USDHook(bpy_struct)

USD Hook Example

This example shows an implementation of `USDHook` to extend USD export and import functionality.

Callback Function API

One may optionally define any or all of the following callback functions in the `USDHook` subclass.

on_export

Called before the USD export finalizes, allowing modifications to the USD stage immediately before it is saved.

Args:

- `export_context` (USDSceneExportContext): Provides access to the stage and dependency graph

Returns:

- `True` on success or `False` if the operation was bypassed or otherwise failed to complete

on_material_export

Called for each material that is exported, allowing modifications to the USD material, such as shader generation.

Args:

- `export_context` (USDMaterialExportContext): Provides access to the stage and a texture export utility function
- `bl_material` (`bpy.types.Material`): The source Blender material
- `usd_material` (`pxr.UsdShade.Material`): The target USD material to be exported

Returns:

- `True` on success or `False` if the operation was bypassed or otherwise failed to complete

Note that the target USD material might already have connected shaders created by the USD exporter or by other material export hooks.

on_import

Called after the USD import finalizes.

Args:

- `import_context` (USDSceneImportContext): Provides access to the stage and a map associating USD prim paths and Blender IDs

Returns:

- `True` on success or `False` if the operation was bypassed or otherwise failed to complete

material_import_poll

Called to determine if the `USDHook` implementation can convert a given USD material.

Args:

- `import_context` (USDMaterialImportContext): Provides access to the stage and a texture import utility function
- `usd_material` (`pxr.UsdShade.Material`): The source USD material to be exported

Returns:

- `True` if the hook can convert the material or `False` otherwise

If any hook returns `True` from `material_import_poll`, the USD importer will skip standard `USD Preview Surface` or

`MaterialX` import and invoke the hook's on_material_import method to convert the material instead.

on_material_import

Called for each material that is imported, to allow converting the USD material to nodes on the Blender material. To ensure that this function gets called, the hook must also implement the `material_import_poll()` callback to return `True` for the given USD material.

Args:

- `import_context` (USDMaterialImportContext): Provides access to the stage and a texture import utility function
- `bl_material` (`bpy.types.Material`): The target Blender material with an empty node tree
- `usd_material` (`pxr.UsdShade.Material`): The source USD material to be imported

Returns:

- `True` on success or `False` if the conversion failed or otherwise did not complete

Context Classes

Instances of the following built-in classes are provided as arguments to the callbacks.

USDSceneExportContext

Argument for on_export.

Methods:

- `get_stage()` : returns the USD stage to be saved
- `get_depsgraph()` : returns the Blender scene dependency graph

USDMaterialExportContext

Argument for on_material_export.

Methods:

- `get_stage()` : returns the USD stage to be saved
- `export_texture(image: bpy.types.Image)` : Returns the USD asset path for the given texture image

The `export_texture` function will save in-memory images and may copy texture assets, depending on the current USD export options. For example, by default calling `export_texture(/foo/bar.png)` will copy the file to a `textures` directory next to the exported USD and will return the relative path `./textures/bar.png`.

USDSceneImportContext

Argument for on_import.

Methods:

- `get_prim_map()` returns a `dict` where the key is an imported USD Prim path and the value a `list` of the IDs created by the imported prim.
- `get_stage()` returns the USD stage which was imported.

USDMaterialImportContext

Argument for material_import_poll and on_material_import.

Methods:

- `get_stage()` : returns the USD stage to be saved.
- `import_texture(asset_path: str)` : for the given USD texture asset path, returns a `tuple[str, bool]`, containing the asset local path and a bool indicating whether the path references a temporary file.

The `import_texture` function may copy the texture to the local file system if the given asset path is a package-relative path for a USDZ archive, depending on the current USD `Import Textures` options. When the `Import Textures` mode is `Packed`, the texture is saved to a

depending on the current USD `Import Textures` options. When the `Import Textures` mode is `Packed`, the texture is saved to a temporary location and the second element of the returned tuple is `True`, indicating that the file is temporary, in which case it may be necessary to pacl the image. The original asset path will be returned unchanged if it's already a local file or if it could not be copied to a local destination.

Errors

Exceptions raised by these functions will be reported in Blender with the exception details printed to the console.

Example Code

The `USDHookExample` class in the example below implements the following functions:

- `on_export()` function to add custom data to the stage's root layer.
- `on_material_export()` function to create a simple `MaterialX` shader on the given USD material.
- `on_import()` function to create a text object to display the stage's custom layer data.
- `material_import_poll()` returns `True` if the given USD material has an `mtlx` context.
- `on_material_import()` function to convert a simple `MaterialX` shader with a `base_color` input.

```
bl_info = {
    "name": "USD Hook Example",
    "blender": (4, 4, 0),
}

import bpy
import bpy.types
import textwrap

# Make `pxr` module available, for running as `bpy` PIP package.
bpy.utils.expose_bundled_modules()

import pxr.Gf as Gf
import pxr.Sdf as Sdf
import pxr.Usd as Usd
import pxr.UsdShade as UsdShade


class USDHookExample(bpy.types.USDHook):
    """Example implementation of USD IO hooks"""
    bl_idname = "usd_hook_example"
    bl_label = "Example"

    @staticmethod
    def on_export(export_context):
        """ Include the Blender filepath in the root layer custom data.
        """

        stage = export_context.get_stage()

        if stage is None:
            return False
        data = bpy.data
        if data is None:
            return False

        # Set the custom data.
        rootLayer = stage.GetRootLayer()
```

```
        rootLayer = stage.GetRootLayer()
        customData = rootLayer.customLayerData
        customData["blenderFilepath"] = data.filepath
        rootLayer.customLayerData = customData

        return True

    @staticmethod
    def on_material_export(export_context, bl_material, usd_material):
        """ Create a simple MaterialX shader on the exported material.
        """

        stage = export_context.get_stage()

        # Create a MaterialX standard surface shader
        mtl_path = usd_material.GetPrim().GetPath()
        shader = UsdShade.Shader.Define(stage, mtl_path.AppendPath("mtlxstandard_surface")
        shader.CreateIdAttr("ND_standard_surface_surfaceshader")

        # Connect the shader.  MaterialX materials use "mtlx" renderContext
        usd_material.CreateSurfaceOutput("mtlx").ConnectToSource(shader.ConnectableAPI(),

        # Set the color to the Blender material's viewport display color.
        col = bl_material.diffuse_color
        shader.CreateInput("base_color", Sdf.ValueTypeNames.Color3f).Set(Gf.Vec3f(col[0],

        return True

    @staticmethod
    def on_import(import_context):
        """ Create a text object to display the stage's custom data.
        """
        stage = import_context.get_stage()

        if stage is None:
            return False

        # Get the custom data.
        rootLayer = stage.GetRootLayer()
        customData = rootLayer.customLayerData

        # Create a text object to display the stage path
        # and custom data dictionary entries.

        bpy.ops.object.text_add()
        ob = bpy.context.view_layer.objects.active

        if (ob is None) or (ob.data is None):
            return False

        ob.name = "layer_data"
        ob.data.name = "layer_data"

        # The stage root path is the first line.
        text = rootLayer.realPath
```

```
        # Append key/value strings, enforcing text wrapping.
        for item in customData.items():
            print(item)
            text += '\n'
            line = str(item[0]) + ': ' + str(item[1])
            text += textwrap.fill(line, width=80)

        ob.data.body = text

        return True

    @staticmethod
    def material_import_poll(import_context, usd_material):
        """
        Return True if the given USD material can be converted.
        Return False otherwise.
        """
        # We can convert MaterialX.
        surf_output = usd_material.GetSurfaceOutput("mtlx")
        return bool(surf_output)

    @staticmethod
    def on_material_import(import_context, bl_material, usd_material):
        """
        Import a simple mtlx material.  Just handle the base_color input
        of a ND_standard_surface_surfaceshader.
        """

        # We must confirm that we can handle this material.
        surf_output = usd_material.GetSurfaceOutput("mtlx")
        if not surf_output:
            return False

        if not surf_output.HasConnectedSource():
            return False

        # Get the connected surface output source.
        source = surf_output.GetConnectedSource()
        # Get the shader prim from the source
        shader = UsdShade.Shader(source[0])
        shader_id = shader.GetShaderId()
        if shader_id != "ND_standard_surface_surfaceshader":
            return False

        color_attr = shader.GetInput("base_color")
        if color_attr is None:
            return False

        # Create the node tree
        bl_material.use_nodes = True
        node_tree = bl_material.node_tree
        nodes = node_tree.nodes
        bsdf = nodes.get("Principled BSDF")
        assert bsdf
```

```
        bsdf_base_color_input = bsdf.inputs['Base Color']

        # Try to set the default color value.
        # Get the authored default value
        color = color_attr.Get()

        if color is None:
            return False

        bsdf_base_color_input.default_value = (color[0], color[1], color[2], 1)

        return True


def register():
    bpy.utils.register_class(USDHookExample)


def unregister():
    bpy.utils.unregister_class(USDHookExample)


if __name__ == "__main__":
    register()
```

base class — bpy_struct

class bpy.types.**USDHook(bpy_struct)**

Defines callback functions to extend USD IO

bl_description

A short description of the USD hook

TYPE:

string, default “”, (never None)

bl_idname

TYPE:

string, default “”, (never None)

bl_label

TYPE:

string, default “”, (never None)

classmethod bl_rna_get_subclass(id, default=None)

PARAMETERS:

id (*str*) – The RNA type identifier.

RETURNS:

The RNA type or default when not found.

RETURN TYPE:

bpy.types.Struct subclass

classmethod bl_rna_get_subclass_py(id, default=None)

PARAMETERS:

PARAMETERS:

id (*str*) – The RNA type identifier.

RETURNS:

The class or default when not found.

RETURN TYPE:

type

Inherited Properties

- bpy_struct.id_data

Inherited Functions

- bpy_struct.as_pointer
- bpy_struct.driver_add
- bpy_struct.driver_remove
- bpy_struct.get
- bpy_struct.id_properties_clear
- bpy_struct.id_properties_ensure
- bpy_struct.id_properties_ui
- bpy_struct.is_property_hidden
- bpy_struct.is_property_overridable_library
- bpy_struct.is_property_readonly
- bpy_struct.is_property_set
- bpy_struct.items
- bpy_struct.keyframe_delete
- bpy_struct.keyframe_insert
- bpy_struct.keys
- bpy_struct.path_from_id
- bpy_struct.path_resolve
- bpy_struct.pop
- bpy_struct.property_overridable_library_set
- bpy_struct.property_unset
- bpy_struct.type_recast
- bpy_struct.values