# Python Access to the Translated V-Ray Scene

This page provides information on the V-Ray scene access python API.

## Overview

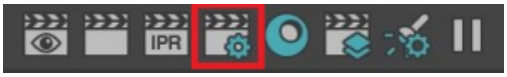
The V-Ray scene access python API allows you to modify the V-Ray scene after it is translated by the V-Ray for Maya translator, and before it is rendered and/or exported to a .vrscene file. Note that the V-Ray scene may be quite different from its representation in Maya. As such, some knowledge about the way V-Ray translates various Maya constructs may be required. It would be best to study .vrscene files exported by V-Ray for Maya.

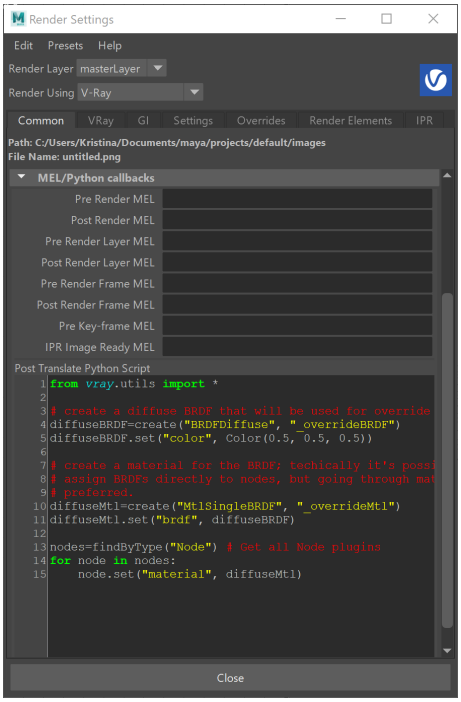
The scene access API allows you to expand the V-Ray for Maya translator by providing custom translation for constructs that are not recognized by V-Ray, or for modifying the scene before rendering without changing the original Maya scene.

You can specify the post-translate python script to be executed in the VRayCommon tab of the Render Settings dialog, in the MEL/Python callbacks rollout. The script is executed right after the scene is translated, and before it is rendered and/or exported to a .vrscene file. At present, when rendering an animation, the script is executed just once, before any frame is rendered.

For more information on various types of script access , see the Scripting page.

**UI Path**: ||Render Settings window|| > **Common tab** > **MEL/Python callbacks** rollout




## Available Python Functions and Classes

The declarations (and in some cases the implementations) for all the python functions and classes are available in the C:\Program Files\Autodesk\Maya20xx\vray\scripts\vray.

## Functions

The following python functions are available in the vray.utils module.

**create(pluginType, pluginName)** – create a plugin with the given name and type.

**delete(pluginName)** – delete a plugin with the given name.

**findByName(pattern)** – return a list of all plugins of a given name. **pattern** may contain wildcards.

**findByType(pattern)** – return a list of all plugins of a given type. **pattern** may contain wildcards.

**getPluginParams(plugin, getValues=**False**, onlyExisting=**False**)** – return a list of all available parameters for the given plugin. If **getValues** is True the value for each parameter is also returned. If **onlyExisting** is True, then only parameters that are actually set in the V-Ray scene are returned; otherwise all plugin parameters are returned and the ones that are not set in the V-Ray scene are listed with their default values.

**getTypeParams(pluginType)** – return a list of all parameters for a given plugin type.

**exportTexture(texureName)** – export the texture from the Maya scene and return the exported plugin. This function can be useful if the texture hasn't been exported while translating the scene.

**exportMaterial(materialName)** – export the material from the Maya scene and return the exported plugin. This function can be useful if the material hasn't been exported while translating the scene.

**addSceneContent(scenefile, prefix="")** – load all plugins from the given .vrscene file and insert them in the current scene. You can optionally specify a prefix that will be prepended to every plugin's name. The scene file is loaded in a new namespace from the rest of the scene - if there are plugins that have the same name as other plugins in the scene, they remain as separate plugins.

**appendSceneContent(scenefile, prefix="")** – load all plugins from the given .vrscene file and append them to the current scene, with an optional prefix. The plugins are appended in the namespace of the last loaded scene, so if there are already existing plugins with the same names, the data for their parameters is appended to the existing plugins. This can be used to load multiple .vrscene files from the same animated sequence. Wildcards can be used to specify scene files.

## Classes

The following python classes are available in the vray.utils module.

### Plugin

This class represents an instance of a V-Ray plugin in the scene.

**duplicate(self, newName)** – return a new plugin instance of the same type and with the same parameters.

**get(self, paramName)** – return the value of a parameter.

**has(self, paramName)** – check whether the plugin instance has a parameter with this name.

**name(self)** – return the name of the plugin instance.

**params(self, getValues=**False**, onlyExisting=**False**)** – return a list of all available parameters for this plugin. If **getValues** is True the value for each parameter is also returned. If **onlyExisting** is True, then only parameters that are actually set in the V-Ray scene are returned; otherwise all plugin parameters are returned and the ones that are not set in the V-Ray scene are listed with their default values.
**output(self, paramName)** – return a reference to a plugin output parameter, which can be set as a value for parameters of other plugins.
**set(self, paramName, value)** – set the value of a parameter.
**type(self)** – get the type of the plugin instance.

### Classes Representing Plugin Parameter Values

Parameters of the V-Ray plugins in the scene can be simple numbers, or more complex data types. Simple values can be directly manipulated, whereas complex types are represented by a dedicated python class.

### PluginOutput

This class represents a reference to an output parameter of a V-Ray plugin. It is used whenever one plugin is connected to another plugin's output parameter.

### AColor

This class represents a four-component color value (red, green, blue, alpha).

### Color

This class represents a three-component color value (red, green, blue).

### Vector

This class represents a three-component vector or point in 3D space.

### Matrix

This class represents a 3x3 matrix.

### Transform

This class represents a 3x4 transformation in 3D space.

## V-Ray Plugin Parameters Reference

You can find the list of all V-Ray plugins and their respective parameters with short description in the V-Ray for Maya instalation:

`C:\Program Files\Autodesk\Maya20xx\vray\PluginDoc`

where Maya20xx is the installed Maya version.

## Examples

### Example: Overriding All Materials in the Scene with Grey

This example overrides the materials of all objects in the scene with a grey diffuse material.

```
from vray.utils import *

# create a diffuse BRDF that will be used for override
diffuseBRDF=create("BRDFDiffuse", "_overrideBRDF")
diffuseBRDF.set("color", Color(0.5, 0.5, 0.5))

# create a material for the BRDF; techically it's possible to
# assign BRDFs directly to nodes, but going through materials is
# preferred.
diffuseMtl=create("MtlSingleBRDF", "_overrideMtl")
diffuseMtl.set("brdf", diffuseBRDF)

nodes=findByType("Node") # Get all Node plugins
for node in nodes:
    node.set("material", diffuseMtl)
```

---

### Example: Creating a Slightly Rotated Rectangle Light

This example creates a rectangle light with a slightly rotated orientation.

```
from vray.utils import *
import math

ry=math.radians(-90)
rx=math.radians(40)

# Create a matrix to rotate around the Y axis
rotYmat=Matrix(Vector(math.cos(ry), 0, -math.sin(ry)),
               Vector(0, 1, 0),
               Vector(math.sin(ry), 0, math.cos(ry)))

# Create a matrix to rotate around the X axis
rotXmat=Matrix(Vector(1, 0, 0),
               Vector(0, math.cos(rx), -math.sin(rx)),
               Vector(0, math.sin(rx), math.cos(rx)))

# Compose the final matrix
rotMat=rotYmat*rotXmat

# Create the rectangle light
light=create("LightRectangle", "newLight")
light.set('transform', Transform(rotMat, Vector(0, 2, 0)))
```

---

## Example: Changing the Color of a Material and Moving a Node

This example script changes the color of the material of the first node in the scene (assuming the original material is a lambert one) and moves the node one unit up.

```
from vray.utils import *

l=findByType("Node") # Get all Node plugins
p=l[0].get("material") # Get the material of the first node
brdf=p.get("brdf") # Get the BRDF for the material
brdf.set("color_tex", Color(1.0, 0.0, 0.0)) # Set the BRDF color to red
t=l[0].get("transform") # Get the transformation for the first node
t.offs+=Vector(0.0, 1.0, 0.0) # Add one unit up
l[0].set("transform", t) # Set the new transformation
```

---

## Example: Converting Meshes to Subdivision Surfaces

This example shows how to converts all regular meshes in the scene to subdivision surfaces.

```
from vray.utils import *

def smooth(geom, maxSubdivs = None, edgeLength = None):
  subdiv = create('GeomStaticSmoothedMesh', geom.name() + '@subdivGeometry') # Create a smoothed mesh plugin
  subdiv.set('mesh', geom) # Set the base geometry for the subdivision to be the original mesh plugin
  nodes = findByType('Node') # Get a list of all nodes

  for node in nodes:
    if node.get('geometry') == geom: # If a node references the original geometry...
      node.set('geometry', subdiv) # ...replace it with the subdivided one

  if maxSubdivs is not None: # Set the max. subdivs if specified
    subdiv.set('use_globals', False)
    subdiv.set('max_subdivs', maxSubdivs)

  if edgeLength is not None: # Set the max. edge length if specified
    subdiv.set('use_globals', False)
    subdiv.set('edge_length', edgeLength)

p = findByType('GeomStaticMesh') # Find all mesh plugins
for geom in p: # Replace each mesh plugin with a smoothed one
        smooth(geom)
```

---

## Example: Instancing the First Node to Form a Helix

This example instances the first node in the scene a number of times to form a helix.

```
from vray.utils import *
import math

geomNodes = findByType('Node') # Get all Node plugins
sp = geomNodes[0] # The first node
geom = sp.get('geometry') # The geometry of the first node

r = 12 # helix radius
n = 20 # number of objects per helix turn
m = 3 # number of helix turns
h = 0.5 # height difference between objects on the helix

for i in range(n * m):
  d = sp.duplicate('sphereNode' + str(i)) # Create a copy of the first scene node
  tr = d.get('transform') # Get the transformation matrix
  tr.offs += Vector(r * math.cos(i * 2 * 3.14159 / n), i * h, r * math.sin(i * 2 * 3.14159 / n)) # Add an
offset according to the helix formula
  d.set('transform', tr) # Set the new transformation
```

---

## Example: Exporting and Changing Materials

This example exports several textures and materials and changes the material of an object.

```
from vray.utils import *

lambert = exportMaterial('lambert1') # Export lambert1 material
checker = exportTexture('checker1') # Export checker1 texture
cloth = exportTexture('cloth1') # Export cloth1 texture

brdf = lambert.get('brdf') # Get the 'lambert1@material' plugin
brdf.set('transparency_tex', checker.output('color')) # Set the transparency_tex parameter of the brdf
brdf.set('color_tex', cloth) # Set the color_tex parameter of the brdf

planeNode = Plugin('pPlaneShape1@node') # Get the plane node
planeNode.set('material', lambert) # Set the material of the plane
```

---

## Example: Combining the Contents of Several Scenes

This example adds the contents of several scenes and uses some of the newly created plugins.

```
from vray.utils import *

addSceneContent('blinn.vrscene', 'basic_') # Create all plugins in this scene file and prepend 'basic_' to the
names of all plugins
addSceneContent('vraymtl.vrscene') # Create all plugins in this scene file and use the original names

blinn = Plugin('basic_blinn1@material') # Get the blinn material from the first added scene
vraymtl = Plugin('VRayMtl1@material') # Get the V-Ray material from the second added scene

nodes = findByType('Node') # Find all nodes in the scene

# Change the materials of some of the nodes
nodes[0].set('material', blinn)
nodes[1].set('material', vraymtl)
```

---

## Example: Modifying the Parameters

This example demonstrates how to change the value of a more complex parameter. For instance, the ignored_lights parameter of the SettingsLightLinker plugin is a list of several sub-lists of plugins. The first plugin in each list is a light plugin and the rest of the plugins are geometry nodes. This is how the parameter can be modified:

```
from vray.utils import *

lightLinker = findByType('SettingsLightLinker')[0] # Get the SettingsLightLinker plugin
directLight = findByType('MayaLightDirect')[0] # Get the first directional light
spotLight = findByType('LightSpot')[0] # Get the first spot light
nodes = findByType('Node') # Get all geometry nodes in the scene

param = [[directLight, nodes[0], nodes[1]], [spotLight, nodes[2]]] # Just create a list of lists
lightLinker.set("ignored_lights", param)
```

## Example: Connect a Plugin Output

This example shows how to get a specific output from a plugin and use it as an input to another plugin's parameter.

First, we assume we know the name of an object in the scene. We get its material, then the material's BRDF, then the BRDF's diffuse texture. We create a TexAColorOp texture plugin, where we multiply the BRDF's diffuse texture by another color. The second color we multiply by could also be another texture reference. Finally, we use the 'product' output of TexAColorOP as the diffuse input of the BRDF. Any plugin's specific output can be used in the same way.

```
from vray.utils import *

# Assuming an object pPlaneShape1@node has a VRayMtl assigned
# Get the material's diffuse texture and pass it through a TexAColorOp
# Create a TexAColorOp plugin
# Multiply the texture by a color inside TexAColorOp
# Get the 'product' output from TexAColorOp and use it as the diffuse input in the VRayMtl

node = findByName('pPlaneShape1@node')[0] # Find the node by name
node_mtl = node.get('material') # Get the node's material
node_brdf = node_mtl.get('brdf') # Get the material's brdf to reach the BRDFVRayMtl plugin
diffuseTex = node_brdf.get('diffuse') # Get the diffuse texture of BRDFVRayMtl

colorOp = create('TexAColorOp', 'TexAColorOp1') # Create a TexAColorOp texture
colorOp.set('color_a', diffuseTex) # Set the VRayMtl's diffuse texture as color_a
colorOp.set('color_b', Color(1.0, 0.1, 0.1)) # Set a Color() as color_b

node_brdf.set('diffuse', colorOp.output('product')) # Set the 'product' output of TexAColorOp as the diffuse
input of BRDFVRayMtl

# Similarly, we can get the 'sum' or any other output
# node_brdf.set('diffuse', colorOp.output('sum'))
```