Skip to content

Depsgraph(bpy_struct)

Dependency graph: Evaluated ID example

This example demonstrates access to the evaluated ID (such as object, material, etc.) state from an original ID. This is needed every time one needs to access state with animation, constraints, and modifiers taken into account.

```
import bpy


class OBJECT_OT_evaluated_example(bpy.types.Operator):
    """Access evaluated object state and do something with it"""
    bl_label = "DEG Access Evaluated Object"
    bl_idname = "object.evaluated_example"

    def execute(self, context):
        # This is an original object. Its data does not have any modifiers applied.
        obj = context.object
        if obj is None or obj.type != 'MESH':
            self.report({'INFO'}, "No active mesh object to get info from")
            return {'CANCELLED'}
        # Evaluated object exists within a specific dependency graph.
        # We will request evaluated object from the dependency graph which corresponds to
        # current scene and view layer.
        #
        # NOTE: This call ensure the dependency graph is fully evaluated. This might be ex
        # if changes were made to the scene, but is needed to ensure no dangling or incorr
        # pointers are exposed.
        depsgraph = context.evaluated_depsgraph_get()
        # Actually request evaluated object.
        #
        # This object has animation and drivers applied on it, together with constraints a
        # modifiers.
        #
        # For mesh objects the object.data will be a mesh with all modifiers applied.
        # This means that in access to vertices or faces after modifier stack happens via
        # object_eval.object.
        #
        # For other types of objects the object_eval.data does not have modifiers applied
        # but has animation applied.
        #
        # NOTE: All ID types have `evaluated_get()`, including materials, node trees, worl
        object_eval = obj.evaluated_get(depsgraph)
        mesh_eval = object_eval.data
        self.report({'INFO'}, f"Number of evaluated vertices: {len(mesh_eval.vertices)}")
        return {'FINISHED'}


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


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


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

Dependency graph: Original object example

This example demonstrates access to the original ID. Such access is needed to check whether object is selected, or to compare pointers.

```
import bpy


class OBJECT_OT_original_example(bpy.types.Operator):
    """Access original object and do something with it"""
    bl_label = "DEG Access Original Object"
    bl_idname = "object.original_example"

    def check_object_selected(self, object_eval):
        # Selection depends on a context and is only valid for original objects. This mean
        # to request the original object from the known evaluated one.
        #
        # NOTE: All ID types have an `original` field.
        obj = object_eval.original
        return obj.select_get()

    def execute(self, context):
        # NOTE: It seems redundant to iterate over original objects to request evaluated c
        # just to get original back. But we want to keep example as short as possible, but
        # world there are cases when evaluated object is coming from a more meaningful sou
        depsgraph = context.evaluated_depsgraph_get()
        for obj in context.editable_objects:
            object_eval = obj.evaluated_get(depsgraph)
            if self.check_object_selected(object_eval):
                self.report({'INFO'}, f"Object is selected: {object_eval.name}")
        return {'FINISHED'}


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


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


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

Dependency graph: Iterate over all object instances

Sometimes it is needed to know all the instances with their matrices (for example, when writing an exporter or a custom render engine). This example shows how to access all objects and instances in the scene.

```
import bpy


class OBJECT_OT_object_instances(bpy.types.Operator):
    """Access original object and do something with it"""
    bl_label = "DEG Iterate Object Instances"
    bl_idname = "object.object_instances"

    def execute(self, context):
        depsgraph = context.evaluated_depsgraph_get()
        for object_instance in depsgraph.object_instances:
            # This is an object which is being instanced.
            obj = object_instance.object
            # `is_instance` denotes whether the object is coming from instances (as an opp
            # being an emitting object. )
            if not object_instance.is_instance:
                print(f"Object {obj.name} at {object_instance.matrix_world}")
            else:
                # Instanced will additionally have fields like uv, random_id and others wh
                # specific for instances. See Python API for DepsgraphObjectInstance for c
                print(f"Instance of {obj.name} at {object_instance.matrix_world}")
        return {'FINISHED'}


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


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


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

Dependency graph: Object.to_mesh()

Function to get a mesh from any object with geometry. It is typically used by exporters, render engines and tools that need to access the evaluated mesh displayed in the viewport.

Object.to_mesh() is closely interacting with dependency graph: its behavior depends on whether it is used on original or evaluated object.

When is used on original object, the result mesh is calculated from the object without taking animation or modifiers into account:

- For meshes this is similar to duplicating the source mesh.
- For curves this disables own modifiers, and modifiers of objects used as bevel and taper.
- For meta-balls this produces an empty mesh since polygonization is done as a modifier evaluation.

When is used on evaluated object all modifiers are taken into account.

Note

The result mesh is owned by the object. It can be freed by calling `to_mesh_clear()` .

Note

The result mesh must be treated as temporary, and cannot be referenced from objects in the main database. If the mesh intended to be used in a persistent manner use `new_from_object()` instead.

Note

If object does not have geometry (i.e. camera) the functions returns None.

```
import bpy


class OBJECT_OT_object_to_mesh(bpy.types.Operator):
    """Convert selected object to mesh and show number of vertices"""
    bl_label = "DEG Object to Mesh"
    bl_idname = "object.object_to_mesh"

    def execute(self, context):
        # Access input original object.
        obj = context.object
        if obj is None:
            self.report({'INFO'}, "No active mesh object to convert to mesh")
            return {'CANCELLED'}
        # Avoid annoying None checks later on.
        if obj.type not in {'MESH', 'CURVE', 'SURFACE', 'FONT', 'META'}:
            self.report({'INFO'}, "Object cannot be converted to mesh")
            return {'CANCELLED'}
        depsgraph = context.evaluated_depsgraph_get()
        # Invoke to_mesh() for original object.
        mesh_from_orig = obj.to_mesh()
        self.report({'INFO'}, f"{len(mesh_from_orig.vertices)} in new mesh without modifie
        # Remove temporary mesh.
        obj.to_mesh_clear()
        # Invoke to_mesh() for evaluated object.
        object_eval = obj.evaluated_get(depsgraph)
        mesh_from_eval = object_eval.to_mesh()
        self.report({'INFO'}, f"{len(mesh_from_eval.vertices)} in new mesh with modifiers.
        # Remove temporary mesh.
        object_eval.to_mesh_clear()
        return {'FINISHED'}


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


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


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

Dependency graph: bpy.data.meshes.new_from_object()

Function to copy a new mesh from any object with geometry. The mesh is added to the main database and can be referenced by objects. Typically used by tools that create new objects or apply modifiers.

by tools that create new objects or apply modifiers.

When is used on original object, the result mesh is calculated from the object without taking animation or modifiers into account:

- For meshes this is similar to duplicating the source mesh.
- For curves this disables own modifiers, and modifiers of objects used as bevel and taper.
- For meta-balls this produces an empty mesh since polygonization is done as a modifier evaluation.

When is used on evaluated object all modifiers are taken into account.

All the references (such as materials) are re-mapped to original. This ensures validity and consistency of the main database.

> Note
>
> If object does not have geometry (i.e. camera) the functions returns None.

```
import bpy


class OBJECT_OT_mesh_from_object(bpy.types.Operator):
    """Convert selected object to mesh and show number of vertices"""
    bl_label = "DEG Mesh From Object"
    bl_idname = "object.mesh_from_object"

    def execute(self, context):
        # Access input original object.
        obj = context.object
        if obj is None:
            self.report({'INFO'}, "No active mesh object to convert to mesh")
            return {'CANCELLED'}
        # Avoid annoying None checks later on.
        if obj.type not in {'MESH', 'CURVE', 'SURFACE', 'FONT', 'META'}:
            self.report({'INFO'}, "Object cannot be converted to mesh")
            return {'CANCELLED'}
        depsgraph = context.evaluated_depsgraph_get()
        object_eval = obj.evaluated_get(depsgraph)
        mesh_from_eval = bpy.data.meshes.new_from_object(object_eval)
        self.report({'INFO'}, f"{len(mesh_from_eval.vertices)} in new mesh, and is ready f
        return {'FINISHED'}


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


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


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

Dependency graph: Simple exporter

This example is a combination of all previous ones, and shows how to write a simple exporter script.

```
import bpy
```

```
class OBJECT_OT_simple_exporter(bpy.types.Operator):
    """Simple (fake) exporter of selected objects"""
    bl_label = "DEG Export Selected"
    bl_idname = "object.simple_exporter"

    apply_modifiers: bpy.props.BoolProperty(name="Apply Modifiers")

    def execute(self, context):
        depsgraph = context.evaluated_depsgraph_get()
        for object_instance in depsgraph.object_instances:
            if not self.is_object_instance_from_selected(object_instance):
                # We only export selected objects
                continue
            # NOTE: This will create a mesh for every instance, which is not ideal at all.
            # reality destination format will support some sort of instancing mechanism, s
            # code here will simply say "instance this object at object_instance.matrix_wo
            mesh = self.create_mesh_for_object_instance(object_instance)
            if mesh is None:
                # Happens for non-geometry objects.
                continue
            print(f"Exporting mesh with {len(mesh.vertices)} vertices "
                  f"at {object_instance.matrix_world}")

            self.clear_mesh_for_object_instance(object_instance)

        return {'FINISHED'}

    def is_object_instance_from_selected(self, object_instance):
        # For instanced objects we check selection of their instancer (more accurately: ch
        # selection status of the original object corresponding to the instancer).
        if object_instance.parent:
            return object_instance.parent.original.select_get()
        # For non-instanced objects we check selection state of the original object.
        return object_instance.object.original.select_get()

    def create_mesh_for_object_instance(self, object_instance):
        if self.apply_modifiers:
            return object_instance.object.to_mesh()
        else:
            return object_instance.object.original.to_mesh()

    def clear_mesh_for_object_instance(self, object_instance):
        if self.apply_modifiers:
            return object_instance.object.to_mesh_clear()
        else:
            return object_instance.object.original.to_mesh_clear()


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


def unregister():
```

```
def unregister():
    bpy.utils.unregister_class(OBJECT_OT_simple_exporter)


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

Dependency graph: Object.to_curve()

Function to get a curve from text and curve objects. It is typically used by exporters, render engines, and tools that need to access the curve representing the object.

The function takes the evaluated dependency graph as a required parameter and optionally a boolean apply_modifiers which defaults to false. If apply_modifiers is true and the object is a curve object, the spline deform modifiers are applied on the control points. Note that constructive modifiers ar modifiers that are not spline-enabled will not be applied. So modifiers like Array will not be applied and deform modifiers that have Apply On Spline disabled will not be applied.

If the object is a text object. The text will be converted into a 3D curve and returned. Modifiers are never applied on text objects and apply_modifiers w be ignored. If the object is neither a curve nor a text object, an error will be reported.

Note

The resulting curve is owned by the object. It can be freed by calling `to_curve_clear()` .

Note

The resulting curve must be treated as temporary, and cannot be referenced from objects in the main database.

```
import bpy


class OBJECT_OT_object_to_curve(bpy.types.Operator):
    """Convert selected object to curve and show number of splines"""
    bl_label = "DEG Object to Curve"
    bl_idname = "object.object_to_curve"

    def execute(self, context):
        # Access input original object.
        obj = context.object
        if obj is None:
            self.report({'INFO'}, "No active object to convert to curve")
            return {'CANCELLED'}
        if obj.type not in {'CURVE', 'FONT'}:
            self.report({'INFO'}, "Object cannot be converted to curve")
            return {'CANCELLED'}
        depsgraph = context.evaluated_depsgraph_get()
        # Invoke to_curve() without applying modifiers.
        curve_without_modifiers = obj.to_curve(depsgraph)
        self.report({'INFO'}, f"{len(curve_without_modifiers.splines)} splines in a new cu
        # Remove temporary curve.
        obj.to_curve_clear()
        # Invoke to_curve() with applying modifiers.
        curve_with_modifiers = obj.to_curve(depsgraph, apply_modifiers=True)
        self.report({'INFO'}, f"{len(curve_with_modifiers.splines)} splines in new curve w
        # Remove temporary curve.
        obj.to_curve_clear()
        return {'FINISHED'}
        return {'FINISHED'}


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


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


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

base class — `bpy_struct`

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

ids

All evaluated data-blocks

TYPE:

`bpy_prop_collection` of `ID`, (readonly)

mode

Evaluation mode

- `VIEWPORT` Viewport – Viewport non-rendered mode.
- `RENDER` Render – Render.

TYPE:

enum in [‘VIEWPORT’, ‘RENDER’], default ‘VIEWPORT’, (readonly)

object_instances

All object instances to display or render (Warning: Only use this as an iterator, never as a sequence, and do not keep any references to its items)

TYPE:

`bpy_prop_collection` of `DepsgraphObjectInstance`, (readonly)

objects

Evaluated objects in the dependency graph

TYPE:

`bpy_prop_collection` of `Object`, (readonly)

scene

Original scene dependency graph is built for

TYPE:

`Scene`, (readonly)

scene_eval

Scene at its evaluated state

TYPE:

`Scene`, (readonly)

updates

updates

Updates to data-blocks

TYPE:

`bpy_prop_collection` of `DepsgraphUpdate`, (readonly)

view_layer

Original view layer dependency graph is built for

TYPE:

`ViewLayer`, (readonly)

view_layer_eval

View layer at its evaluated state

TYPE:

`ViewLayer`, (readonly)

debug_relations_graphviz(*, filepath='')

debug_relations_graphviz

PARAMETERS:

filepath (*string, (optional, never None)*) – File Name, Optional output path for the graphviz debug file

RETURNS:

Dot Graph, Graph in dot format

RETURN TYPE:

string

debug_stats_gnuplot(filepath, output_filepath)

debug_stats_gnuplot

PARAMETERS:

- **filepath** (*string, (never None)*) – File Name, Output path for the gnuplot debug file
- **output_filepath** (*string, (never None)*) – Output File Name, File name where gnuplot script will save the result

debug_tag_update()

debug_tag_update

debug_stats()

Report the number of elements in the Dependency Graph

RETURNS:

result

RETURN TYPE:

string, (never None)

update()

Re-evaluate any modified data-blocks, for example for animation or modifiers. This invalidates all references to evaluated data-blocks from th dependency graph.

id_eval_get(id)

id_eval_get

PARAMETERS:

id (`ID`) – Original ID to get evaluated complementary part for

RETURNS:

Evaluated ID for the given original one

RETURN TYPE:

`ID`

id_type_updated(id_type)

id_type_updated

PARAMETERS:

id_type (enum in Id Type Items) – ID Type

RETURNS:

Updated, True if any datablock with this type was added, updated or removed

RETURN TYPE:

boolean

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:

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`

References

- BlendDataMeshes.new_from_object
- Context.evaluated_depsgraph_get
- ID.evaluated_get
- Object.calc_matrix_camera
- Object.camera_fit_coords
- Object.closest_point_on_mesh
- Object.crazyspace_eval
- Object.ray_cast
- Object.to_curve
- Object.to_mesh
- RenderEngine.bake
- RenderEngine.draw
- RenderEngine.render
- RenderEngine.update
- RenderEngine.view_draw
- RenderEngine.view_update
- Scene.ray_cast
- ShaderNodeTexPointDensity.cache_point_density
- ShaderNodeTexPointDensity.calc_point_density
- ShaderNodeTexPointDensity.calc_point_density_minmax
- ViewLayer.depsgraph

Previous
DecimateModifier(Modifier)

N
DepsgraphObjectInstance(bpy_stru

Report issue on this page