Tutorial 6: How to Work with the V2 Interface Using Services and Objects

The *Tutorial 6: How to Work with the V2 Interface Using Services and Objects* video covers examples with cameras, viewpoints, and sceneplates in VR.

Download the example scripts here: https://github.com/raemorris/VRED-Python-Tutorial-Examples-Scripts/blob/main/tutorial6.zip?raw=true

Sample Python Code

Here are the accompanying example Python scripts for the *Tutorial 6: How to Work with the V2 Interface Using Services and Objects* video.

- To view the video, visit: https://www.youtube.com/watch?v=ynYbsMU0oug
- To review the video caption, see *Video Captions*.

camera.py

```
import math


# Example 1.0
# Get active camera with vrCameraService
camera = vrCameraService.getActiveCamera()

# Set Focal Length
camera.setFocalLength(35)



# Example 2.0
# Creating a camera using API v1 and settings its fov to 35
vrCamera.selectCamera("Perspective")
camera_v1 = vrCamera.getActiveCameraNode()
```

```
# read field of view mode (0 := vertical, 1 := horizontal)
fov_mode = camera_v1.fields().getUInt32("fovMode")
sensor_size = camera_v1.fields().getVec("sensorSize", 2)

# calculate fov based on sensor size
fov = vrOSGWidget.getFov()
focal_length = 35

if fov_mode is 0:
    fov = 2 * math.degrees(math.atan((sensor_size[1] / 2) / focal_length))
else:
    fov = 2 * math.degrees(math.atan((sensor_size[0] / 2) / focal_length))

vrOSGWidget.setFov(fov)



# Example 3.0
# Creating a camera using API v2 and calculating the fov manually
camera_v2_1 = vrCameraService.createCamera("Camera v2_1")

# read field of view mode (vertical or horizontal)
fov_mode = camera_v2_1.getFovMode()

# read sensor size as QtVector2D
sensor_size = camera_v2_1.getSensorSize()

# calculate fov based on sensor size
fov = camera_v2_1.getFov()
focal_length = 50

if fov_mode is vrCameraTypes.FovMode.Vertical:
    fov = 2 * math.degrees(math.atan((sensor_size.y() / 2) / focal_length))
```

```
else:
    fov = 2 * math.degrees(math.atan((sensor_size.x() / 2) / focal_length))

camera_v2_1.setFov(fov)
```

light.py

```
# Example 1

# Get vrdPointLightNode with the vrLightService
myPointLight = vrLightService.findLight("MyPointLight")

# Set the ground shadow intensity
myPointLight.setGroundShadowIntensity(0.5)
```

service.py

```
#Example 1

# Get the active camera from the camera service
camera = vrCameraService.createCamera("My New Camera")

# Create a new camera track on the camera object
cameraTrack = vrCameraService.createCameraTrack("My New Camera Track",
camera)

# Create a new viewpoint on the camera track
viewpoint = vrCameraService.createViewpoint("My New Viewpoint", cameraTrack)



#Example 2
```

```
# Create a new pointlight with the vrLightService
newPointLight = vrLightService.createLight(
                    "My New PointLight",
                    vrLightTypes.LightType.Point
                )

# Set the translation of this pointlight
newPointLight.setTranslation(QVector3D(0,0,1000))



#Example 3

# Create a new camera with the vrCameraService
camera = vrCameraService.createCamera("New Camera")

# Create vectors that define the location and look-at point of the camera
cameraFrom = QVector3D(5000, 8000, 3000)
cameraAt = QVector3D(0, 0, 0)
cameraUp = QVector3D(0, 0, 1)

# Set the camera parameters
fromAtUp = vrCameraFromAtUp(cameraFrom, cameraAt, cameraUp)
camera.setFromAtUp(fromAtUp)

# Activate the camera
camera.activate()



# Create a new camera track on the camera object
cameraTrack = vrCameraService.createCameraTrack("My New Camera Track",
camera)
```

```
# Create a new viewpoint on the camera track
viewpoint = vrCameraService.createViewpoint("My New Viewpoint", cameraTrack)


viewpoint.createPreview(True)
```

Video Captions

Hello and welcome to this new Python tutorial for VRED Pro. My name is Christopher and today I will talk about the new API version 2, why it's different to work with, and how you use it.

With the release of the API version 2, Autodesk introduced a new object model that aims to make working with the Python interfaces in VRED easier. It moves critical functionality in, so called service classes, and offers more object-oriented data types to work with. To understand why this is a good thing, we first have to discuss the disadvantages of the previous API.

Let's have look at the following example. The version 1 API often used separate values as parameters. This is fine when you, for example, just want to work with the x, y, and z values of a single vector. But when a function expects multiple vectors, or vector components, this can get very confusing. A better approach would be to encapsulate these parameters in an object that clearly describes what it's supposed to be, like a vector. The same is true for return values.

In the API version 1, we see functions that either return a single value or a list of values, if the return values are the components of a vector. A developer has to know the return type of a function to know what this list of return values is supposed to be. When the function returns a single object, like a vector, it is immediately recognizable as such.

Another disadvantage of API version 1 is, that it was based on Python 2.7. Not that Python 2.7 is a bad release, but Python 3 made it much easier to offer a fully statically typed API. Dynamic typing certainly has its moments, but it is way easier to build a consistent API with static types.

The API version 2, in my opinion, makes a lot of things better. First, it introduced several specialized service classes that now act as the main gateway to interact with VRED. Services like vrNodeService or vrSessionService combine all functionality for that particular topic. The session service, for example, contains functions to manage collaborative sessions. The camera service enables you to manage cameras and viewpoints. This makes finding the correct functions much easier, because, for example, all camera-related functionality is now located in the `vrCameraService` and the new camera types vrdCameraNode and vrdCameraBaseNode.

The new API introduced a whole bunch of data types that encapsulate the functionality of a piece of data. These new types are all prefixed with "vrd". Like I just mentioned, the vrdCameraNode, for example, contains all information about a camera in your scene. Likewise, an object of type vrdSessionUser holds all information about a particular user in a collaborative session.

Each new type contains functions to add or change data of an object. So, if you want to change the settings of a camera, like its focal length, you would first get an object of type vrdCameraNode from the vrCameraService and then use the function setFocalLength to change the setting. There are lots of other

new types for all kind of data in VRED and we can barely scratch the surface here. Like for example, the vrdPointLightNode type that provides exactly the functionality you need to manipulate your point light sources in the scene.

Another advantage is the range of other supporting data types that come with the new API version 2. I already mentioned the vector data types. They combine separate numbers that are now clearly labeled as a vector. This makes your code far more readable and understandable. Each developer immediately knows that this function expects a vector as an input and that another function returns a vector. Another great plus is, that these new data types come with math operations and you can intuitively work with them. Okay, let's have a look at some examples.

First, I will show the basic usage of the new services. There isn't really anything difficult in using them. In this example, we want to see how we can work with cameras using the vrCameraService. At first, let's create a new camera in our scene. We use the function createCamera from the vrCameraService and give it a name. The return value of this function is a vrdCameraNode that we assign to a variable.

Now we can use this camera node object to create a new camera track. This is also done with the vrCameraService and the function createCameraTrack. This time, the return value is a vrdCameraTrack object that we also store in a variable.

We can spin this further and create a new viewpoint on this camera track. All we have to do is to use the createViewpoint function from the "camera service" and use the camera track object as an input parameter. When we have a look at the Camera Editor, we can see all the objects we created.

In the next example, we want to create some light sources in our scene. Again, we can use the vrLightService and its function createLight. This time, we have to supply a name and the light type as input parameters.

The API version 2 also includes classes that contain a type of some sort. In this case we need the vrLightType, "pointlight". Please don't confuse these types with the data types we were talking about earlier in this video. All classes that end with "type" in the new API define a set of distinct things, or, well, types. A light, for example, can be of type pointlight, spotlight, directional light, and so on. With this type, we can tell the light service which kind of light we want to create.

In our camera example, we already came in touch with some of the new "vrd objects". When we get back to the example, we cannot only create new objects with the camera service, but we can also change the camera, the camera track, and the viewpoint we created.

At first, I want to change the position of the camera. This can be done by defining the location where the camera should be, the point where it's looking at, and a direction that tell us where "up" is in our scene. We set these values by using the fromAtUp data type that we feed with the vectors we defined. This will aim the camera at the center of our scene. After that, we activate the camera to immediately see the results.

Of course, we can change our other objects in a similar way. For example, we can update the preview of our viewpoint with the function createPreview(). This will create a new preview of our viewpoint from the current scene.

In the last example, I want to show you how the new data types can make your life easier. We have a more detailed tutorial in an upcoming video; therefore, I will not go too deep right now. Just notice how we can use the new vector types to encapsulate our vector components and make the whole script more readable and also shorter.

On the right side, we create a few geometries and set its position in 3D space. On the left side, is the same example scripted with functions from the API version 1. At least for me, it's far more readable and understandable. You can also see that we can use the math operations provided by the vector types to make the script even shorter.

A last note on the documentation. In the documentation, you can see all the functions in version 1 that have been replaced by version 2. These functions are called "deprecated". Which basically means that they are still available but may be removed or replaced in the future.

For example, with version 2 almost all functions contained in vrCamera have been replaced by functions that are now contained either in vrCameraService, vrdCameraNode or vrdCameraBaseNode. You should avoid using deprecated functions when writing new scripts. They still work, but with the new API you are on the safe side. When you look at the top of the page, there is also a link that points to the classes in version 2 that now contain this functionality.

With the API version 2, Autodesk introduced new and exciting changes to VREDs Python interface. When you are familiar with writing scripts, you should have no problem using the new classes and data types. And when you are just starting, I hope I could give you some ideas where to begin. That's it for today. I hope you enjoyed this video and see you next time.