3D Node Appearance

Description

Candera::Appearance groups following render attributes:

- Candera::Material,
- Candera::Texture,
- Candera::RenderMode
- Candera::Shader and
- Candera::ShaderParamSetter.

Render attributes define the distinctive visualization of a geometry like Candera::Mesh, Candera::Billboard, and Candera::PointSprite. Further, render attributes can be shared across multiple objects, which conserve memory and enable sharing of appearance characteristics.

A Candera::Appearance object is mandatory for any object in order to get rendered. Per default no Appearance attributes are attached.

If the Appearance is activated, all render attributes that are set become activated. If no RenderMode is defined (null), then the default RenderMode is used instead.

Appearance: Material

Material Attributes

Candera::Material describes the color attributes of an object's surface and is primarily used for lighting computations. If no material is set, lighting calculations in associated shaders can not be applied.

The color attributes are defined as follows:

Ambient	RGB color that interacts with the ambient attribute of light.
Emissive	RGB color that defines the self-lighting of the material. This color is visible, even when Material is unlighted. The emissive color attribute does neither interact with any type of light source nor with other 3D objects.

Diffuse	RGBA color that interacts with the diffuse attribute of light. The alpha value of the diffuse color, defines the alpha factor of the entire Material, supposed that alpha blending is enabled (For details see Candera::RenderMode).
Specular	RGB color that interacts with the specular attribute of light.

Furthermore with Specular Power the sharpness of a specular highlight, if lit by specular light, can be defined.

Create a new Material

```
    m_material = Material::Create();
```

Set Colors

This sample code shows how to change the ambient color, the code for changing the diffuse color, the emissive color or the specular color looks similarly.

```
    // Set ambient color
    if (node->GetAppearance() != 0 && node->GetAppearance()->GetMaterial() != 0){
        node->GetAppearance()->GetMaterial()->SetAmbient(m_color);
    }
    // end Set ambient color
```

Appearance: Texture

Description

Candera::Texture encapsulates a Candera::TextureImage and a set of attributes specifying how it is applied to a vertex's texture coordinate. Candera implements a sharing mechanism based on TextureImages. These hold the actual VRAM handle. Multiple textures can share one TextureImage object. The TextureImage manages its upload to the VRAM.

Attention:

- The Texture is only useable with an associated TextureImage.
- Do not share TextureImages, if TextureImage modification shall not be shared across Textures.
- In order to support MipMapping or repeated wrapping, the bitmap's width and height must be a power of 2 (n^2), like e.g. 2, 4, 8, 16 ,32, etc. However, height and width can be different, e.g.: 256x128.

Texture Filtering

- **Minification and magnification:**

Following texture filters can be used to improve visual quality of Textures: Magnification to upscale, Minification to downscale.

Nearest	Value of the texel that is nearest to the center of the pixel being textured.
Linear	The weighted average of the four texels that are closest to the center of the pixel being textured.

- **Mipmapping:** MipMapping can be used to avoid texture aliasing effects.

None	MipMapping disabled
Nearest	MipMapping uses nearest mip map level
Linear	MipMapping uses bilinear mipmap interpolation of the two nearest mip map levels

- **WrapMode:**

It specifies how the texture is wrapped:

Repeat	Repeat the texture
ClampToEdge	Clamp fetches to the edge of the texture
Linear	MipMapping uses bilinear mipmap interpolation of the two nearest mip map levels

- **MaxAnisotropy:**

It specifies the maximum degree of anisotropy to account for in texture filtering for the Texture object. Anisotropic filtering improves the quality of the textures when they are not uniformly scaled because, for example, the textured triangle is not exactly facing the camera. The value of maxAnisotropy must be greater or equal to 1.0f (isotropy) and is limited by the max detail of anisotropy supported by hardware, which can be retrieved with the function Candera::RenderDevice::GetMaxAnisotropySupportedByDevice.

Changing Texture Image And Setting Filters

```
    m_textureImage = BitmapTextureImage::Create();
    m_textureImage->SetName("LabelTextureImage");
    static_cast<void>(m_textureImage->SetBitmap(m_bitmap));
    if (m_textureImage == 0) {
        FEATSTD_DEBUG_ASSERT(false);
        return; // memory is freed in destructor/UpdateBitmap()
    }


    m_texture = Texture::Create();
    m_texture->SetName("LabelTexture");
    m_texture->SetTextureImage(m_textureImage);
    m_texture->SetWrapModeU(Texture::ClampToEdge);
    m_texture->SetWrapModeV(Texture::ClampToEdge);
    m_texture->SetMinificationFilter(Texture::MinMagLinear);
    m_texture->SetMagnificationFilter(Texture::MinMagLinear);
```

Appearance: Render Mode

Description

Candera::RenderMode is an Appearance component that encapsulates polygon-level and per-fragment compositing render attributes. If an object's Appearance has set RenderMode to null, then the default RenderMode is used instead. For details how to set the default render mode, see Candera::Renderer::SetDefaultRenderMode.

If a Camera has a RenderMode attached [camera->GetApperance()->SetRenderMode(...)], then the Camera's RenderMode overrules the DefaultRenderMode during its render pass. Thus, all Nodes rendered by the Camera that do not have their own RenderMode set, use the RenderMode applied to the Camera.

If a RenderMode has set an inheritance bit for a certain render attribute, then the property of the base render mode is used, this is either the Camera's RenderMode if set or the DefaultRenderMode otherwise.

```
        bool m_isColorWriteRedEnabled;       // Default value: true
        bool m_isColorWriteGreenEnabled;     // Default value: true
        bool m_isColorWriteBlueEnabled;      // Default value: true
        bool m_isColorWriteAlphaEnabled;     // Default value: true
        bool m_isDepthWriteEnabled;          // Default value: true
        bool m_isDepthTestEnabled;           // Default value: true
        bool m_isStencilTestEnabled;         // Default value: false
        bool m_isBlendingEnabled;            // Default value: false
```

Set a rendermode:

```
SharedPointer<RenderMode> rm = RenderMode::Create();
rm->SetBlendingEnabled(true);
rm->SetBlendMode(RenderMode::SourceAlpha, RenderMode::InverseSourceAlpha, RenderMode::Add);
rm->SetDepthTestEnabled(false);
rm->SetDepthWriteEnabled(false);
appearance->SetRenderMode(rm);
```

Render Mode Attributes

- **blending**

The next chapter Color Blending describes possible blending operations in detail.

- **culling**

Culling determines which side of a polygon is removed before rasterisation.

FrontFaceCulling	Front of the polygon is removed
BackFaceCulling	Back of the polygon is removed
NoCulling	Front and back of the polygon are rendered

- **winding**

Winding defines the front face of a polygon. A polygon side is the front-face if its screen-space vertices are in the same order as the winding specifies.

ClockWise	Clockwise ordered vertices define the front of the polygon.
CounterClockWise	Counterclockwise ordered vertices define the front of the polygon.

Candera::RenderMode::ComparisonFunction can be used with depth-, stencil-, or sampler state operations. It specifies how the source (new) data is compared against the destination (existing) data before passing the comparison operations (storing the data).

CompareNever	Never pass the comparison.
CompareLess	If source data is less than destination data, the comparison passes.
CompareEqual	If source data is equal than destination data, the comparison passes.

CompareLessEqual	If source data is less than or equal to destination data, the comparison passes.
CompareGreater	If source data is greater than destination data, the comparison passes.
CompareNotEqual	If source data is not equal to destination data, the comparison passes.
CompareGreaterEqual	If source data is greater than or equal to destination data, the comparison passes.
CompareAlways	Always pass the comparison.

- **depth bias**

Depth bias that can be applied to co-planar primitives to reduce z-fighting, according to following function: *Depth bias = (max * scaleFactor) + (r * units)*; where *max* is the maximum depth slope of the triangle being rendered and *r* is an implementation-defined constant that is guaranteed to produce the smallest resolvable offset.

> olygons that are coplanar can be made to appear not coplanar by adding a z-bias to each one. This is a technique commonly used to ensure that shadows, decals, or hidden-line images on coplanar surfaces are displayed properly. The depth bias is added before the depth test is performed but does not influence the original depth value written into depth buffer.

Via Candera::RenderMode::SetDepthBias the depth bias can be set and Candera::RenderMode::SetDepthTestEnabled and Candera::RenderMode::SetDepthWriteEnabled enables it.

- **stencil buffer**

StencilPlane specifies the face associated with the provided stencil function, which are *FrontFace*, *BackFace* and *FrontAndBackFace*. Front face stencil affects non-polygons and front-facing polygons whereas back face stencil affects back-facing polygons only.
StencilOperation defines the stencil-buffer operation.

SetToZero	**Set the stencil-buffer entry to zero.**
Keep	Do not update the entry in the stencil buffer.
Replace	Replace the stencil-buffer entry with the reference value.
Increment	Increment the stencil-buffer entry, clamping to 2^n where n is the number of bits in the stencil buffer.

Decrement	Decrement the stencil-buffer entry, clamping to zero.
Invert	Invert the bits in the stencil-buffer entry.
IncrementWrap	Increment the stencil-buffer entry, wrapping to zero if the new value exceeds the maximum value
DecrementWrap	Decrement the stencil-buffer entry, wrapping to the maximum value if the new value is less than zero.

Appearance: Render Mode - Color Blending

Description

When Blending is enabled, the output from the fragment shader is blended with the current contents of the frame buffer, rather than merely overwriting it. Blending is mostly used to make objects appear transparent, but can also produce various other effects. (eg. anti-aliasing, DOF, or multi-pass rendering.) There are multiple ways of using blending, but generally, the procedure is as follows:

- Set alpha value of a node
- Set alpha value of a material color
- Set alpha value of the material (this changes the alpha value of the materials diffuse color)
- Enable blending via RenderMode and use the alpha value of the texture in combination with Candera::RenderMode::BlendMode

Blend Operations

```
enum BlendOperation
{
    Add = 0,
    Subtract = 1,
    ReverseSubtract = 2,
    Min = 3,
    Max = 4
};
```

Blend Factor

```
enum BlendFactor
{
    Zero = 0,
    One  = 1,
    SourceColor = 2,
    InverseSourceColor = 3,
    SourceAlpha = 4,
    InverseSourceAlpha = 5,
    DestColor = 6,
```

```
            InverseDestColor = 7,
            DestAlpha = 8,
            InverseDestAlpha = 9,
            ConstantColor = 10,
            InverseConstantColor = 11,
            ConstantAlpha = 12,
            InverseConstantAlpha = 13,
            SourceAlphaSaturate = 14
        };
```

BlendMode combines the source and destination blend factors with the operation used in blending equation.

- RGB blending equation: *final_RGB = source_RGB * sourceRGBFactor (+operationRGB+) destination_RGB * destRGBFactor*.
- Alpha blending equation: *final_alpha = source_alpha * sourceAlphaFactor (+operationAlpha+) destination_alpha * destAlphaFactor*.

For the Blending Options ConstantColor, InverseConstantColor, ConstantAlpha, or InverseConstantAlpha a blending color can be set.

Set blending:

```
    SharedPointer<RenderMode> rm = RenderMode::Create();
    rm->SetBlendingEnabled(true);
    rm->SetBlendMode(RenderMode::SourceAlpha, RenderMode::InverseSourceAlpha, RenderMode::Add);
    rm->SetDepthTestEnabled(false);
    rm->SetDepthWriteEnabled(false);
    appearance->SetRenderMode(rm);
```

Appearance: Shader

Description

Candera::Shader is an Appearance component representing a graphical processing unit (GPU) program, thus a vertex and fragment shader pair.

- The Shader program can be parametrized with uniforms and attributes. It's guaranteed by the shader that the program is only created once in VRAM.
- Shader is a device object that counts its uploads to VRAM automatically. Thus, a shader is only uploaded if it has not been uploaded before.
- Further, a shader is unloaded from VRAM, if the upload count is decreased to zero. Take care, that upload is invoked as many times as unload.

Candera and SceneComposer offer reference shaders of both vertex and fragment shaders.

For details refer to section Shader Usage.

In SceneComposer via the appearance of a node its ShaderProgram can be chanced easily. Further it is possible to change shaders, the default shader configuration, to create new shaders and combine two of them to a new ShaderProgram.

The Default Shader Configuration in SceneComposer

Node	Vertex Shader	Fragment Shader
Billboard	RefTrans	RefTex
Mesh without texture	RefTransLight1	RefColor
Mesh with texture	RefTransLight1	RefColorTex
MorphingMesh without texture	RefTransLight1Morph	RefColor
MorphingMesh with texture	RefTransLight1Morph	RefColorTex
PointSprite	RefTransPointSprite	RefPointSprite
SkyBox	RefTransCubeMap	RefCubeMapTex

Appearance: Shader Parameter Setters

Description

The Candera::ShaderParamSetter class is used to pass uniform parameters to a Shader program.

The most relevant methods for this class are Candera::ShaderParamSetter::SetUniform which creates a new uniform (see example below):

```
    shaderParamSetter->SetUniform(m_uniformName, Shader::Float, &m_uniformValue);
```

and the Candera::ShaderParamSetter::Activate which activates all the uniforms previously set by the SetUniform method in the Shader.

> In a class derived from ShaderParamSetter, the methods which calculates the auto-uniforms (e.g. ModelViewProjectionMatrix4), should be invoked in the Activate method.

Generic Shader Param Setters

Candera::GenericShaderParamSetter is a class derived from Candera::ShaderParamSetter which bundles uniform shader parameters that are calculated by Candera. The class interface offers, for each of these parameters, a specialized method which enables / disables the parameter.

```
    shaderParamSetter->SetModelMatrix4Enabled(true);
    shaderParamSetter->SetModelMatrix3Enabled(true);
    shaderParamSetter->SetNormalModelMatrix3Enabled(true);
    shaderParamSetter->SetModelViewMatrix4Enabled(true);
    shaderParamSetter->SetModelViewMatrix3Enabled(true);
    shaderParamSetter->SetNormalModelViewMatrix3Enabled(true);
    shaderParamSetter->SetModelViewProjectionMatrix4Enabled(true);
    shaderParamSetter->SetCameraLookAtVectorEnabled(true);
    shaderParamSetter->SetCameraPositionEnabled(true);
    shaderParamSetter->SetLightActivationEnabled(true);
    shaderParamSetter->SetMaterialActivationEnabled(true);
    shaderParamSetter->SetTextureActivationEnabled(true);
    shaderParamSetter->SetLightsCoordinateSpace(Light::World);
```

If a parameter is enabled then it will be calculated and passed to the shader, if disabled the parameter is neither calculated nor passed.

Default Parameters

Some often needed shader parameters are enabled by default:

- ModelViewProjectionMatrix4
- LightActivation
- MaterialActivation
- TextureActivation

If needed, for performance reasons, those parameters might be disabled.

Generic Param Setters in SceneComposer

In SceneComposer the generic parameter setters are accessible through the graphic interface in two ways:

- as customizable uniform setter, from the Toolbox panel in the Attachments bar. Once added in the Appearance node list this uniform setter should be configured in the Properties panel by enabling the desired uniforms.
- as a predefined uniform setter, from the Templates panel in the UniformSetters list.

SceneComposer provides the following predefined uniform setters:

- TransAnisotropicLightShaderParamSetter
- TransLightBumpMapShaderParamSetter
- TransLightShaderParamSetter

- TransLightSphereMapShaderParamSetter
- TransShaderParamSetter

The table below shows some examples of shaders that can be set by each of the predefined shader parameter setters:

Shader Param Setter	Shader Program
TransAnisotropicLightShaderParamSetter	RefTransAnisotropicLight1_RefAnisotropicLight1SpecularTex
TransLightBumpMapShaderParamSetter	RefTransLight1BumpMap_RefLight1BumpMap
TransShaderParamSetter	RefTransLight1_RefColor
TransLightSphereMapShaderParamSetter	RefTransLight1SphereMap_RefColorTex
TransLightShaderParamSetter	RefTransWorldLight1_RefColor

Shader parameter setters can be applied to a 3D node by selecting it in the panel and then drag-and-drop it in the Appearance list of the node.

Customized Shader Parameter Setter

Customized Shader Parameter Setter - Example

If for any reason the standard shader parameter setters are not sufficient, it is possible to create a customized shader parameter setter. However, a custom shader parameter setter cannot be configured in SceneComposer, it must be associated to the desired node by coding.

The following example explains how to create and use a customized shader parameter setter in a widget. Refer to the

- *ShaderParamSetterWidget* in combination with the
- *ShaderParamSetterSolution*,
  both present in the *cgi_studio_player* folder.

The widget *ShaderParamSetterWidget* allows to set the u_Material.emissive and the u_MVPMatrix uniform parameters for any node using an appropriate shader program (for example, the nodes in the ShaderParamSetterSolution are using the **RefTransLight1_RefColor** shader program).

Simple Shader Parameter Setter for u_Material.emissive & u_MVPMatrix Uniforms

In order to pass the u_Material.emissive and u_MVPMatrix uniforms to a shader program, the widget uses an instance of the class *SimpleShaderParameterSetter*:

```
    // Create uniform setter instance
    // m_shaderParamSetter is of type SharedPointer<SimpleShaderParamSetter> ;
```

```
    m_shaderParamSetter = SimpleShaderParamSetter::Create();
    // end Create uniform setter instance
```

The only method that needs to be overridden in the SimpleShaderParameterSetter is Candera::ShaderParamSetter::Activate.

The example *ShaderParamSetterWidget* uses this SimpleShaderParameterSetter by attaching the created instance to the node, which is associated to the widget:

```
    // Attach uniform setter
    if ( (GetNode() != 0) && (GetNode()->GetAppearance() != 0) ) {
        GetNode()->GetAppearance()->SetShaderParamSetter(m_shaderParamSetter);
    }
    // end Attach uniform setter
```

Whenever the widget property "Uniform Color" is modified, the widget sets the given color value (m_color) as u_Material.emissive uniform to the SimpleShaderParameterSetter:

```
    // Set u_Material.emissive uniform
    FEATSTD_UNUSED(m_shaderParamSetter->SetUniform(ShaderParamNames::GetUniformName(ShaderParamN
    // end Set u_Material.emissive uniform
```

The node vertices are mapped from the model space to the screen space using the Model-View-Projection Matrix (u_MVPMatrix). SimpleShaderParameterSetter calculates the u_MVPMatrix using the node and camera attributes and then passes it to the shader:

```
    // Set u_MVPMatrix uniform
    const Matrix4 mvpMatrix = node.GetWorldTransform() * RenderDevice::GetActiveCamera()->GetVie
    return shader.SetUniform( mvpMatrix.GetData(), ShaderParamNames::GetUniformName(ShaderParamN
    // end Set u_MVPMatrix uniform
```

Modifications made using the SetUniform method become effective each time the camera renders the scene because the method Activate is invoked internally by the node.

Multipass Appearance

Description

A Candera::MultiPassAppearance is a dedicated Appearance which enables a node to be rendered multiple times with different appearance settings which are blended in order to achieve a certain

visual result like e.g. fur or blurred rendering.

The sequence of render passes is created by chaining instances of MultiPassAppearance using the SetNextPass method:

```
    multiPassAppearance1->SetNextPass(multiPassAppearance2);
    multiPassAppearance2->SetNextPass(multiPassAppearance3);
```

Revision #8
Created 2023-02-21 06:19:18 UTC by Tsuyoshi.Kato
Updated 2025-01-10 11:21:14 UTC by Elisabeth Zauner