shininess" factor of specular highlights. The higher the number, the smaller the specular highlight and the shinier the material appears.

```
Script reference: specularPower = float
```

Pixel and Vertex Specular

These are flags indicating which type of specular, if any, is applied to the material.

Pixel specular is not supported for cards that do not have pixel shader 2.0 compatibility. It is more accurate than vertex specular as it is calculated every pixel, but it is also more expensive.

```
Script reference: pixelSpecular = bool
```

Vertex Specular is not yet supported.

Glow

Flag indicating if material should glow. Glow amount is dictated by how "bright" a material is in color space. White will glow brightest, black will not glow at all.

```
Script reference: glow = bool
```

Emissive

This flag removes any shading calculations performed on a material. It is useful for glowing objects that appear to emit light.

```
Script reference: emissive = bool
```

Translucent

Indicates that the material is translucent.

```
Script reference: translucent = bool
```

TranslucentBlendOp

Indicates the blend operation to perform for a translucent stage. Options include additive, subtractive, multiplicative, and others. See material.h and the BlendOp enum for a complete list.

```
Script reference: translucentBlendOp = blendOp
```

TranslucentZWrite

Translucent materials do not write to the zbuffer by default. This parameter allows this to occur.

```
Script reference: translucentZWrite = bool
```

Cubemap

Useful for simulating reflective surface properties. Cubemaps are six sided "sky box" structures. See the CubemapData and GFXCubemap classes as well as the sample materials.cs file for more info. Also see Direct3D and OpenGL documentation.

```
Script reference: cubemap = filename
```

DynamicCubemap

This is a flag indicating that the material is going to use a cubemap created by the scenegraph. Dynamic cubemaps are updated frequently for very realistice reflections. They are very expensive as a result.

Dynamic cubemaps only currently work for ShapeBase objects. To enable dynamic cubemapping on those objects, the flag dynamicReflection must be set to true for in an object's datablock. The dynamic cubemap is then rendered from that object's center every frame.

```
Script reference: dynamicCubemap = bool
```

Planar Reflections

Marks a material for dynamic planar reflections. For more info see Planar Reflections (*http://www.garagegames.com/docs/tse/general/ch07.php#id2515078*).

```
Script reference: planarReflection = bool
```

Animation Flags

The *animFlags* property determins the combination of animations to perform on that stage. You can combine more than one of the animations like so:

```
animFlags[0] = $scroll | $rotate | $sequence;
```

The flags themselves and their individual special parameters are described here below.

Scroll

Scrolls the material in the direction and speed defined by *scrollDir* and *scrollSpeed* parameters in script.

```
datablock Material(ScrollMat)
{
   baseTex[0] = "demo/data/shapes/spaceOrc/orc_ID1_skin";
   animFlags[0] = $scroll;
   scrollDir[0] = "1 0"; // scroll the texture in the U direction.
   scrollSpeed[0] = 2.0; // that's two texture lengths per second.
};
```

Rotate

Rotates the material with the *rotPivotOffset* and *rotSpeed* parameters indicating the pivot point in texture space and the rotation speed respectively.

```
datablock Material(ScrollMat)
{
   baseTex[0] = "demo/data/shapes/spaceOrc/orc_ID1_skin";
   animFlags[0] = $rotate;
   rotPivotOffset[0] = "-0.5 -0.5"; // this is the center point UV offset for
                                       the rotation.
   rotSpeed[0] = 2.0; // that's two full rotations per second.
};
```

Wave

This is a modifier flag for the *$scroll*, *$rotate*, and *$scale* properties. It can be controlled with the *waveType*, *waveFreq*, and *waveAmp* parameters indicating the type, frequencey and amplitude of the wave. An example of it's use is:

```
datablock Material(ScrollMat)
{
   baseTex[0] = "demo/data/shapes/spaceOrc/orc_ID1_skin";
   animFlags[0] = $rotate | $wave;
   rotPivotOffset[0] = "-0.5 -0.5";
   rotSpeed[0] = 1.0;
   waveType[0] = $sinWave; // This could also be $triangleWave or $squareWave.
   waveFreq[0] = 0.25; // the time in seconds for the wave to complete one full
                          cycle.
   waveAmp[0] = 1.0; // the
};
```

Scale

Scales the material over time and only works with the Wave modifier set. An example of it's use is:

```
datablock Material(ScrollMat)
{
   baseTex[0] = "demo/data/shapes/spaceOrc/orc_ID1_skin";
   animFlags[0] = $scale | $wave;
   waveType[0] = $sinWave;
   waveFreq[0] = 0.25;
   waveAmp[0] = 1.0;
};
```

Sequence

Sequences are a series of images that reside in a single texture. They are arranged in the horizontal of a texture and offset at regular intervals. The sequence texture doesn't need to be square, but it does need to be a power of two to work.

The sequenceFramePerSec parameter indicates how many sequence segments are displayed in a second. The parameter sequenceSegmentSize indicates the size of each segment in a sequence in texture space.

For example for a sequence of 4 32x32 images, create a 128x32 size texture. Then create the images side by side in the image. To show all four images in one second as a looping animation, set sequenceFramePerSec to be 4.0. The sequenceSegmentSize parameter should be set to 0.25 because there are 4 segments (1/4 equals 0.25).

An example of it's use is:

```
datablock Material(ScrollMat)
{
   baseTex[0] = "demo/data/shapes/spaceOrc/orc_ID1_skin";
   animFlags[0] = $sequence;
   sequenceFramePerSec[0] = 0.25;
   sequenceSegmentSize[0] = 0.25;
};
```

Mapping Materials to Textures

Materials are mapped on to objects and interiors through the script command "addMaterialMapping()". It maps the material name specified in a Material datablock to a texture name specified in the relevant art tool. (See materials.cs.) A sample list of mappings can be found in example/demo/data/materialMap.cs. If no mapping is specified then the texture will be rendered unlit (full ambient). The same material can be mapped to multiple textures.

Sample Materials

Example

```
 datablock Material(OrcSkin)
 {
    baseTex[0]       = "demo/data/shapes/spaceOrc/orc_ID1_skin";
    bumpTex[0]       = "demo/data/shapes/spaceOrc/orc_ID1_skin_bump";
    pixelSpecular[0] = true;
    specular[0]      = "1.0 1.0 1.0 1.0";
    specularPower[0] = 4.0;
 };

 datablock Material(GunBlade)
 {
    // STAGE 0
    //----------------------------
    baseTex[0]       = "demo/data/shapes/spaceOrc/gun_ID4_blades";
    bumpTex[0]       = "demo/data/shapes/spaceOrc/gun_ID4_blades_bump";

    // STAGE 1 (second rendering pass)
    //----------------------------
    glow[1]          = true;
    emissive[1]      = true;
    pixelSpecular[1] = true;
    specular[1]      = "0.5 0.5 0.5 0.5";
    specularPower[1] = 32.0;
```

```
   cubemap           = WChrome;
};
```

Custom Materials

Materials offer a lot of options for surface properties, but if even more control is desired, CustomMaterials can be used. CustomMaterials allow the user to specify their own shaders via the ShaderData datablock.

CustomMaterial Properties

CustomMaterials are derived from Materials, so they can hold a lot of the same properties, but it is up to the user to code how these properties are used.

texture[x]

Specifies either a texture filename, or a texture coming from the scenegraph. The "x" is a number from 0 to the max number of texture units supported on the hardware the CustomMaterial shader is targeting (see the version property description, below).

There are several textures that can be grabbed from the scenegraph and used in shaders. Here is a list:

$lightmap

Interior lightmap

$normmap

Interior light-normal map (not a bumpmap, but used for bumpmapping)

$fog

The fog texture generated by the scenegraph

$cubemap

The cubemap specified with the cubemap parameter of the CustomMaterial $dynamicCubemap

Cubemap passed in from scenegraph

$backbuff

A copy of the screenbuffer - useful for refraction effects

shader

The shader parameter indicates the pixel and/or vertex shader to be used with the CustomMaterial. This parameter expects a ShaderData datablock. See Shaders.

version

Each CustomMaterial datablock targets a specific level of pixel shader hardware. The version parameter indicates this level. Ie. nVIDIA's Geforce 2 would be 0.0, Geforce 3 and 4ti's would be 1.1, the Geforce FX cards would be 2.0, and the Geforce 6xxx cards would be 3.0.

fallback

High level CustomMaterials (ones targeting 3.0 or 2.0 pixel shader hardware) can specify fallbacks for lower levels of hardware. This allows complete control over how a material looks on each possible platform. The fallback parameter is simply another CustomMaterial that is targeting a

lower level hardware.

When mapping a CustomMaterial to a texture, the highest level CustomMaterial in the fallback chain should be specified. That way the TSE can follow the fallback chain down until it reaches a material that can be rendered.

pass

In addition to targeting a specific level of hardware, each CustomMaterial also targets just a single rendering pass. If more passes are desired they can be specified using the pass parameter. It just takes another CustomMaterial definition.

Sample CustomMaterial

Example

```
datablock CustomMaterial( ShinyMetal2_0 )
{
   texture[0] = "test/metalBump";   // bumpmap texture in texture unit 0
   texture[3] = "$cubemap";         // cubemap texture in texture unit 3

   cubemap    = Lobby;
   shader     = BumpCubemap;
   version    = 2.0;
   fallback   = ShinyMetal1;          // specify fallback for 1.1 hardware
   pass[0]    = ShinyMetal2_1;        // specify second pass

   specular = "1.0 1.0 1.0 0.0";    // can use the specular parameter from
                                       Material
   specularPower = 8.0;
};
```

Material Instances

Materials can be placed on any type of surface in the TGE. Since different types of surfaces have different underlying geometry and lighting data, they need to generate different types of shaders even though the Material may be the same. The mapping of a material to different types of geometry is handled through a material instance (MatInstance).

MatInstances take information from the Material, the scenegraph, the surface, and the target hardware and filter it down into shader passes before sending it off to the shader generation module. They also store pointers to the shaders created and use them later when the TSE sets up render states for the material.