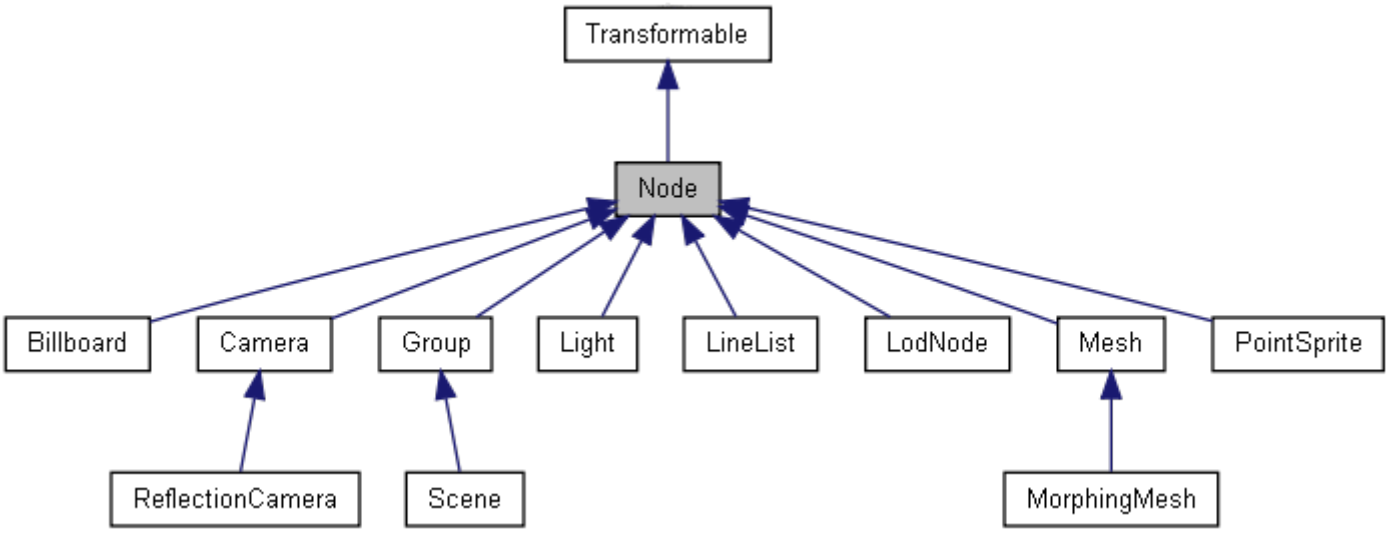
Tutorial for 3D Scene Graph and Nodes

- 3D Scenes and Nodes
- 3D Node Transformations
- 3D Node Appearance
- 3D Scene Graph Dynamics
- 3D Render Order
- Searching for Nodes with SearchTreeTraverser
- Vertex Geometry Builder
- Vertex Geometry Modifier
- Candera 3D Listeners

3D Scenes and Nodes

Scenes and Nodes

Candera::Node is an abstract base class for all scene graph nodes.



Each node defines a local coordinate system relative to the coordinate system of the parent node.

The functionality to define the local coordinate system is derived from Candera::Transformable and consists of following components:

- position,
- rotation,
- scale,
- and a generic matrix called transform matrix.

If the node is transformed from the local coordinate system to the world's coordinate system, then the node's local transformation is multiplied with all its parent's transformations. A node can also store a set of nodes as its children but a node can only have one parent at a time. Cycles are prohibited. The nodes alpha value is multiplied with the alpha value of its descendent nodes.

Candera::Scene is the top level scene node which cannot be part of any other node.

3D Node Transformations

3D Node Manipulation Example

To illustrate basic node transformation operations, the *NodeManipulationWidget_3D* and the *DisplayPositionWidget3D* part of the Tutorial Widgets can be used. An usage example for *NodeManipulationWidget_3D* widget is presented in **NodeManipulationSolution_3D** solution provided in the content folder of *cgi_studio_player*.

Please consider:

- The widget property *Node* is derived from a base class and provides the node associated to the widget (e.g. the root node for content managed by the widget).
- Only if the widget is *enabled* (this property is also derived from a base class), the setter methods of the widget properties will take effect.
- If a widget property setter changes a node property of the associated node, the visual effect can be seen, but note that the static node properties maintained by SceneComposer are not changed accordingly!
- Widget dynamics are never reflected to static scene structure. It is recommended to use SceneComposer only for property configuration and dynamics only in the Player.

Translations

Once the NodeManipulationWidget_3D is linked to a node from the static scene tree and enabled, the position coordinates can be set in the Player for this property to change the position.

```
        // Set node position
        node->SetPosition(m_position);
        // end Set node position
```

Use TranslateX property to move the node on the x axis with a delta value only.

Candera::Transformable::Translate adds the given Vector to the actual position .

```
       // Translate node on x axis
        node->Translate(Vector3(m_translateX, 0.0F, 0.0F));
        // end Translate node on x axis
```

Rotations

Similar to translation, rotation can also be applied via Candera::Transformable::SetRotation() or Candera::Transformable::Rotate().

```
        // Set node rotation
        node->SetRotation(m_rotation);
        // end Set node rotation
```

Scaling

Again there is also a Candera::Transformable::SetScale and Candera::Transformable::Scale method.

```
        // set node scale
        node->SetScale(m_scale);
        // end node scale
```

Pivot Point

It is possible to change your pivot point with the Candera::Transformable::TranslatePivotPoint (or Candera::Transformable::SetPivotPoint) method. The pivot point can be used to define e.g. the rotation point that the node should rotate around. It affects the scaling too.

How to get the screen space coordinates of a node

With the DisplayPositionWidget3D you can retrieve the screen coordinates of an associated node. Please consider that a node and a camera have to be selected in the widget. In the solution the widget is disabled by default. To see the output of the node's display position, enabling the widget and log filter (via CgiAppLog -> Info) in SceneComposer is mandatory. To get the screen space coordinates of an associated node you need the function Candera::Math3D::TransformPointToScreen, which transforms a point in world space into a point in screen space.

- First you need the world position of your node. With Node::GetWorldPosition you get the screen coordinates as output.
  ```
          Vector3 worldPosition = node->GetWorldPosition();
  ```
- With the world position of your node and with your camera you can transform the point to screen space and you retrieve the display position as a Candera::Vector2.
  ```
          Vector2 displayPos;
          static_cast<void>(Math3D::TransformPointToScreen(worldPosition, camera, displayPos)
  ```
- displayPos now contains the X and Y screen space coordinates of the selected node.

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);
```

3D Scene Graph Dynamics

Description

Usually all required nodes are already specified and preconfigured via SceneComposer, so in most cases it won't be necessary to make any adaptations within the scene graph loaded from an asset at runtime.

However, some use cases might require such adaptations. This tutorial covers the most common means to manipulate the scene graph structure dynamically.

Example Solution

- SceneGraphDynamicsSolution_3D in folder *cgi_studio_player/content/Tutorials/03_SceneGraphAndNodes*

Adding and Removing 3D Nodes

3D Scene Graph Dynamics Example

For creating and adding a new node, the following can be used:

- **SceneGraphDynamicsSolution_3D** from folder *cgi_studio_player/content/Tutorials/03_SceneGraphAndNodes*
- **SceneGraphDynamicsWidget_3D** (this is used in the example solution - refer to SceneGraphDynamicsWidget_3D)

As an example use case, a Candera::Billboard can be created, added and removed. Use the SceneGraphDynamicsWidget_3D properties described below:

- **AddBillboard**: If AddBillboard is enabled a new Billboard is created and added to the widget node. If disabled, the Billboard will be removed.

```
        // Definition of WidgetProperty "AddBillboard"
        CdaProperty(AddBillboard, bool, GetAddBillboard, SetAddBillboard)
            CdaDescription("If AddBillboard is enabled a new Billboard is created and added to t
        CdaPropertyEnd()
        // End Definition of WidgetProperty "AddBillboard"
```

Creating a new Billboard

The following code snippet generates a basic Candera::Billboard (size 30 x 30) with a Candera::Material and a matching Candera::Shader. The Shader has to be provided in the asset and is delivered through the AssetProvider by its name. Further, some translation is applied to the new Billboard.

```
        // Create and init a new Billboard

        SharedPointer<Appearance> myAppearance = Appearance::Create();
        SharedPointer<Material> myMaterial = Material::Create();
        myMaterial->SetEmissive(Color(0.5F, 0.0F, 0.0F, 1.0F));
        myAppearance->SetMaterial(myMaterial);
        myAppearance->SetShader(Base::GetAssetProvider()->GetShader(CgiAssetNames::RefTransLight
        myAppearance->SetName("Generated Appearance");

        m_billboard = Billboard::Create(30.0F, 30.0F);
        m_billboard->SetAppearance(myAppearance);
        m_billboard->Translate(Vector3(30.0F, 0.0F, 0.0F));

        // end Create and init a new Billboard
```

Adding the new Billboard to the scene graph

Next the Billboard is uploaded to VRAM and appended to the node associated with the widget.

```
        // Add Billboard

        static_cast<void>(m_billboard->Upload());
        static_cast<void>(node->AddChild(m_billboard));

        // end Add Billboard
```

Removing and destructing the Billboard

To remove the Billboard simply from the scene graph, use Candera::Node::RemoveChild. If the Billboard will be used for further operations e.g. attaching it to an other node removing would be enough. Since in this example the Billboard isn't used further it should not only be removed but also be destructed completely (freeing the resources).

The following code snippet removes the previously generated billboard from its parents, unloads the Billboard from VRAM and frees with Candera::Node::Dispose the resources.

```
        // Remove previously generated billboard and destruct it
```

```
        Node* parent = m_billboard->GetParent();
        if (parent != 0){
            static_cast<void>(parent->RemoveChild(m_billboard));
            FEATSTD_LOG_ERROR("- Node %s removed from parent %s ...\n", m_billboard->GetName(),
        }
        // Destruct node

        static_cast<void>(m_billboard->Unload());
        m_billboard->Dispose();
        m_billboard = 0;

        // end Destruct node

        // end Remove previously generated billboard and destruct it
```

As Candera::Node::Dispose internally calls the Candera::Node::RemoveChild on parent node following code would be enough for removing and destructing a node.

```
        // Destruct node

        static_cast<void>(m_billboard->Unload());
        m_billboard->Dispose();
        m_billboard = 0;

        // end Destruct node
```

> In SceneComposer the newly added Node does not appear in the Scene Tree view because it is only added dynamically by the widget.
> - The used Shader for the generated Billboard in the widget has to be included when generating an asset from the solution - ensure that "Always include in asset" flag is marked at the Shaders properties.
> - The widget must control memory and VRAM management of its internal dynamic subtree autonomously.

Widget Interaction in the Player

First, the asset must be exported and loaded in the Player. Then, one of the two existing *SceneGraphDynamicsWidget_3D* widgets must be selected for the 3D scene. After, **ChangeAppearance**, **CloneOperation** and **AddBillboard** properties can be enabled/disabled (1/0) to obtain different results.

Changing Appearance and other 3D Node Attachments

For changing the appearance of a node, the following property of the SceneGraphDynamicsWidget_3D can be used:

- **ChangeAppearance**: If ChangeAppearance is enabled, the appearance of the widgets associated node will be changed to a new generated appearance. If disabled, the previous appearance will be restored.

```
        // Definition of WidgetProperty "ChangeAppearance"
        CdaProperty(ChangeAppearance, bool, GetChangeAppearance, SetChangeAppearance)
            CdaDescription("If ChangeAppearance is enabled, the appearance will be exchanged to
        CdaPropertyEnd()
        // End Definition of WidgetProperty "ChangeAppearance"
```

Try the ChangeAppearance property at the SceneGraphDynamics_3D_Billboard widget in the example solution to change the Billboards appearance from a texture appearance to a material appearance.

Creating and changing an Appearance

The following code snippet generates a basic Candera::Appearance. A Candera::Material is created and its emissive color is set to a light green. The material and a proper shader is set to the Appearance. The shader has to be provided in the asset and is delivered through the AssetProvider by its name.

```
    // Create and set a new Appearance

    SharedPointer<Appearance> myAppearance = Appearance::Create();
    myAppearance->SetMaterial(Material::Create());
    myAppearance->GetMaterial()->SetEmissive(Color(0.0F, 0.5F, 0.0F, 1.0F));
    myAppearance->SetShader(Base::GetAssetProvider()->GetShader(CgiAssetNames::RefTransLight1_Re
    myAppearance->SetName("Generated Appearance");

    // end Create and set new Appearance
```

In the next step the node's previous Appearance is saved (for restoring purpose) and the new generated appearance is set to the node associated with the widget.

```
    // Change Appearance

    m_previousAppearance = node->GetAppearance();
    node->SetAppearance(myAppearance);
    static_cast<void>(myAppearance->Upload());

    // end Change Appearance
```

Restoring / releasing an Appearance

Restore the previous Appearance and destruct the dynamically generated appearance.

```
        // Restoring appearance and destructing

        SharedPointer<Appearance> currentAppearance = node->GetAppearance();
        static_cast<void>(currentAppearance->Unload());
        currentAppearance.Release();
        node->SetAppearance(m_previousAppearance);

        // end Restoring appearance and destructing
```

> In SceneComposer the changed Appearance does not appear in the Appearance panel because it is only changed dynamically by the widget.
> - The used shader for the generated Billboard in the widget has to be included when generating an asset from the solution - ensure that "Always include in asset" flag is marked at the Shaders properties.
> - The widget must control memory and VRAM management of dynamically created node attachments autonomously.

Cloning 3D Nodes

For cloning a node, the following property of the SceneGraphDynamicsWidget_3D can be used:

- **NodeToClone**: Select an arbitrary node in the scene graph, which shall be cloned and added to the widget node

```
        // Definition of WidgetProperty "NodeToClone"
        CdaProperty(NodeToClone, Candera::Node*, GetNodeToClone, SetNodeToClone)
            CdaDescription("Select a Candera::Node which shall be cloned and added to the widget
        CdaPropertyEnd()
        // End Definition of WidgetProperty "NodeToClone"
```

- **CloneOperation**: If enabled, the clone operation will be executed and the cloned node will be added to the widget node.

```
        // Definition of WidgetProperty "CloneOperation"
        CdaProperty(CloneOperation, bool, GetCloneOperation, SetCloneOperation)
            CdaDescription("If CloneOperation is enabled, the Candera::Node to clone will be clo
        CdaPropertyEnd()
        // End Definition of WidgetProperty "CloneOperation"
```

For the cloning operation, refer to Candera::Node::Clone. The following code snippets illustrate, what the SceneGraphDynamicsWidget_3D is actually doing for cloning a node and destructing it again.

Validating Node to Clone

This example should allow to clone nodes, which are part of the scene graph the widget is linked to. The scene graph can be traversed from a given node up to its actual top level scene node to check whether it is part of the scene graph.

```
    // Search the parent scene of a given node

    Node* currentNode = node;

    while (currentNode != 0) {
        if (currentNode->IsTypeOf(Scene::GetTypeId())) {
            return static_cast<Scene*>(currentNode);
        }
        currentNode = currentNode->GetParent();
    }
    return 0;

    // end Search the parent scene of a given node
```

Creating and Adding Clone

If the widget node and the node to clone have same top level scene, the node can be cloned and added to the widget node according to the following code snippet.

```
        // Clone the node and add it to the widgets associated node

        m_clonedNode = TreeCloner().CreateClone(*m_nodeToClone);
        static_cast<void>(node->AddChild(m_clonedNode));
        static_cast<void>(m_clonedNode->UploadAll());
        m_clonedNode->SetName("Clone");

        // end Clone the node and add it to the widgets associated node
```

> Nodes can be attached to any 3D node.

TreeCloner and Cloning Strategies

For more complex cloning see Candera::TreeCloner and Candera::TreeCloneStrategy.

Candera::TreeClonerBase is provided for cloning trees.

- The default clone strategy is to call the Clone interface on each node, and build the clone tree with the same node pattern as the original tree.
- For deeper cloning, TreeClonerBase supports attaching a Candera::TreeCloneStrategy. This TreeCloneStrategy is typically aware of different types of nodes, and delegates to other strategies.

Removing and Destructing Clone

Analogous to Removing and destructing the Billboard (removing/destructing a 3D node) the cloned node is destructed.

```
        // Remove a previously cloned node

        static_cast<void>(m_clonedNode->UnloadAll());
        m_clonedNode->Dispose();
        m_clonedNode = 0;

        // end Remove a previously cloned node
```

> In SceneComposer the newly added node clone does not appear in the Scene Tree view because it is only added dynamically by the widget.
> - It is not allowed to modify the scene graph maintained by SceneComposer outside of the subtree owned by the widget!
> - The widget must control memory and VRAM management of its internal dynamic subtree autonomously.

3D Render Order

Description

There are various reasons why sorting of objects to render is required:

- **Transparency**: Transparent objects must be rendered from back to front, while opaque objects must be rendered from front to back.
- **Performance**: Front to back order or advanced techniques like state sorting provide the best performance
- **Other**: Custom rank, Skybox, other background geometry, dialogs, pop-ups, etc.

Render Order: Involved Classes

Involved Render Order Classes

Candera::RenderOrder	Defines the sequence of nodes to render per scene and camera. Sorting is done according to following criteria: 1. Rank of render order bins: Candera::RenderOrder::SetBinRank. 2. Sorting of Nodes within bin (see below). Default bins are 'opaque' and 'transparent'. If no assignment is set, nodes are automatically assigned to opaque or transparent, according to the Node's transparency settings (RenderMode state).
Candera::RenderOrderBin	RenderOrder maintains a set of RenderOrderBins. A bin • Contains nodes that have rendering enabled • Candera::RenderOrderBin::SetOrderCriterion: Sorts nodes according to applied Candera::OrderCriterion • Candera::RenderOrderBin::SetRank: defines sequence of bin within render order • Distinction between camera dependent and camera independent bins

Candera::OrderCriterion	Nodes within a bin are sorted according to an OrderCriterion. Predefined for **camera dependent bins:** • Candera::DistanceToCameraOrderCriterion (sorting from front to back for opaque bin) • Candera::ReverseDistanceToCameraOrderCriterion (sorting from back to front for transparent bin) Predefined for **camera independent bins, sorted by explicit rank values:** • Candera::RankOrderCriterion (lower render order rank is before higher render order rank.) • Candera::ReverseRankOrderCriterion (higher render order rank is before lower render order rank) • Candera::RenderStateOrderCriterion (takes in consideration the Node's Shader, Texture and RenderMode, whose changes are considered the most expensive)
Candera::Node	A node must be assigned to a bin, and if this bin is sorted by rank, the node must also specify it's rank inside the bin. • Candera::Node::SetRenderOrderBinAssignment: Defines bin assignment. If not set, then auto assignment is processed: Opaque or transparent, according to Node's RenderMode state • Candera::Node::SetRenderOrderRank: Defines render order rank that is utilized if the bin assigned to is sorted by rank
Candera::Scene	Candera::Scene::SetRenderOrder: One RenderOrder is exactly assigned to one Scene.
Candera::Renderer	Uses RenderOrder to render Nodes according to its defined sequence.

Default Render Order: Sorting by Distance to Camera

Default Render Order

By default, all objects part of a scene are automatically sorted by Candera according to their render attributes into two predefined render order bins:

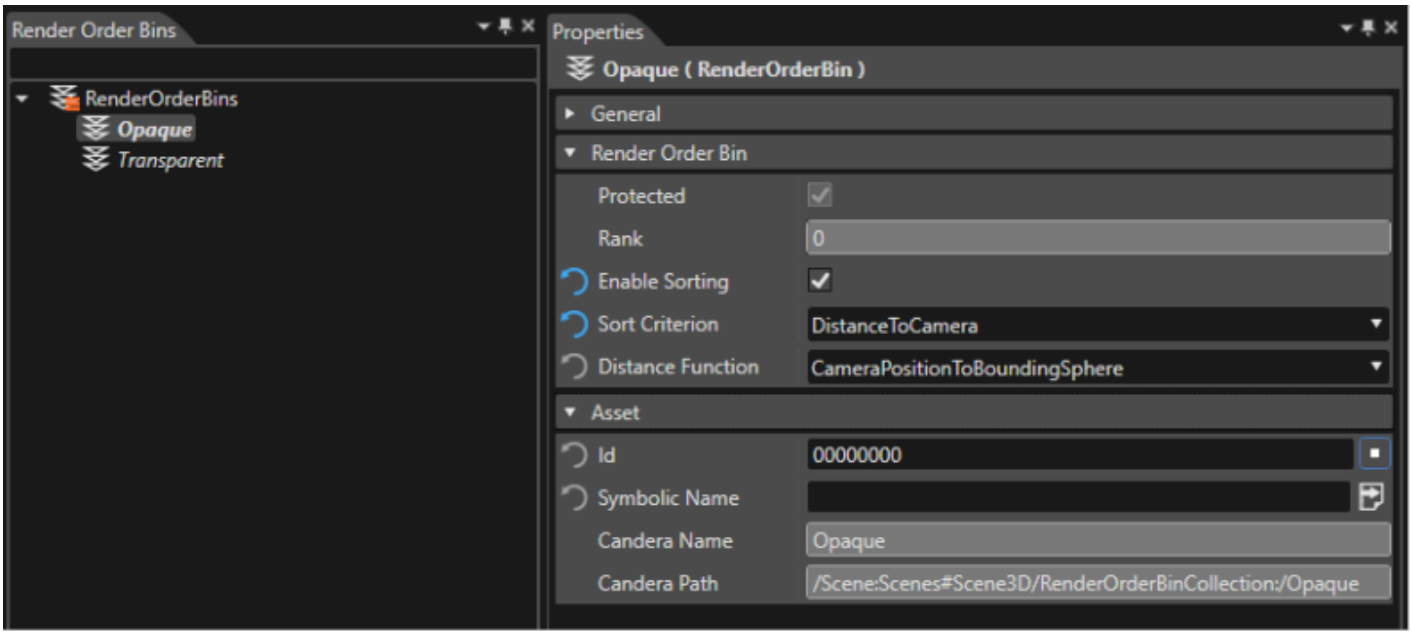
- **Opaque bin**: Opaque objects are rendered using Candera::DistanceToCameraOrderCriterion

- **Transparent bin**: Transparent objects are rendered using Candera::ReverseDistanceToCameraOrderCriterion

Both default bins are protected, which means that they are under control of Candera and cannot be deleted.

Predefined Render Order in SceneComposer

In SceneComposer, each 3D scene gets the predefined render order bins assigned by default:



Predefined Render Order in Candera

To apply the default behaviour with auto-assignment of transparent and opaque objects via Candera API, refer to following example:

```
// Create default render order with 2 default bins and max. number of 10 nodes each (opaque and
// Default rank for opaque bin is 10 and transparent bin is 20
static RenderOrder* renderOrder = RenderOrder::Create(10, 10);
// Assign render order to scene.
m_scene->SetRenderOrder(&renderOrder);
```

Fixed Render Order

To customize the default render order with a fixed render order, scene nodes must be assigned to custom render order bins sorted by explicit rank values.

- Using Candera::RankOrderCriterion, the scene will be rendered so that lower render order rank is before higher render order rank.

- When using Candera::ReverseRankOrderCriterion, higher render order rank is before lower render order rank.

Code Example: Fixed Render Order

As an example, create a render order with 3 bins and max. number of 10 nodes for O&T bin:

```
static RenderOrder renderOrder(3, 10, 10);
m_scene->SetRenderOrder(&renderOrder);
```

Create new render order bin with name "UserBin", rank "30", max. Number of 10 nodes:

```
renderOrder.CreateBin("UserBin", 30, 10);
```

Create dedicated order criterion which sorts nodes by its render order rank:

```
static RankOrderCriterion userCriterion;
```

Assign order criterion to UserBin:

```
renderOrder.SetBinOrderCriterion("UserBin", &userCriterion);
```

Assign nodes to dedicated bins:

```
m_mesh1->SetRenderOrderBinAssignment("UserBin");
m_mesh2->SetRenderOrderBinAssignment("UserBin");
```

Set dedicated render order rank for nodes:

```
m_mesh1->SetRenderOrderRank(2);
m_mesh2->SetRenderOrderRank(1);
```

Performance Optimized Render Order

Candera::RenderStateOrderCriterion implements a proof of concept OrderCriterion used for batching nodes to minimize render state changes. It takes into consideration the nodes shader, texture and render mode, whose changes are considered the most expensive.

- After sorting all the nodes using the Candera::OrderCriterionValue, rendering is done in batches of nodes with the same shader.
- Inside a batch of nodes with the same shader, the rendering will be done in batches of nodes with the same texture.

- Inside a batch of nodes with the same shader and texture, the rendering will be done in batches of nodes with the same blending enabled value.

The batching continues until the least expensive state change (CullingMode).

> - Usage of Candera::RenderStateOrderCriterion does not guarantee best batching algorithm, which should be implemented after benchmarking the target render device's state change costs.
> - RenderStateOrderCriterion groups the nodes only by their first appearance and texture. Nodes with multi-pass rendering (MultiPassAppearance), or with multiple texture units, most probably will break the batches and cancel some performance gains.
> - CANDERA_RENDER_STATE_CACHING_ENABLED should be defined to use Candera's render state caching mechanism.

Searching for Nodes with SearchTreeTraverser

Searching for Nodes

With Candera::SearchTreeTraverser it is possible to find nodes within a given scene tree by certain criteria. The search scene graph traverser is a Candera::TreeTraverserBase implementation that evaluates each node with a given Candera::SearchCriterion.

Following code snippets illustrate how to use the traverser.

Creating a plugin that registers a configuration and uses the default configuration editor.

First create a Candera::SearchTreeTraverser and then set its Candera::SearchCriterion. The search criterion validates nodes based on the implemented criterion.

In the example below the criterion is a Candera::ScopeMaskSearchCriterion.

```
    // Create criterion
    ScopeMaskSearchCriterion<Node> scopeMaskCriterion;
    scopeMaskCriterion.SetScopeMask(scopeA);

    // Create SearchTreeTraverser
    SearchTreeTraverser<Node> scopeSearchTraverser;
    scopeSearchTraverser.SetSearchCriterion(&scopeMaskCriterion);
```

Tree Traversing

To start a search within a node tree, call Candera::SearchTreeTraverser::Traverse(). With the Candera::SearchTreeTraverser::SetOnFoundAction you can define if the search should stop after the first node that fulfills the search criterion is found, or if it should continue to gather all valid nodes.

Possible TraverserActions are:

- *ProceedTraversing*: Find all nodes matching the given search criterion (traversing is stopped at the end of the tree)
- *StopTraversingForDescendants*: Find only nodes on the top level of the tree (traversing stops at child nodes and proceeds at siblings, until the end of the tree)

- *StopTraversing*: Find only the first node matching the search criterion (traversing is stopped immediately).

The nodes fulfilling the search criterion are retained and are available to be later retrieved. Get the nodes one by one with Candera::SearchTreeTraverser::GetSearchResult. The order of the nodes is not defined and each node can be retrieved only once. When no more nodes are available, the method returns 0.

Following are examples for the different TraverserActions.

ProceedTraversing

This action is used if the traversing should continue with the next child (if the current node has any) or with the next sibling.

First define the action and then call the Traverse() method. All nodes that fulfill the search criterion are found and stored internally in a list.

```
    // Set on found action and traverse
    scopeSearchTraverser.SetOnFoundAction(Candera::TreeTraverserBase<Node>::ProceedTraversing);
    scopeSearchTraverser.Traverse(*groupRoot);

    // Get results
    UInt32 nodeCount = 0;
    for (Node* node = scopeSearchTraverser.GetSearchResult(); node != 0; node = scopeSearchTrave
        nodeCount++;
    }
```

StopTraversingForDescendants

Used if the traversing should stop for all child nodes and continue with the next sibling.

```
    // Set on found action and traverse
    scopeSearchTraverser.SetOnFoundAction(
Candera::TreeTraverserBase<Node>::StopTraversingForDescendants);
    scopeSearchTraverser.Traverse(*groupRoot);
```

StopTraversing

If you search for only one node which fulfills your criteria, you can use StopTraversing. The traversing will stop immediately after the first node is found.

```
    // Set on found action and traverse
    scopeSearchTraverser.SetOnFoundAction(Candera::TreeTraverserBase<Node>::StopTraversing);
    scopeSearchTraverser.Traverse(*groupRoot);
    Node* resultNode = scopeSearchTraverser.GetSearchResult();
```

> Another call of Candera::SearchTreeTraverser::GetSearchResult would return 0, because we use StopTraversing here. This leads to one result only.

Vertex Geometry Builder

Description

The Candera::VertexGeometryBuilder class helps effectively build Candera::VertexGeometry objects.

Procedural Geometry

Example: Wireframe Cube with LineList

First example shows how to generate a wireframe cube using a Candera::LineList object.

The vertex geometry will be generated procedurally using a Candera::VertexGeometryBuilder object (e.g. *m_builder*).

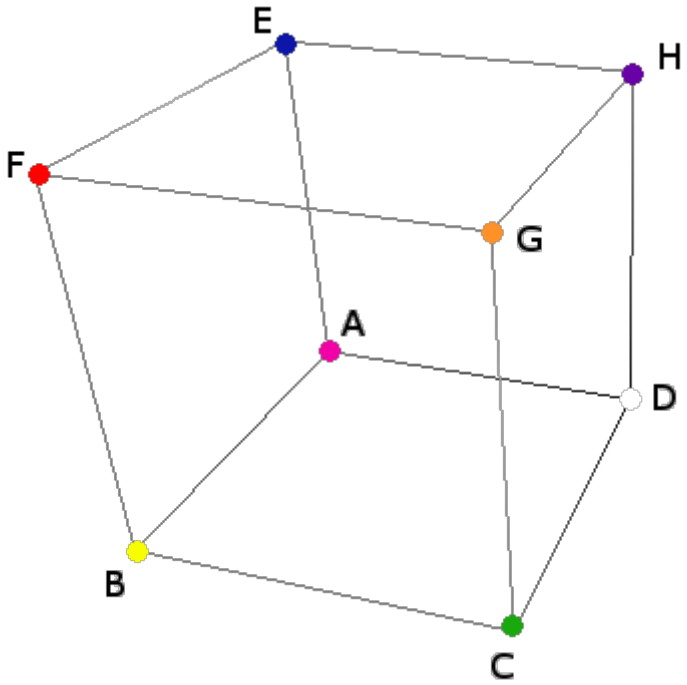
Vertex Element Format: Position and Color

At the beginning we have to specify what kind of information will be set in the vertices. Of course we need information about the position and, optionally, for an improved visual effect, we could also add color information.

```
    // Set the data type to be used: position & color
    m_builder.SetVertexElementFormat(VertexGeometry::Position, 0, VertexGeometry::Float32_3);
    m_builder.SetVertexElementFormat(VertexGeometry::Color, 0, VertexGeometry::Float32_4);
```

Vertex Data: Position

The position information for the vertices is obtained from the array *c_cubeData* which, in fact, stores the coordinates of the points A and G. The coordinates for the rest of the vertices are obtained relatively to this points (e.g. F(X_A, Y_G, Z_G) or D(X_G, Y_A, Z_A)




```
// Coordinates for the points A & G
static const Float c_cubeData[2][3] = {
    {-1.0F, -1.0F, -1.0F },
    {1.0F, 1.0F, 1.0F }
};
```

VertexGeometryBuilder: Set Vertex Elements

Because the LineType property of the LineList object will be set to LineType::Lines, the vertex buffer will have to contain a pair of vertices for each line of the cube: 2 vertices/line * 12 lines = 24 vertices.

```
    // Set the position & color information for each vertex
    // Line segment AD
    m_builder.SetVertexElement(VertexGeometry::Position, 0, c_cubeData[0][0], c_cubeData[0][1],
    m_builder.SetVertexElement(VertexGeometry::Color, 0, 1.F, 0.F, 1.F, 1.F); // magenta
    // Increment the vertex cursor
    m_builder.IncrementVertexCursor();
    m_builder.SetVertexElement(VertexGeometry::Position, 0, c_cubeData[1][0], c_cubeData[0][1],
    m_builder.SetVertexElement(VertexGeometry::Color, 0, 1.F, 1.F, 1.F, 1.F); // white
    m_builder.IncrementVertexCursor();
    // Line segment AB
    m_builder.SetVertexElement(VertexGeometry::Position, 0, c_cubeData[0][0], c_cubeData[0][1],
    m_builder.SetVertexElement(VertexGeometry::Color, 0, 1.F, 0.F, 1.F, 1.F); // magenta
    m_builder.IncrementVertexCursor();
    m_builder.SetVertexElement(VertexGeometry::Position, 0, c_cubeData[0][0], c_cubeData[0][1],
    m_builder.SetVertexElement(VertexGeometry::Color, 0, 1.F, 1.F, 0.F, 1.F); // yellow
    // ...
```

Create Vertex Buffer

Create a Candera::VertexBuffer object and attach to it the vertex geometry obtained from the Candera::VertexGeometryBuilder:

```
    VertexGeometry* vertexGeom = m_builder.GetVertexGeometry();
    SharedPointer<VertexBuffer> vertexBuffer = VertexBuffer::Create();
    static_cast<void>(vertexBuffer->SetVertexGeometry(vertexGeom, VertexBuffer::VertexGeometryDi
    vertexBuffer->SetPrimitiveType(VertexBuffer::Lines);
```

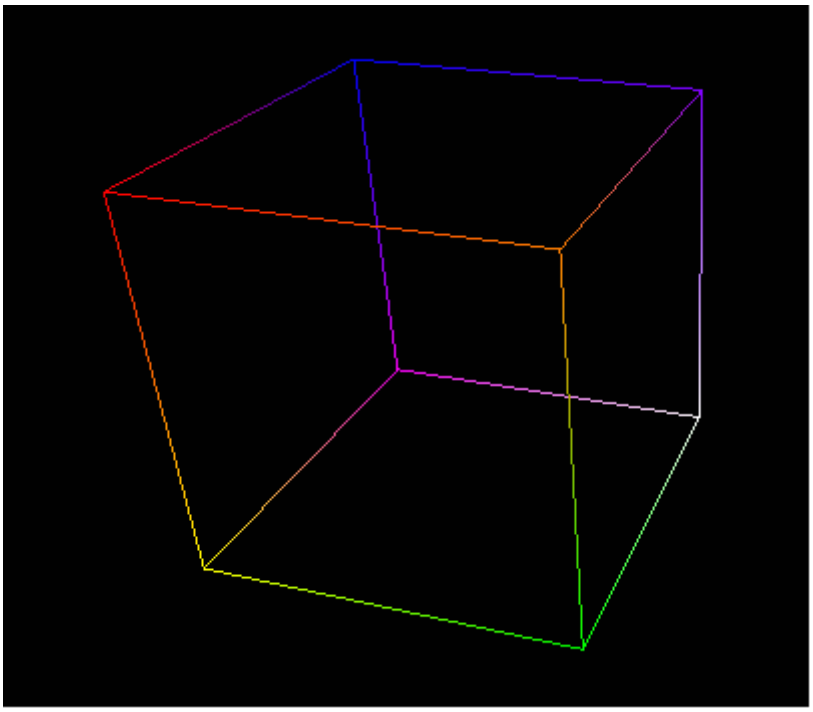
Create LineList using the Vertex Buffer

Attach the vertex buffer to LineList object:

```
    // Create and configure the line list object
    m_lineList = LineList::Create();
    m_lineList->SetAppearance(Appearance::Create());
    m_lineList->GetAppearance()->SetMaterial(Material::Create());
    m_lineList->GetAppearance()->GetMaterial()->SetEmissive(Color(1.0F, 1.0F, 0.0F));
    m_lineList->GetAppearance()->SetShader(m_shaderLightMaterial);
    m_lineList->SetLineType(LineList::Lines);
    m_lineList->SetWidth(1.0);
    m_lineList->SetIntersectionTestEnabled(false);
    m_lineList->SetVertexBuffer(vertexBuffer);
    m_lineList->SetScopeMask(ScopeMask(false));
    static_cast<void>(m_lineList->SetScopeEnabled(1, true));
    m_lineList->SetRenderingEnabled(true);
    m_lineList->SetName("Wireframe");
    static_cast<void>(m_lineList->Upload());
```

Wireframe Cube Result

The image below shows the resulting wireframe cube:



> For an easier manipulation of the widget in SceneComposer all the shaders are hardcoded in the source file and then created by the widget at the initialization time.

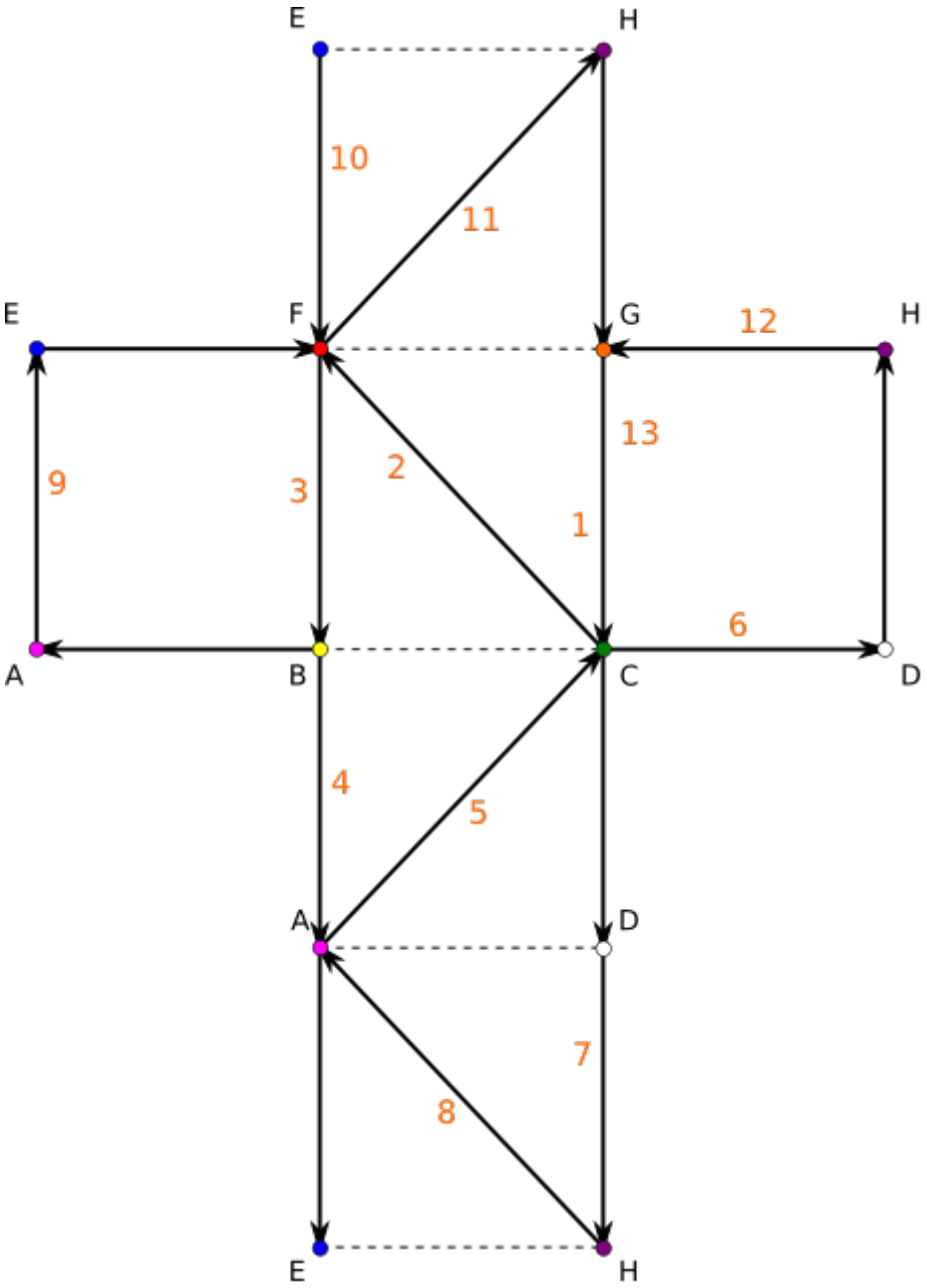
> Alternatively obtain shaders from Candera::AssetProvider (e.g. GetAssetProvider()->GetShader("...")) in case the application uses an asset library.

Indexed Geometry

Example: Solid Cube from one Triangle Strip

The next example will show how to create a solid cube from one triangle strip using a Candera::Mesh object. In this case the cube is determined by only 14 vertices which have to be ordered in a special way.

The figure below presents a cube unfolded and a possible solution of ordering the vertices:



Indexed Vertex Sequence

The vertex sequence is stored in the *c_index_order* array, encoding the order:

- **G->C->F->B->A->C->D->H->A->E->F->H->G->C**.

Each array element represents an offset it the vertex buffer.

```
// The vertex sequence - triangle strip
static const UInt16 c_index_order[] = { 12, 11, 7, 3, 0, 11, 1, 17, 0, 5, 7, 17, 12, 11 };
```

The vertex geometry builder object used in the previous example has already 24 vertices set with position and color information. Because only 14 vertices out of 24 are needed in this case and, also, because their sequence has to be different, an index buffer is added:

```
    // Change the order of the vertices using indexes
    for (Int index = 0; index < 14; index++) {
        // Define the value of the index
        m_builder.SetIndexElement(c_index_order[index]);
        // Increment the index cursor
        m_builder.IncrementIndexCursor();
    }
```

Create TriangleStrip Vertex Buffer

Create the vertex buffer using the geometry from the builder:

```
    VertexGeometry* vertexGeom = m_builder.GetVertexGeometry();
    SharedPointer<VertexBuffer> vertexBuffer = VertexBuffer::Create();
    static_cast<void>(vertexBuffer->SetVertexGeometry(vertexGeom, VertexBuffer::VertexGeometryDi
    vertexBuffer->SetPrimitiveType(VertexBuffer::TriangleStrip);
```

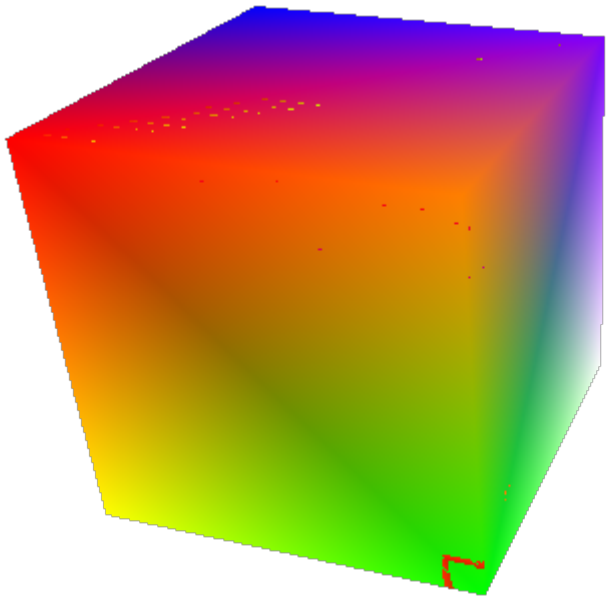
Create Mesh using the Vertex Buffer

Configure mesh and attach the vertex buffer:

```
    m_mesh = Mesh::Create();
    m_mesh->SetVertexBuffer(vertexBuffer);
    m_mesh->SetAppearance(appearance);
    m_mesh->SetRenderingEnabled(false);
    m_mesh->SetName("SolidNonTextured");
    static_cast<void>(m_mesh->Upload());
```

Solid Cube Result

The image below shows the resulting solid cube:



Remove & Add Vertex Elements

Example: Textured Cube

In this example the vertex geometry builder object will be used to generate the vertex geometry for a textured cube.

Adding texture to a mesh requires that its vertices should contain specific information like *normal* and *texture coordinate*. Also, the *color* information is no longer needed so it can be removed:

```
    // Vertex Elements Manipulation
    // Remove color data associated to the vertices
    m_builder.RemoveVertexElement(VertexGeometry::Color,0);
    // Add data type to be used: texture coordinate & normal
    m_builder.SetVertexElementFormat(VertexGeometry::TextureCoordinate, 0, VertexGeometry::Float
    m_builder.SetVertexElementFormat(VertexGeometry::Normal, 0, VertexGeometry::Float32_3);
```

Vertex Data: Normals and Texture Coordinates

After adding the new data types for the vertices it is time to add the corresponding information. First, add normals for vertices:

```
    // Move the cursor to the first vertex
    m_builder.SetVertexCursor();
    for (UInt32 i = 0; i < m_builder.GetVertexCount(); i++ )
    {
        m_builder.SetVertexElement(VertexGeometry::Normal, 0, 0.0F, 0.0F, 1.0F);
        m_builder.IncrementVertexCursor();
    }
```

In order to map the UV texture coordinate values correctly, the cube faces should be drawn using 4 vertices per face, so, in this case, all 24 vertices will be used. Of course, the vertex order has to be adjusted.

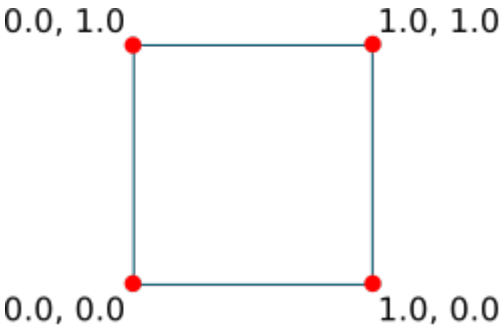
The new vertex order is stored in the *c_indexDataTexturedCube* array.

```
// Vertex order - textured cube
static const UInt16 c_indexDataTexturedCube[24] = {
    1,17,0,5,
    2,6,3,7,
    9,8,11,12,
    13,14,19,21,
    15,23,16,22,
    4,10,20,18
};
```

Now, reorder the vertices:

```
    // Overwrite the index buffer
    // Move the cursor to the first index
    m_builder.SetIndexCursor();
    for (UInt32 i = 0; i < m_builder.GetVertexCount(); i++ )
    {
        // Define the value of the index
        m_builder.SetIndexElement(c_indexDataTexturedCube[i]);
        m_builder.IncrementIndexCursor();
    }
```

The vertices of each face of the cube should be set with texture coordinate information as shown below:



Set the texture coordinates for vertices:

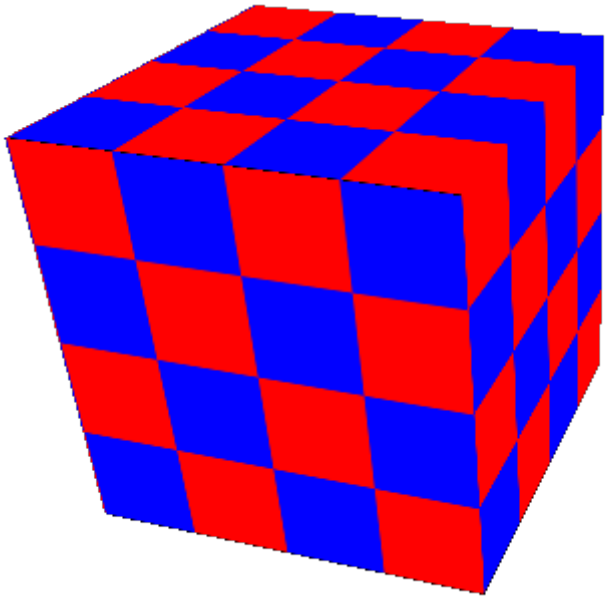
```
    Int idx = 0;
    for (Int nb_of_faces = 0; nb_of_faces<6; nb_of_faces++) {
        for (Int u=0; u<=1; u++) {
            for (Int v=0; v<=1; v++) {
                //Set the position of the vertex cursor
                m_builder.SetVertexCursor(c_indexDataTexturedCube[idx]);
                m_builder.SetVertexElement(VertexGeometry::TextureCoordinate, 0, static_cast<Flo
                idx++;
            }
        }
    }
```

The vertex buffer is created and configured in the same way as in the previous example.

The mesh setup is also similar but, unlike the solid cube, the textured cube appearance has to be set with a texture and a texture shader.

Textured Cube Result

Here is the final result:



Range Data Usage

Example: Solid Cube using Range Data

This example will show how to create a solid cube as in the second example but using range data.

First, all existing data from the vertex geometry builder is removed.

```
    // Clear all data stored by the builder
    m_builder.Clear();
```

The vertex data is taken from the two buffers *c_positionData* and *c_colorData*.

```
static const Float c_positionData[8][3] = {
    { -1.0F, 1.0F, 1.0F},
    { 1.0F, 1.0F, 1.0F},
    { -1.0F, -1.0F, 1.0F},
    { 1.0F, -1.0F, 1.0F},
    { -1.0F, 1.0F, -1.0F},
    { 1.0F, 1.0F, -1.0F},
    { -1.0F, -1.0F, -1.0F},
    { 1.0F, -1.0F, -1.0F}
};

static const Float c_colorData[8][4] = {
    { 0.0F, 0.0F, 1.0F, 1.0F},
    { 0.0F, 1.0F, 0.0F, 1.0F},
    { 0.0F, 1.0F, 1.0F, 1.0F},
```

```
    { 1.0F, 0.0F, 0.0F, 1.0F},
    { 1.0F, 0.0F, 1.0F, 1.0F},
    { 1.0F, 1.0F, 0.0F, 1.0F},
    { 1.0F, 1.0F, 1.0F, 1.0F},
    { 1.0F, 0.5F, 1.0F, 1.0F}
};
```

The vertex sequence is taken from the array *c_indexData*.

```
// Range data index
static const UInt16 c_indexData[] = {3, 2, 1, 0, 4, 2, 6, 3, 7, 1, 5, 4, 7, 6 };
```

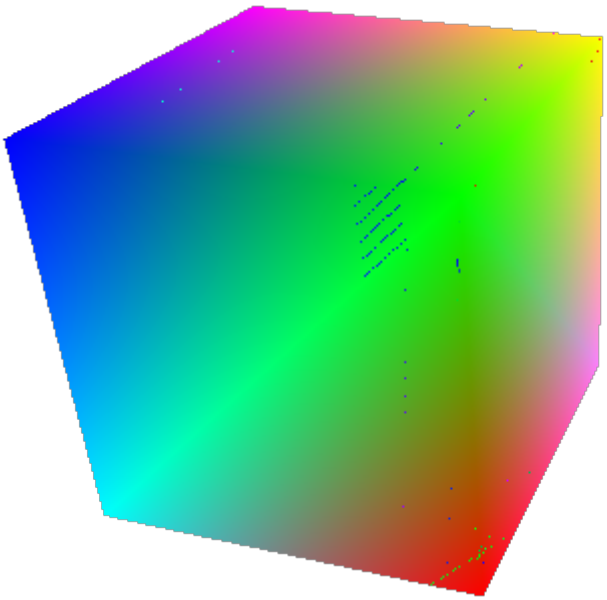
Use the methods Candera::VertexGeometryBuilder::SetVertexElementRange and Candera::VertexGeometryBuilder::SetIndexElementRange to add the required information.

```
    // Add position, color and an index from range data
    // Splice values to the position buffer starting with the first vertex
    static_cast<void>(m_builder.SetVertexElementRange(VertexGeometry::Position, 0, 0, VertexGeom
        8, 3*sizeof(Float), c_positionData ));
    // Splice values to the color buffer starting with the first vertex
    static_cast<void>(m_builder.SetVertexElementRange(VertexGeometry::Color
, 0, 0, VertexGeometry::Float32_4,
        8, 4*sizeof(Float), c_colorData ));

    // Splice values to the index buffer starting with the first index
    static_cast<void>(m_builder.SetIndexElementRange(0, sizeof(c_indexData)/sizeof(UInt16),c_inc
```

The vertex buffer configuration and the mesh setup are identical with the ones from the second example.

Here is the final result:



Concatenate Geometries

VertexGeometryBuilder::SpliceGeometry

In some cases it might be useful to concatenate two vertex geometries. This can be done by using the Candera::VertexGeometryBuilder::SpliceGeometry method as shown below:

```
    m_builder.SpliceGeometry(0, 0, vertexGeometry1);
    m_builder.SpliceGeometry(vertexGeometry1->GetVertexCount(), vertexGeometry1->GetIndexCount()
    vertexGeometry1cat2 = builder.GetGeometry();
```

Vertex Geometry Modifier

Overview

The VertexGeometryModifier, in conjunction with a VertexAccessor, helps to modify

VertexGeometry objects by the means of the setters offered by the class interface. The interface offers also getters which are helpful to retrieve the values of the vertex elements from a given vertex buffer.

Example

The example below shows how to change the position values for the vertices from a vertex buffer of a given mesh.

```
VertexGeometry* vertexGeom;
bool isMutable = m_mesh3->GetVertexBuffer()->GetVertexGeometry()->GetVertexArrayResc
// Get the vertex geometry of the mesh
if (isMutable) {
    vertexGeom = m_mesh3->GetVertexBuffer()->GetVertexGeometry();
}
else {
    DiagnosticPlatform::ConsoleOut("VertexGeometryData not mutable");
    break;
}
 // The vertex accessor will be used by the vertex geometry modifier to access the v
VertexAccessor accessor(*vertexGeom);

// Create a vertex geometry modifier specifying as usage the type of vertex attribut
// which will be changed
VertexGeometryModifier modifier(*vertexGeom, VertexGeometry::Position, 0);
Float vertexData[3];
for(Int i = 0; i<8; i++) {
    if(s_inflate) {
    // Retrieve the position data of the vertex "i" in the vertexData array
    modifier.GetVertexElement(accessor.GetVertex(i), vertexData, 3);

    // Alter the data and set it back
    modifier.SetVertexElement(accessor.GetMutableVertex(i), vertexData[0]*1.3F, vert
    }
    else {
        // Set the position of the vertex with data get from an array (Float c_posit
        modifier.SetVertexElement(accessor.GetMutableVertex(i), c_positionData[i], 3
    }
}

// Update the vertex buffer on the mesh
m_mesh3->GetVertexBuffer()->Update(0, m_mesh3->GetVertexBuffer()->GetVertexGeometry(
```

Candera 3D Listeners

Applications can receive notifications on scene graph events. Therefore implement and use a listener, which defines hooks to receive those notifications.

Following listener interfaces are provided in Candera 3D:

- Candera::RendererListener defines hooks that are called before or after a node is rendered. It supports events for OnNodePreRender and OnNodePostRender, so you are informed before and after a node is rendered.
- Candera::NodeListener defines hooks that are called when certain node functions are triggered, e.g. when a node's transformation changes.
- Candera::SceneListener defines hooks that are called on certain scene actions. The listener informs when a scene is activated.
- Candera::AnimationPlayerListener defines several hooks for the AnimationPlayer. It informs about changes of the animation, like when an animation has started, stopped, finished, resumed, paused and when the direction has changed.
- Candera::CameraListener defines hooks that are called before or after a camera is rendered. It supports events for OnPreRender, OnPostRender, OnProjectionChanged and OnViewChanged.
- Candera::ProjectionListener defines hooks that are called when projection parameters are changed. It supports OnProjectionChanged, which means the camera reacts to changes of the projection matrix associated and updates frustums accordingly.

In order to register a listener simply derive from the listener class and override pure virtual functions with custom code.

Example Listener Implementation

Refer to Using Animation Callback Functions for an example how to implement and use an Animation listener.