Shader Configuration

> The Solution option allows for Default Shader Configurations.

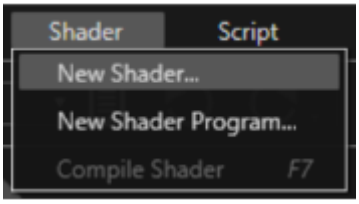
Add a New Vertex or Fragment Shader

Shaders are software programs that include instructions on how the graphics processing unit (GPU) should render certain images on the screen. A shader program consists of two types of shaders:

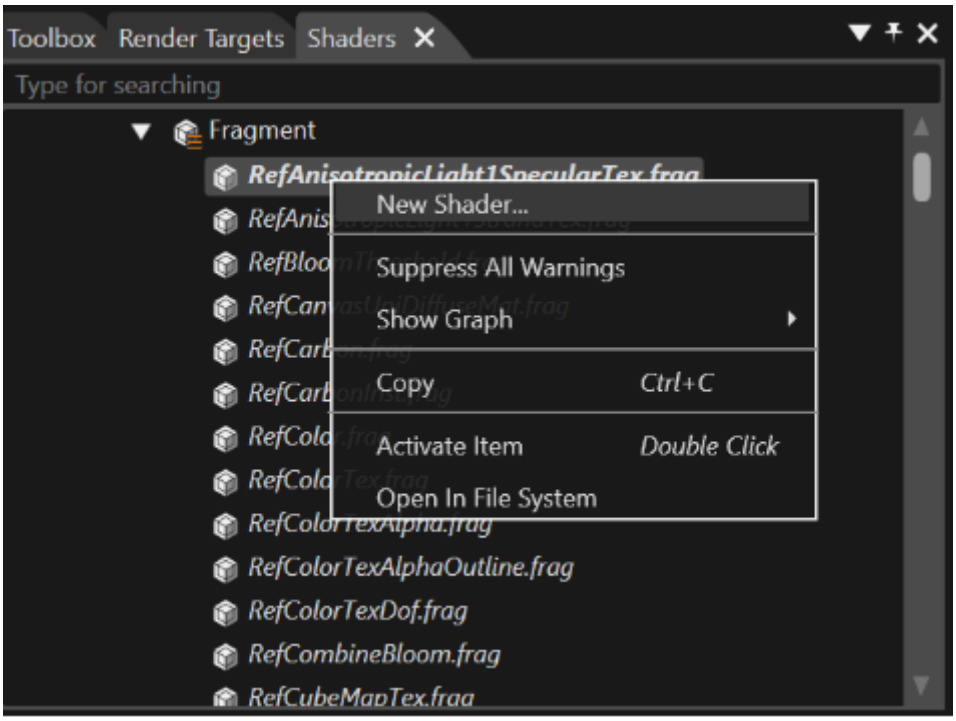
- Vertex Shaders
- Fragment (or Pixel) Shaders

There are following options to create a new (empty) shader, which can later be combined into appropriate shader pairs or programs.

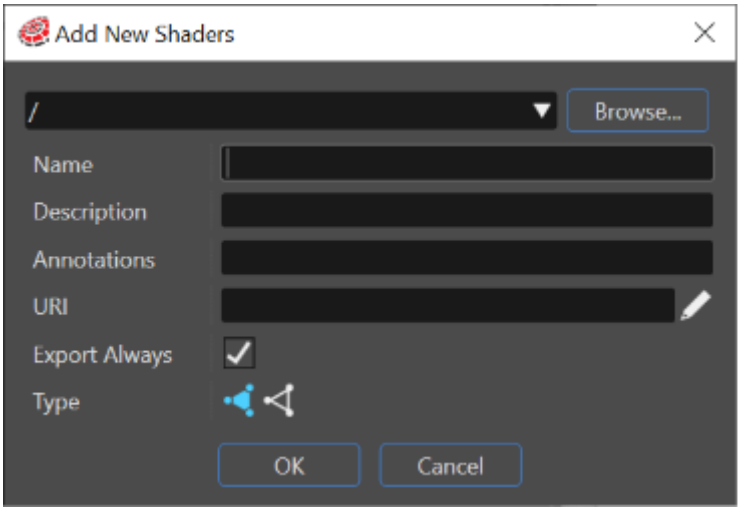
Use the "Shader" > "New shader" menu option



In the Shaders panel, use the context menu option "New Shader" of any node within the "Shaders" category:



In the "Add New Shader" dialog, specify the shader location, shader name, and shader type.



After creating the shaders, the shader instructions need to be added in the Shader Editor.

Import Vertex and Fragment Shaders

It is also possible to import already existing shaders.

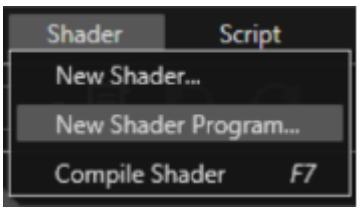
For information about how to import a Vertex or Fragment shader please refer to section How to Import Resources.

Add a New Shader Program

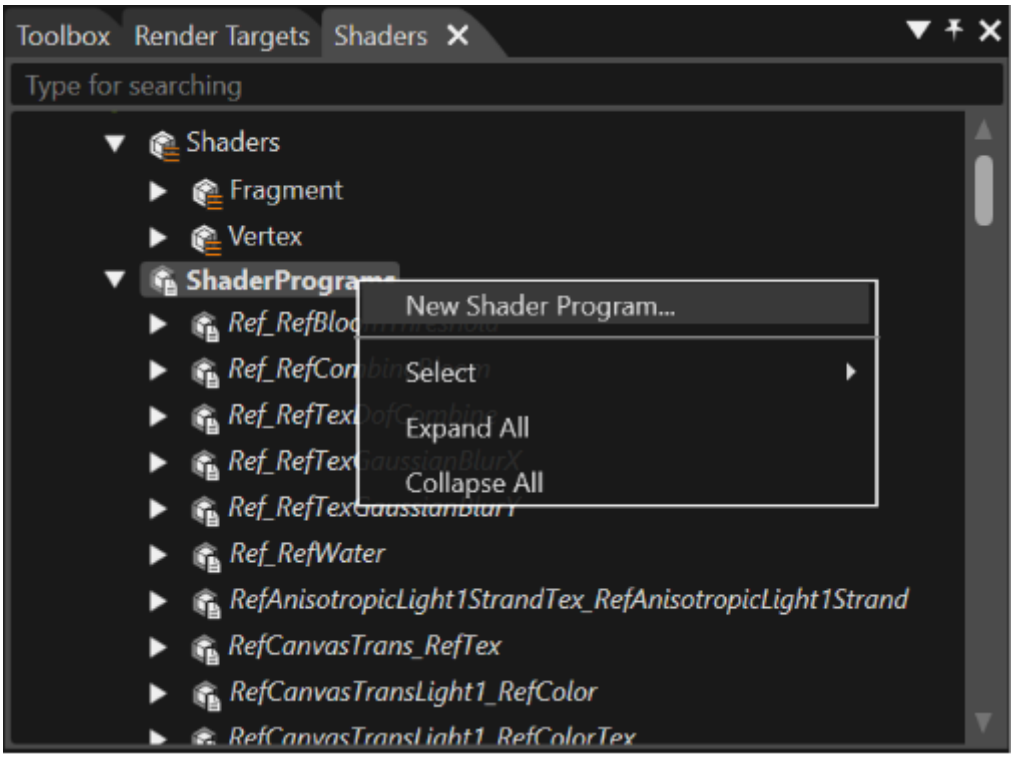
Fragment and vertex shaders are combined into shader pairs or programs which can then be applied to node appearances.

Create a new shader program:

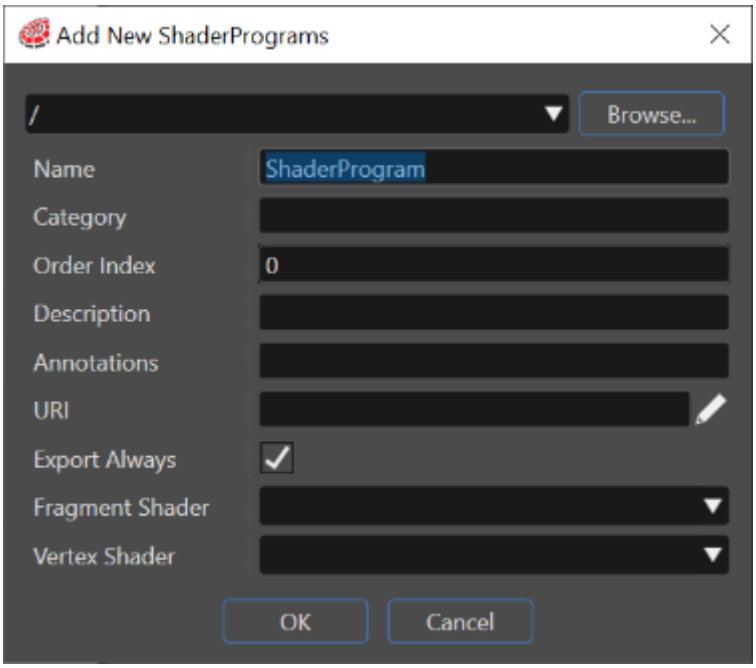
Use the "Shader" > "New Shader Program" menu option.



Use the context menu option "New Shader Program" of any node within the shader program category from the Shaders panel.



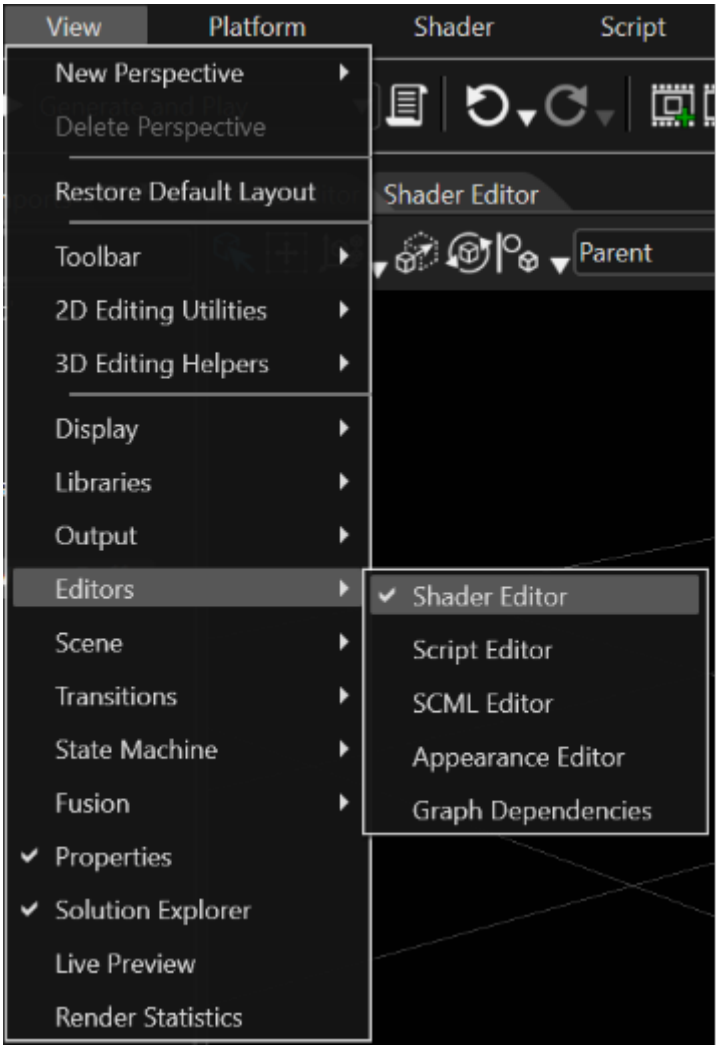
Specify location and name and the desired shader pair consisting of one fragment shader and one vertex shader in the dialog "Add New Shader Program".



Tick the "Export Always" checkbox to always include the shader program to the generated assets.

Shader Editor

Edit shaders in the "Shader Editor". Use the menu option "View" > "Editors" > "Shader Editor" to make the panel visible.



Open shaders in Shader Editor for editing:

- Select a vertex or fragment shader in the Shaders panel
- Select a shader program in the Shaders panel

The selected shader(s) are displayed in the Shader Editor panel (in both cases) like this:

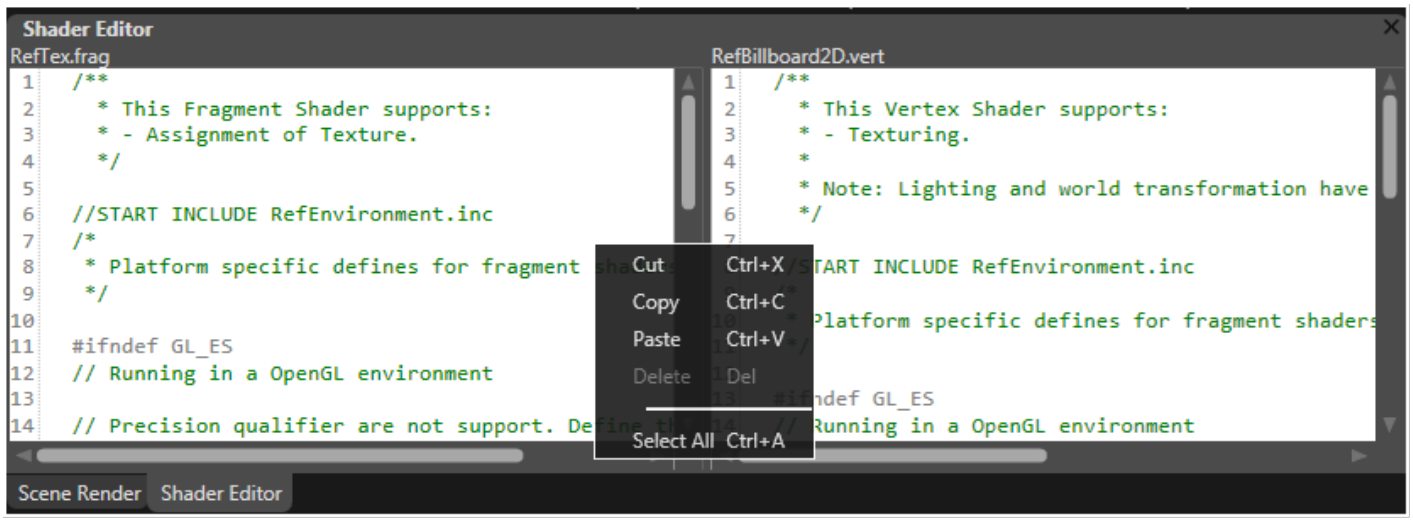
- The vertex shader in the left side and
- The fragment shader in the right side.

Shader requirements - including the number of lights, materials and textures - are displayed in a tab in the Property Grid when a shader program is selected. The tab also contains the available attributes and uniforms.

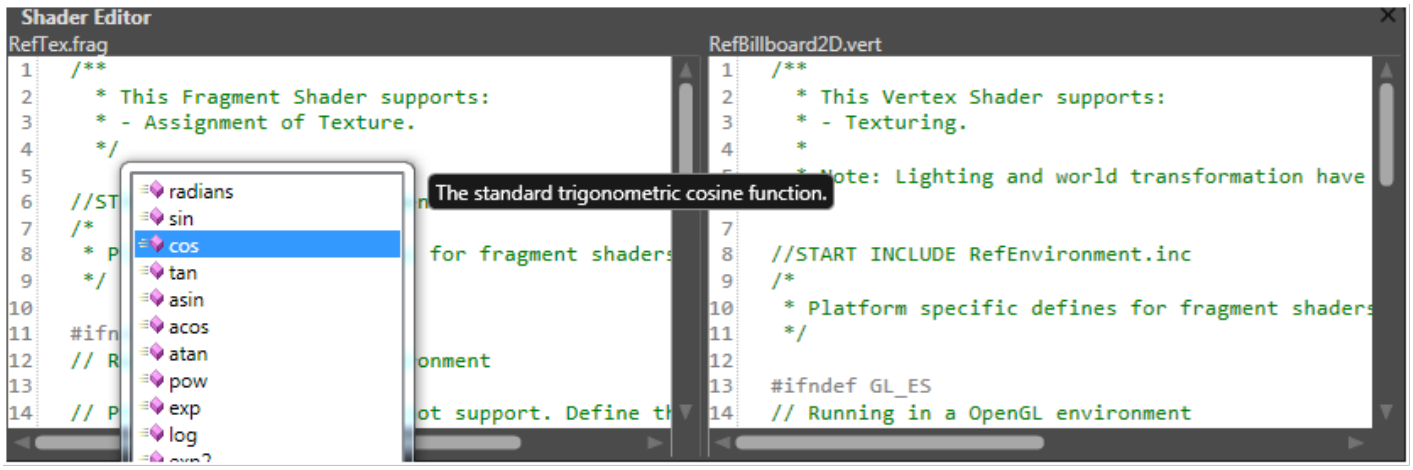
> On OpenGL ES 3.0 targets, up to **16** texture units are supported (CANDERA_MAX_TEXTURE_UNIT_COUNT = 16). On OpenGL ES 2.0, **8** texture units are guaranteed unless ES extensions are enabled.

Shader Editor Features

The context menu of the Shader Editor provides "Cut", "Copy", "Paste", "Delete", "Select All" functions for editing shader instructions:



On pressing [Ctrl+ Space bar], a list with helper functions is available for shader programming:



Compile the edited shader using the platform's shader compiler

- By using menu item "Shader" > "Compile Shader" or
- by pressing F7 key.

The compilation result is reported in the "Shader Compiler Errors" panel. This can be opened via menu item "View" > "Shader Compiler Errors".

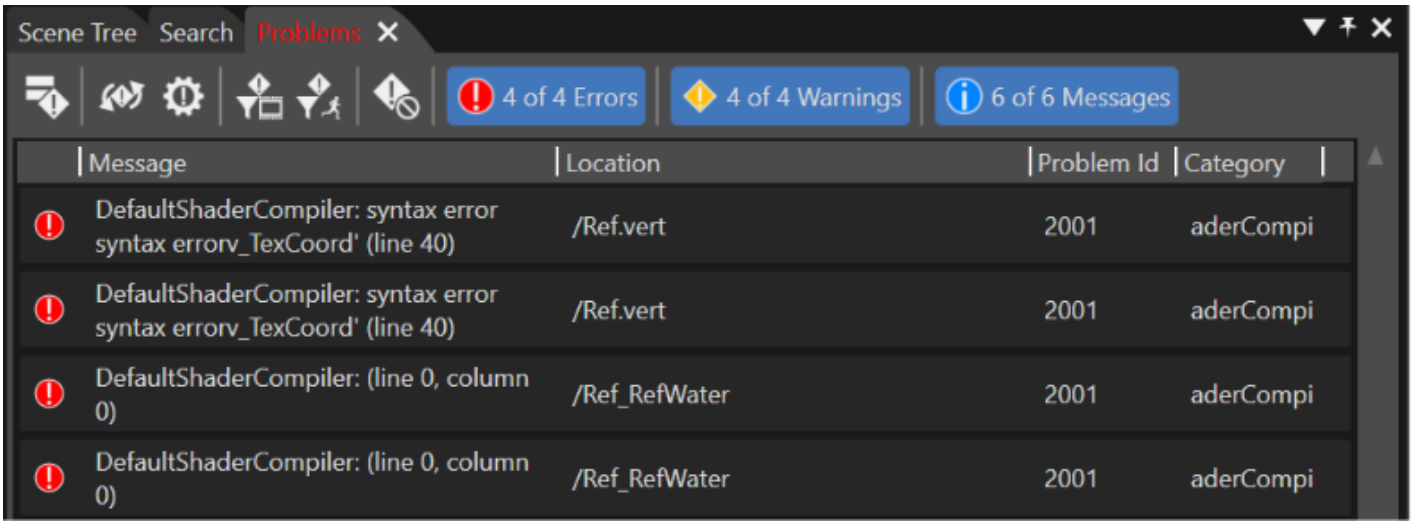
For further information please refer to section Compile Shaders.

Compile Shaders

Shader compilation is triggered in the following situations:

1. During shader editing, if requested via menu item "Shader" > "Compile Shader" or by pressing F7 key, the selected shader or shader program will be compiled using the platform's shader compiler.
2. During scene editing, when the scene dependencies are generated, the shaders will be compiled using an internal shader compiler (so called "builtin://DefaultShaderCompiler").
3. During asset generation, depending on the selected shader compiler type, the shaders will be compiled using platform's shader compiler (Target) or internal shader compiler (Simulation).

The compilation result can be seen in the "Problems Browser" panel. For further information please refer to section Problems Browser.

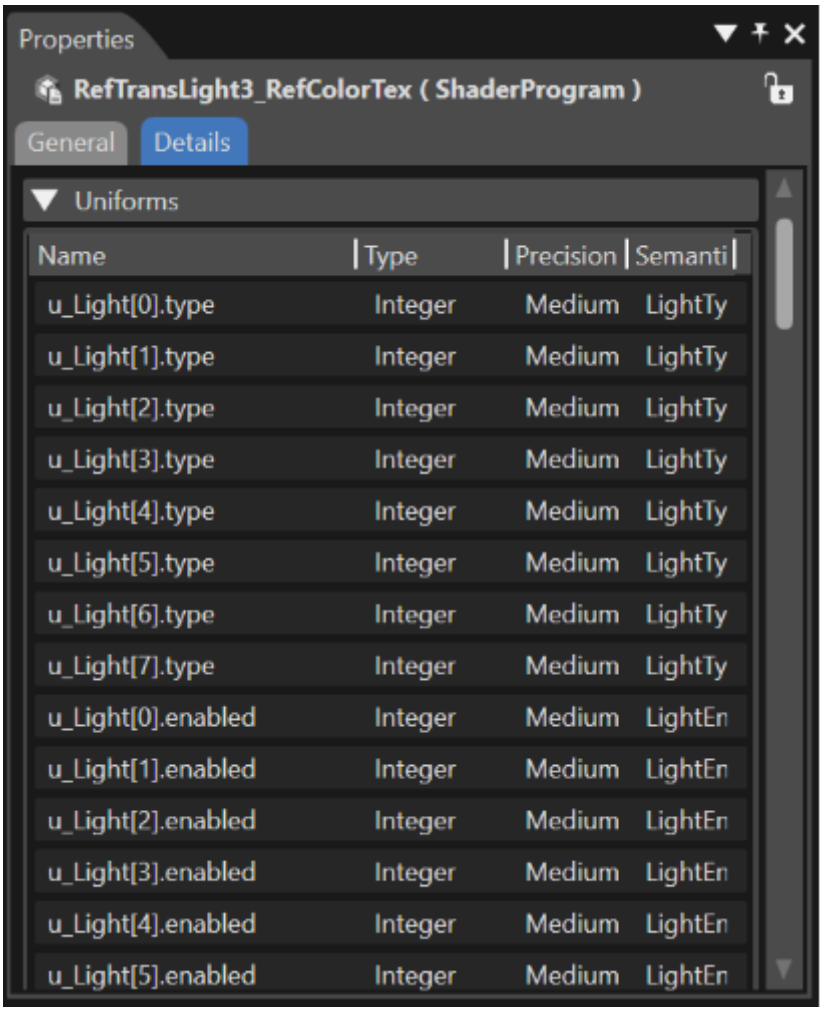


After successful shader compilation, a regarding message is displayed.

Shader Parameters

SceneComposer offers the necessary support to define shader parameters for uniform and attribute values. For every shader, Candera supplies a list of uniforms and their corresponding types. Uniforms are set via UniformSetters.

- A new category named "Shader Parameters" is loaded for every node, after choosing a Shader Program from Properties panel.
- The available uniforms are displayed and desired values can be set for them after choosing a proper Uniform Setter from the "Uniform Setter" property list.



When a shader is edited in the Shader Editor panel and new uniforms are created, those uniforms will only be listed in the Shader Parameters section of the node properties after successful shader compilation.

If the shader program is faulty (e.g. compile error), the previously retrieved uniforms are presented in read-only mode.

Instanceable Shaders

Instanceable shaders are integrated into SceneComposer. They have "Inst" in their names and are special shaders used for instanced draw.

When using an instance-able shader, uniforms that are declared as arrays in the shader, will not be exposed as an array in the SceneComposer view. Instead it has to be exposed as a single value.

When an instanceable shader is selected, the property grid will display the Max Instance Count value (represents the maximum number of element that the shader can render in a batch render).

Instanceable shader validations. In an instanceable shader, the following shall hold true:

- All uniforms (except certain auto-uniforms) are defined as arrays - an error will be emitted otherwise
- The sizes of the array uniforms must be the same - an error will be emitted otherwise

List of auto-uniforms that must not be declared as arrays in instanceable shaders:

- "u_PMatrix", // ProjectionMatrix4
- "u_VPMatrix", // ViewProjectionMatrix4
- "u_CamPosition", // CameraPosition
- "u_CamDirection", // CameraLookAtVector

- "u_Texture", // Texture
- "u_Texture1", // Texture1
- "u_Texture2", // Texture2
- "u_Texture3", // Texture3
- "u_Texture4", // Texture4
- "u_Texture5", // Texture5
- "u_Texture6", // Texture6
- "u_Texture7", // Texture7
- "u_Texture8", // Texture8
- "u_Texture9", // Texture9
- "u_Texture10", // Texture10
- "u_Texture11", // Texture11
- "u_Texture12", // Texture12
- "u_Texture13", // Texture13
- "u_Texture14", // Texture14
- "u_Texture15", // Texture15

- "u_CubeMapTexture", // CubeMapTexture
- "u_CubeMapTexture1", // CubeMapTexture1
- "u_CubeMapTexture2", // CubeMapTexture2
- "u_CubeMapTexture3", // CubeMapTexture3
- "u_CubeMapTexture4", // CubeMapTexture4
- "u_CubeMapTexture5", // CubeMapTexture5
- "u_CubeMapTexture6", // CubeMapTexture6
- "u_CubeMapTexture7", // CubeMapTexture7
- "u_CubeMapTexture8", // CubeMapTexture8
- "u_CubeMapTexture9", // CubeMapTexture9
- "u_CubeMapTexture10", // CubeMapTexture10
- "u_CubeMapTexture11", // CubeMapTexture11
- "u_CubeMapTexture12", // CubeMapTexture12
- "u_CubeMapTexture13", // CubeMapTexture13
- "u_CubeMapTexture14", // CubeMapTexture14
- "u_CubeMapTexture15", // CubeMapTexture15

- "u_Light[0].type", // LightType
- "u_Light[1].type", // LightType1
- "u_Light[2].type", // LightType2
- "u_Light[3].type", // LightType3
- "u_Light[4].type", // LightType4
- "u_Light[5].type", // LightType5
- "u_Light[6].type", // LightType6
- "u_Light[7].type", // LightType7

- "u_Light[0].ambient", // LightAmbient
- "u_Light[1].ambient", // LightAmbient1
- "u_Light[2].ambient", // LightAmbient2
- "u_Light[3].ambient", // LightAmbient3
- "u_Light[4].ambient", // LightAmbient4
- "u_Light[5].ambient", // LightAmbient5
- "u_Light[6].ambient", // LightAmbient6
- "u_Light[7].ambient", // LightAmbient7

- "u_Light[0].diffuse", // LightDiffuse
- "u_Light[1].diffuse", // LightDiffuse1
- "u_Light[2].diffuse", // LightDiffuse2
- "u_Light[3].diffuse", // LightDiffuse3
- "u_Light[4].diffuse", // LightDiffuse4
- "u_Light[5].diffuse", // LightDiffuse5
- "u_Light[6].diffuse", // LightDiffuse6
- "u_Light[7].diffuse", // LightDiffuse7

- "u_Light[0].intensity", // intensity
- "u_Light[1].intensity", // intensity
- "u_Light[2].intensity", // intensity
- "u_Light[3].intensity", // intensity
- "u_Light[4].intensity", // intensity
- "u_Light[5].intensity", // intensity
- "u_Light[6].intensity", // intensity
- "u_Light[7].intensity", // intensity

- "u_Light[0].specular", // LightSpecular
- "u_Light[1].specular", // LightSpecular1
- "u_Light[2].specular", // LightSpecular2
- "u_Light[3].specular", // LightSpecular3
- "u_Light[4].specular", // LightSpecular4
- "u_Light[5].specular", // LightSpecular5
- "u_Light[6].specular", // LightSpecular6
- "u_Light[7].specular", // LightSpecular7

- "u_Light[0].position", // LightPosition
- "u_Light[1].position", // LightPosition1
- "u_Light[2].position", // LightPosition2
- "u_Light[3].position", // LightPosition3
- "u_Light[4].position", // LightPosition4
- "u_Light[5].position", // LightPosition5
- "u_Light[6].position", // LightPosition6
- "u_Light[7].position", // LightPosition7

- "u_Light[0].direction", // LightDirection
- "u_Light[1].direction", // LightDirection1
- "u_Light[2].direction", // LightDirection2
- "u_Light[3].direction", // LightDirection3
- "u_Light[4].direction", // LightDirection4
- "u_Light[5].direction", // LightDirection5
- "u_Light[6].direction", // LightDirection6
- "u_Light[7].direction", // LightDirection7

- "u_Light[0].halfplane", // LightHalfplane
- "u_Light[1].halfplane", // LightHalfplane1
- "u_Light[2].halfplane", // LightHalfplane2
- "u_Light[3].halfplane", // LightHalfplane3
- "u_Light[4].halfplane", // LightHalfplane4
- "u_Light[5].halfplane", // LightHalfplane5
- "u_Light[6].halfplane", // LightHalfplane6
- "u_Light[7].halfplane", // LightHalfplane7

- "u_Light[0].attenuation", // LightAttenuation
- "u_Light[1].attenuation", // LightAttenuation1
- "u_Light[2].attenuation", // LightAttenuation2
- "u_Light[3].attenuation", // LightAttenuation3
- "u_Light[4].attenuation", // LightAttenuation4
- "u_Light[5].attenuation", // LightAttenuation5
- "u_Light[6].attenuation", // LightAttenuation6
- "u_Light[7].attenuation", // LightAttenuation7

- "u_Light[0].spotCosCutoff", // LightSpotCosCutoff
- "u_Light[1].spotCosCutoff", // LightSpotCosCutoff1
- "u_Light[2].spotCosCutoff", // LightSpotCosCutoff2
- "u_Light[3].spotCosCutoff", // LightSpotCosCutoff3
- "u_Light[4].spotCosCutoff", // LightSpotCosCutoff4
- "u_Light[5].spotCosCutoff", // LightSpotCosCutoff5
- "u_Light[6].spotCosCutoff", // LightSpotCosCutoff6
- "u_Light[7].spotCosCutoff", // LightSpotCosCutoff7

- "u_Light[0].spotExponent", // LightSpotExponent
- "u_Light[1].spotExponent", // LightSpotExponent1
- "u_Light[2].spotExponent", // LightSpotExponent2
- "u_Light[3].spotExponent", // LightSpotExponent3
- "u_Light[4].spotExponent", // LightSpotExponent4
- "u_Light[5].spotExponent", // LightSpotExponent5
- "u_Light[6].spotExponent", // LightSpotExponent6
- "u_Light[7].spotExponent", // LightSpotExponent7

- "u_Light[0].range", // LightRange
- "u_Light[1].range", // LightRange1
- "u_Light[2].range", // LightRange2
- "u_Light[3].range", // LightRange3
- "u_Light[4].range", // LightRange4
- "u_Light[5].range", // LightRange5
- "u_Light[6].range", // LightRange6
- "u_Light[7].range", // LightRange7

- "u_Light[0].enabled", // LightEnabled
- "u_Light[1].enabled", // LightEnabled1
- "u_Light[2].enabled", // LightEnabled2
- "u_Light[3].enabled", // LightEnabled3
- "u_Light[4].enabled", // LightEnabled4
- "u_Light[5].enabled", // LightEnabled5
- "u_Light[6].enabled", // LightEnabled6
- "u_Light[7].enabled", // LightEnabled7

- "u_Light[0].cameraLookAtVector",// LightCameraLookAtVector
- "u_Light[1].cameraLookAtVector",// LightCameraLookAtVector1
- "u_Light[2].cameraLookAtVector",// LightCameraLookAtVector2
- "u_Light[3].cameraLookAtVector",// LightCameraLookAtVector3
- "u_Light[4].cameraLookAtVector",// LightCameraLookAtVector4
- "u_Light[5].cameraLookAtVector",// LightCameraLookAtVector5
- "u_Light[6].cameraLookAtVector",// LightCameraLookAtVector6
- "u_Light[7].cameraLookAtVector",// LightCameraLookAtVector7

Shader Consistency Checker

Shader Consistency Checker refers to validations of the shader params (attributes and uniforms defined in shaders) based on some rules. Most shader param problems are reported for appearances of nodes after checking the vertex buffer of the mesh versus the shader program referenced by the appearance versus the uniform setter of the appearance.

Warning	Diagnostic	Possible resolutions

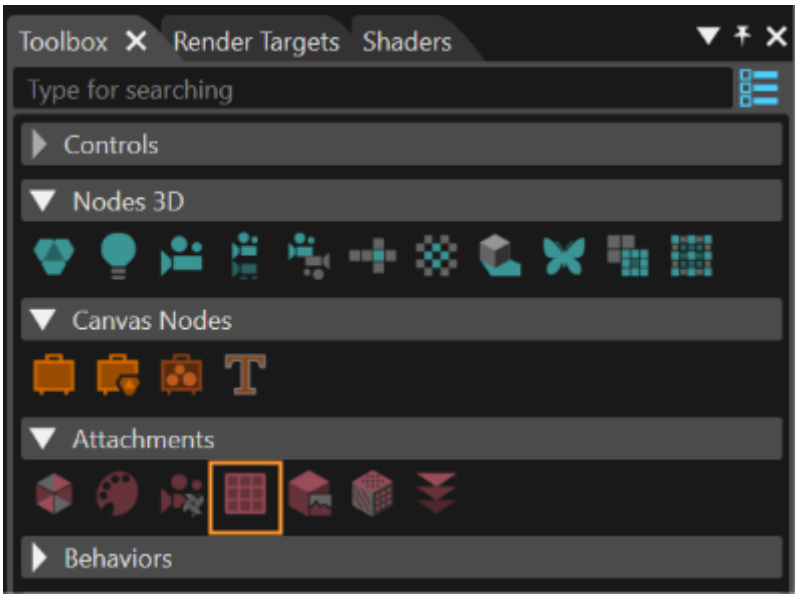
SemanticNotMappedToUniform, SemanticNotMappedToAttribute	This problem is reported when a uniform or attribute semantic is not mapped to any parameter name in the shader.	• change the attribute mapping
ShaderAttributeNotMappedToSemantic	This problem is reported when the shader contains an attribute which is not mapped to a semantic.	• edit shader in case the attribute was spelt incorrectly • change the attribute mapping
VBAttributeMissing	This problem is reported when an attribute defined in the shader has no matching data in the vertex buffer.	• generate the missing attribute (if possible) • select a different shader program or vertex buffer
VBAttributeTypeInvalid	This problem is reported when the type of the vertex data or the type of the attribute defined in the shader is invalid or there was an error retrieving them.	• select a different shader program or vertex buffer
VBAttributeTypeExtraChannels	This problem is reported when the vertex data contains too many channels compared to the attribute defined in the shader. For example an attribute is defined in the shader as a vector of 2 floats and the vertex data has a vector of 3 floats.	• regenerate the attribute from the vertex buffer (if possible) • edit the shader • select a different shader program or vertex buffer
VBAttributeTypePotentialPrecisionLoss	This problem is reported when there is a (potential) precision loss while loading the attribute data contained in the vertex buffer into the attributed defined in the shader. For example vertex data type is Float32 and the shade attribute type is Medium Float.	• change the precision of the attribute from the vertex buffer (if possible) • edit the shader • select a different shader program or vertex buffer
VBAttributeTypePrecisionWaste	This problem is reported when there is a "precision waste" while loading the attribute data contained in the vertex buffer into the attribute defined in the shader. For example vertex data type is Int8 and the shader attribute type is High Int.	• change the precision of the attribute from the vertex buffer (if possible) • edit the shader • select a different shader program or vertex buffer
VBAttributeTypeConversionToInt, VBAttributeTypeConversionToFloat	This problem is reported when there is a type conversion while loading the attribute data contained in the vertex buffer into the attribute defined in the shader. For example vertex data type is float and the shader attribute type is int.	• edit the shader • select a different shader program or vertex buffer

SemanticNotSupportedByUniformSetterButUsedByShaderProgram	This problem is reported when the semantic of a uniform defined in the shader is not supported by the uniform setter. Currently all semantic are supported.	
SemanticNotEnabledInUniformSetterButUsedByShaderProgram	This problem is reported when the semantic of a uniform defined in the shader is not enabled in the uniform setter.	• enable the semantic in the uniform setter • enable flag auto activation for the uniform setter • select a different shader program or vertex buffer
SemanticEnabledInUniformSetterButNotUsedByShaderProgram	This problem is reported when the semantic of a uniform not defined in the shader is enabled in the uniform setter.	• disable enable the semantic in the uniform setter • enable flag auto activation for the uniform setter • select a different shader program or vertex buffer

Generic ShaderParamSetter

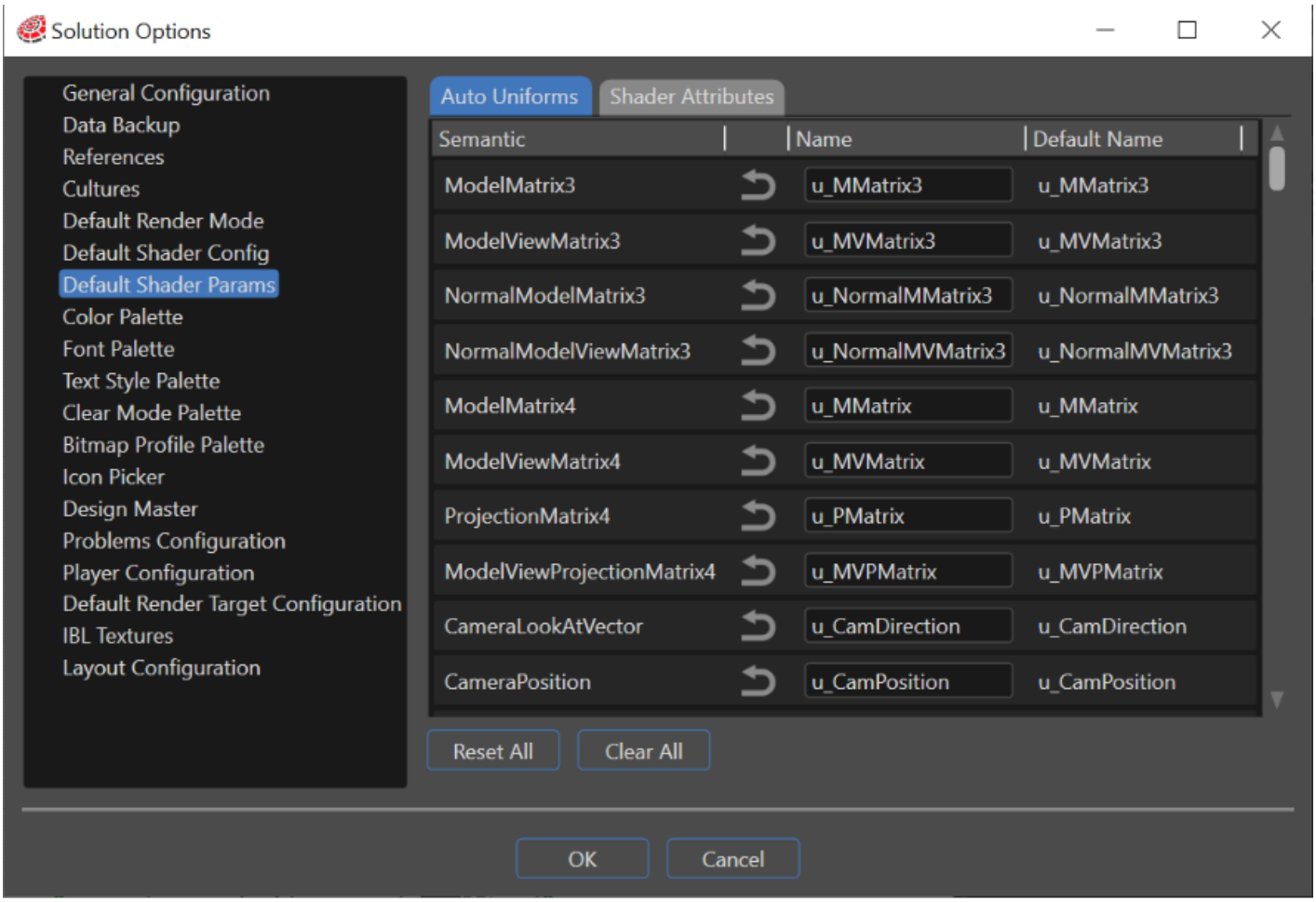
Generate special effects by using a uniform setter associated to a given node in a 3D scene. This can be done by realizing a correspondence between the configuration proper to the UniformSetter and all the parameters of the available shaders.

To get the desired effect, first, the UniformSetter from the Toolbox should be dragged in the the Scene extra-Tree over the node's Appearance.



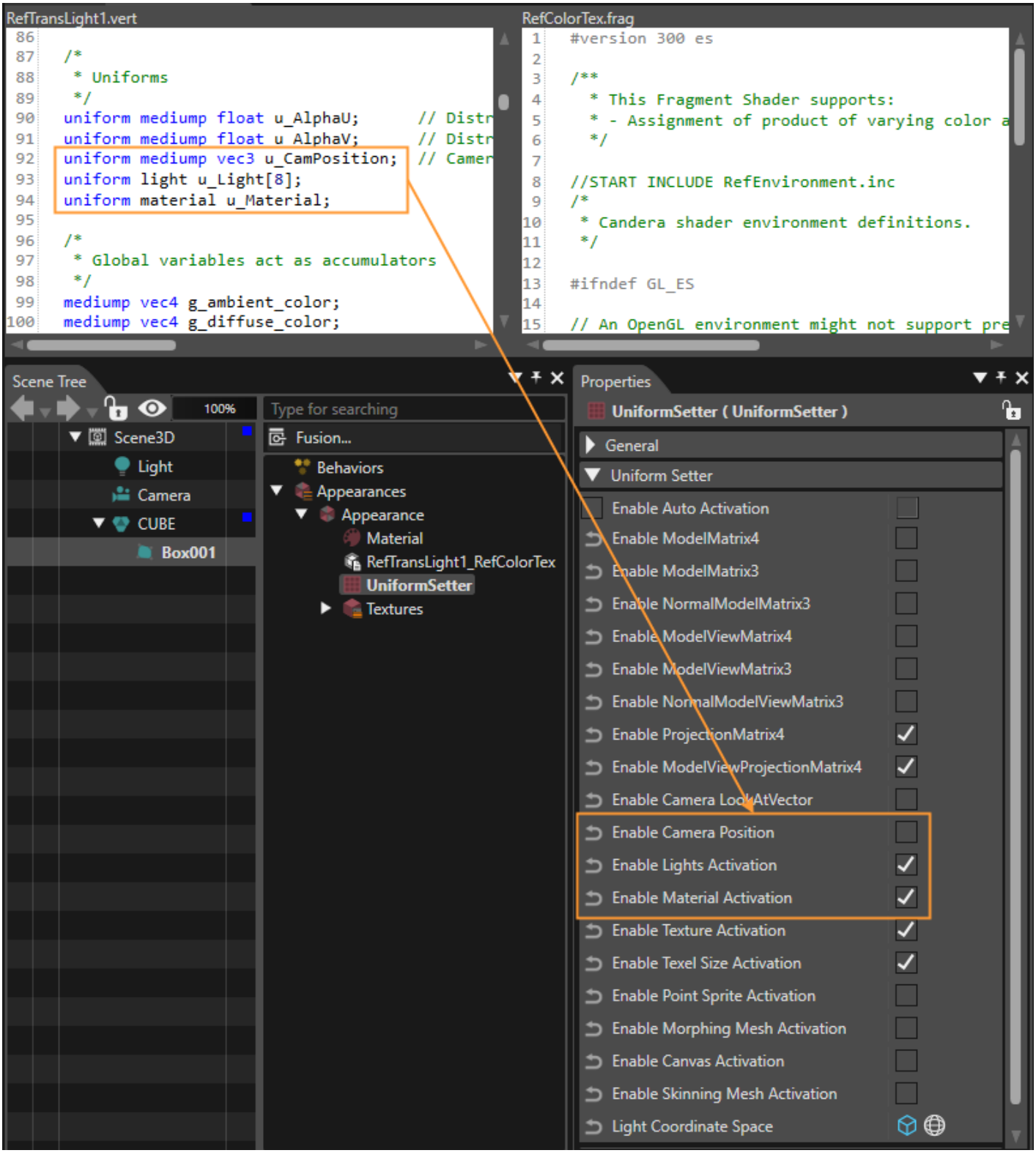
A relationship between the semantics of the UniformSetter and all the properties of the shader should be established. The semantics of the UniformSetter contain specific values that could be activated/deactivated in direct relationship with the shader. In In [Solution Option >Default Shader

Params] from the menu bar allows to change the default names of all these values specific for the semantics of the UniformSetter.



For example, if under an appearance template we have a RefTransLight2SphereMap_RefColorTex shader, as a first step it is necessary to open the Shader Editor panel and search all the uniforms required by this shader.

As a second step, configure the semantic of the UniformSetter in order to create a correspondence between the shader required uniforms and enabled Uniforms in the generic UniformSetter category in the Properties panel.

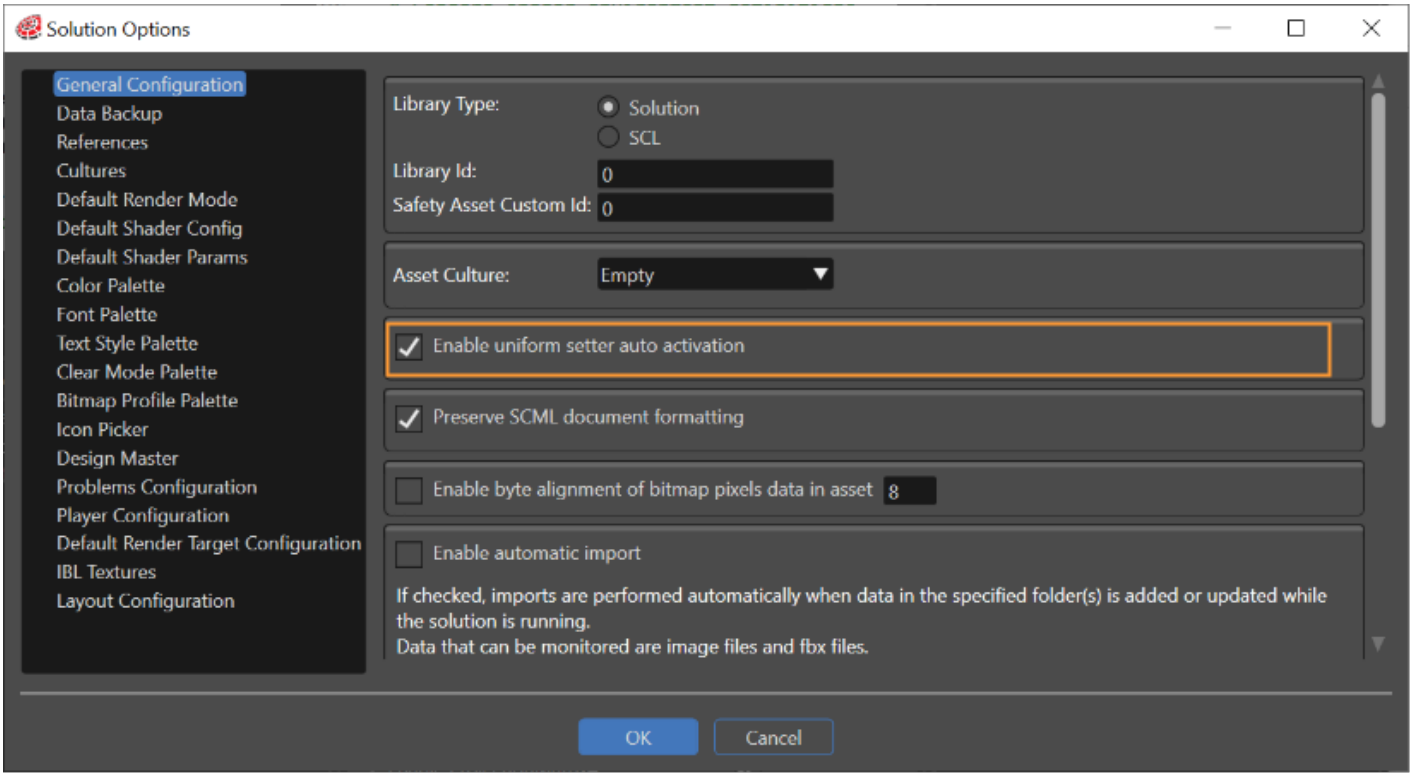


Find all the semantic uniform values of the RefTransLight1SphereMap_RefColorTex shader in the Shader Editor.

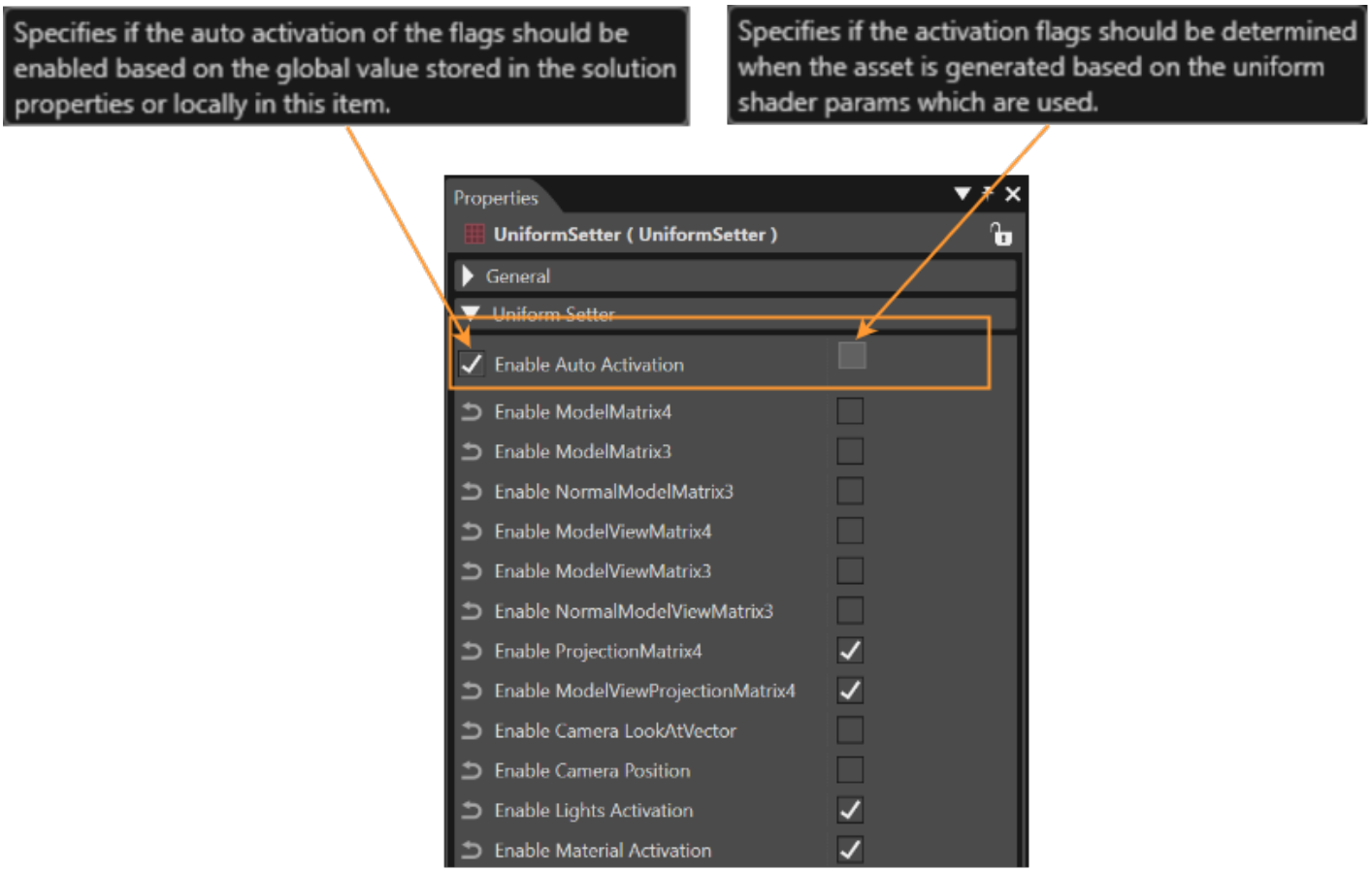
> The given uniform semantic of the UniformSetter in the Shaders Params Editor dialog have to be the same with Uniforms in the generic UniformSetter in the Properties panel.

Enable Auto Activation Property

There is an option to avoid the detailed (i.e. manual) setting of the uniform setters by using the auto activation property. This is possible after the setting of the "Enable uniform setter auto activation" property in the Solution Option panel (section "General Configuration"). If this setting is used and the "Auto activation scope" property is set global, the "Enable auto activation" property will become active and all the activation flags will be determined when the asset is generated based on the auto uniforms which are defined in the shader program.

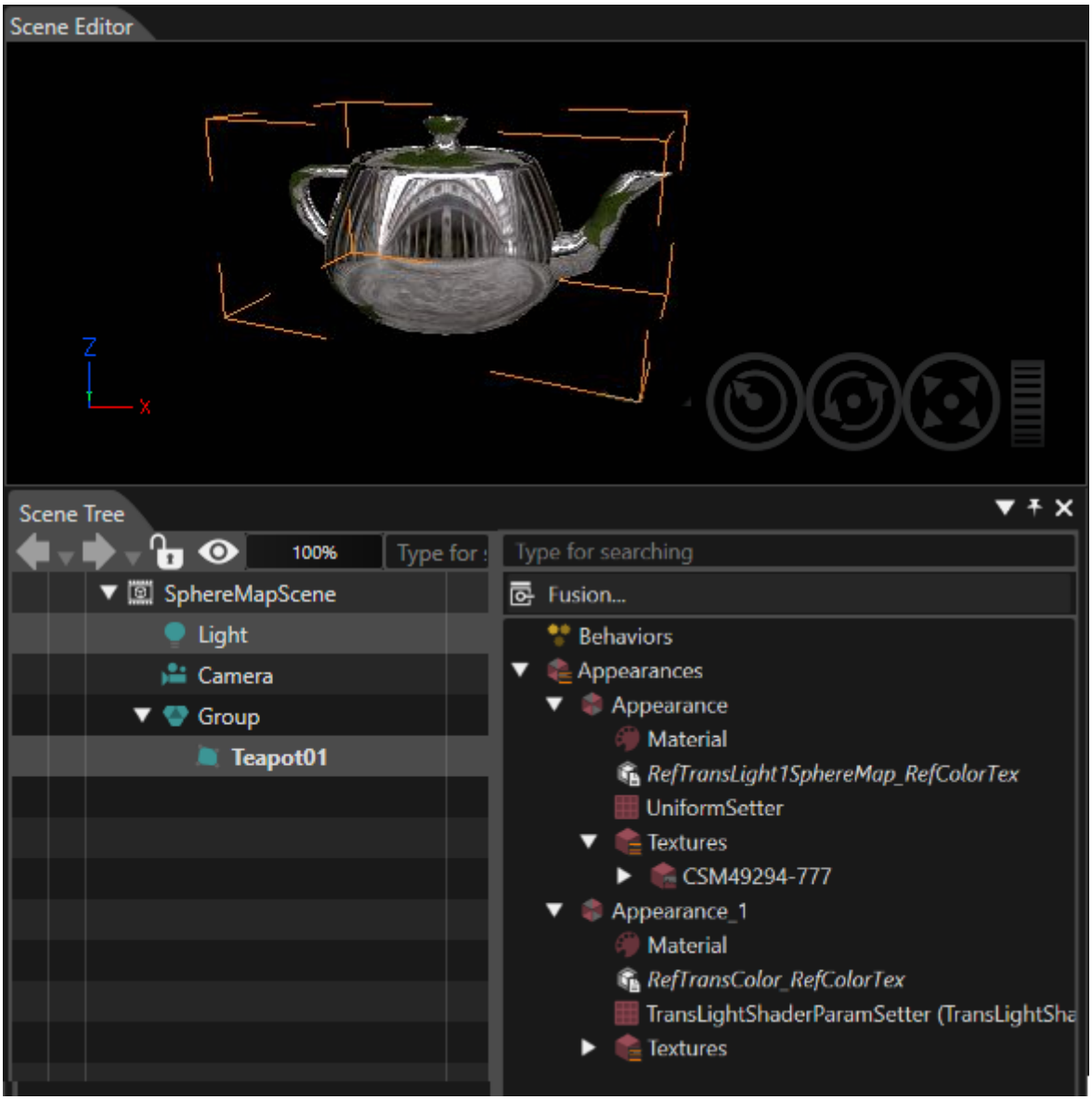


After this operation, a check box with the same name will become available in the Properties panel. This property specifies if the auto activation of the flags should be enabled based on the global values stored in the solution properties or locally in this item.

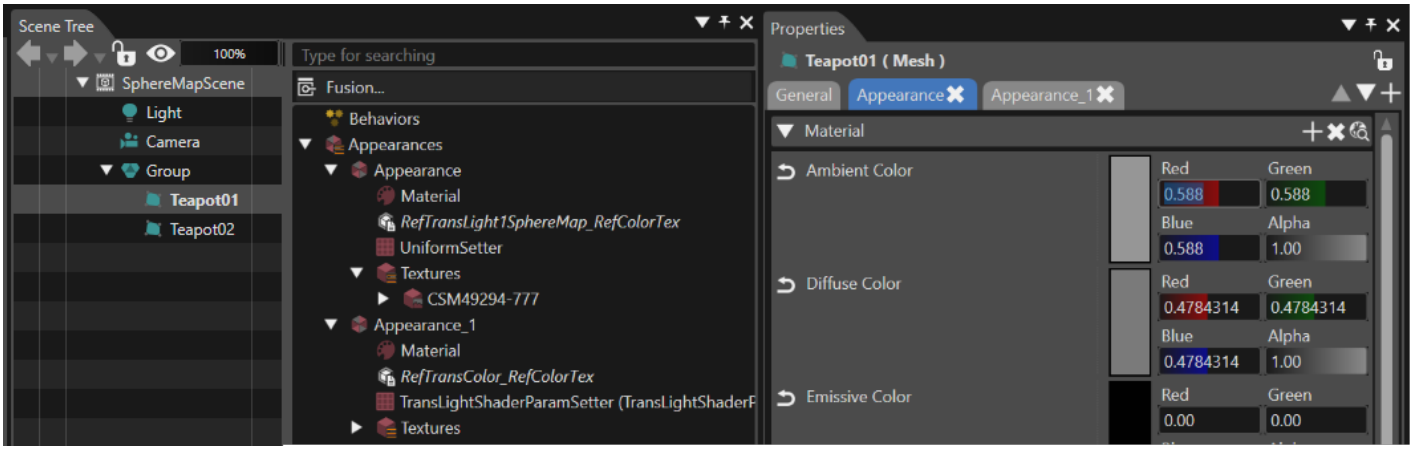


Multi-Pass Effect Configuration

Another way of getting complex effects is combining multiple appearances. The fact that there are multiple appearances attached on the same node implies that the node is rendered with each appearance at a time. All the appearances are rendered sequentially.



For example, if two different appearances associated at the same node after the UniformSetters of the first appearance are set, the second appearance should be set. Every appearance can be configured by using its own properties panel. However, it is possible to see both appearances in the general Properties panel as different tags of it.



Clone, Rename, Delete a Shader

Use the context menu of a shader program in the Shaders panel to trigger one of the following options:

- **"Clone"** will make a copy of the shader program. The name of the copy will be the name of the item suffixed with a number.
- **"Delete"** will cause a "Delete solution item" dialog to appear. If the item is not used somewhere in the solution, it can be deleted by clicking "OK". If the item is used, the "Delete Solution Item" dialog will list the paths of the items where it is used, and the "OK" is disabled, making the deletion not possible.
- **"Rename"** will cause a "Rename Solution Item" dialog to appear. The new desired name can be typed in the text box.

Revision #26
Created 2023-02-15 05:52:40 UTC by Tsuyoshi.Kato
Updated 2025-10-21 03:48:24 UTC by Tsuyoshi.Kato