Customizing the Datasmith Import Process

Describes how to import Datasmith and CAD files using Blueprint or Python, and how to change the way your scene is transformed into Unreal Assets and Actors.

The goal of the Datasmith import process is to bring a set of 3D objects that you've set up in a content creation tool smoothly into the Unreal Editor. To do this, it automatically translates objects such as meshes, lights, cameras and surface materials into their Unreal Engine equivalents while doing its best to respect the intent of your design, and it automatically populates a Level with instances of these Assets for you. For details, see Datasmith Overview.

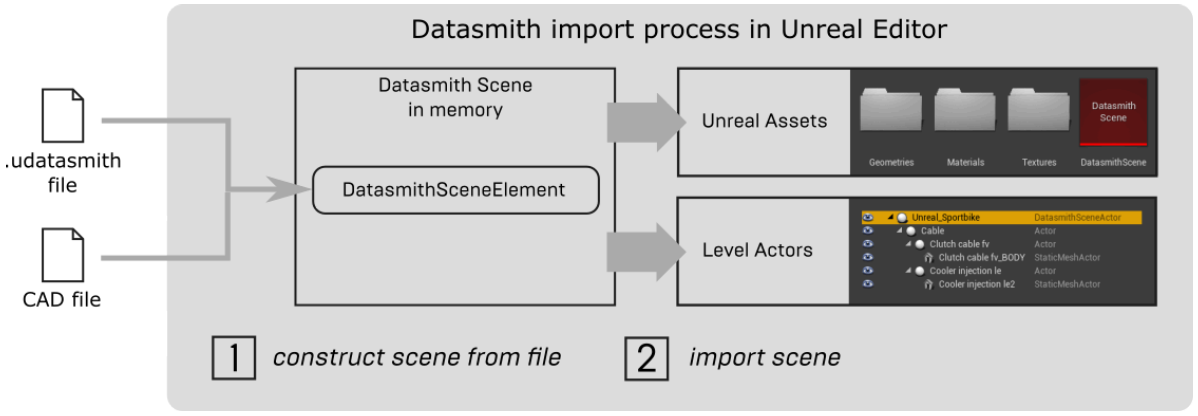
Sometimes, however, you may want to get inside the import process and change the way it translates your original scene into Unreal Engine Assets, or to change what it does with those Assets. For example, if there are parts of your original scene that you know you won't need in your Unreal Engine Project, you might want to filter those objects out before creating your Assets. This reduces the number of different pieces of content you have to deal with in the Unreal Editor, and can speed up the import for large scenes.

This page shows how you can use Blueprint or Python scripting to take control of the Datasmith import process.

Understanding the Import Process

When you import a scene using one of the Datasmith importers in the Unreal Editor Toolbar, as described in Importing Datasmith Content into Unreal Engine, the importer internally performs a two-step process to take your data from a `.udatasmith,` CAD, or other source file on disk and turn it into Assets and Actors in the Unreal Editor:

1. The importer reads the contents of the file into an in-memory data structure called a Datasmith Scene. This contains a representation of the 3D objects in the scene, their relationships, and all the properties of those objects that Datasmith was able to extract from your original scene.
2. When the Datasmith Scene is ready in memory, a second stage of the import process finalizes the scene elements into Unreal Engine Assets in the Content Browser. When the Datasmith Scene Asset is ready, the import process spawns it in the current Level. This in turn spawns all of its children: Actors, Static Mesh Actors, Lights, Cameras, and so on.


Datasmith import process in Unreal Editor
.udatasmith file
CAD file
Datasmith Scene in memory
DatasmithSceneElement
Unreal Assets
Datasmith Scene
Geometries
Materials
Textures
DatasmithScene
Level Actors
Unreal_Sportbike DatasmithSceneActor
Cable Actor
Clutch cable fv Actor
Clutch cable fv_BODY StaticMeshActor
Cooler injection le Actor
Cooler injection le2 StaticMeshActor
1 construct scene from file
2 import scene

Options for Customizing the Import Process

If you use a Blueprint or Python script to launch the Datasmith import process, you can de-construct the process above and carry out each of the two stages separately. This allows you to insert your own processing after you construct the Datasmith Scene in memory, but before you finalize that scene into Assets and Actors.

The overall process is the same for both Blueprint and Python:

1. Construct a new in-memory Datasmith Scene representation from the location of a `.udatasmith`, CAD, or other supported file type on disk.
2. Do any additional scene modifications that you want to affect the way your scene is transformed into Unreal Assets.

   One way you can make it easier to identify what objects you need to change is to take advantage of metadata about the objects in your scene. For details on how to access metadata in the Datasmith Scene, see Using Datasmith Metadata.

3. Set up options for the import process. These options are essentially equivalent to the ones you set up in the Unreal Editor UI when you use the **Import Datasmith** button to start the import. For example, you set the path where the importer should place your imported assets within your project, what types of objects it should create from the Datasmith Scene, tessellation settings for parametric CAD formats, and so on.
4. Complete the import process to finalize your Datasmith Scene into Unreal Assets.
5. When you no longer need the Datasmith Scene that you constructed above, you should destroy the scene in order to clear the memory resources it uses.
6. Now that the import process is done, your new Assets are available in the Content Browser, and your new Actors are available in the current Level (if you requested them to be added in your import options). If you want to do additional *post*-processing on the generated Unreal Assets, like creating collision data or LODs automatically, this is a great time to do that. See also Scripting and Automating the Editor.

Customizing the import process is very likely to have an effect on the re-import process outlined in Datasmith Reimport Workflow. For example, if you use a script to remove elements such as meshes or lights from the Datasmith Scene before you complete the import process, then you re-import the Datasmith Scene Asset, your pre-processing script is bypassed during the re-import. The result is that the objects you originally filtered out from the scene are detected as newly added, and are added to your Project or Level.

For now, we recommend doing most modifications *after* import, using the tools and techniques described under Scripting and Automating the Editor. Modify the Datasmith Scene during import only if you have a particular need that you can't fulfill by modifying Assets and Actors after you finalize the import, such as preventing the creation of certain Assets.

Before You Start

- Make sure that you have the **Datasmith Importer** plugin enabled for your Project.
- For background information on how to use Blueprint and Python scripts within the Editor, see the pages under Scripting and Automating the Editor.

Examples

The following examples show how to use Blueprint and Python to customize the process of importing a *.udatasmith* file and a CAD file into the Unreal Editor.

Choose your implementation method

Blueprints Python

If you want to use Blueprint to customize the import process, you'll mainly be using nodes from the **Editor Scripting > Datasmith**, **Datasmith > Scene** and **Datasmith > Element** categories.

To reach the nodes you need, you'll need to use an Editor Utility Widget, and Editor Utility Blueprint, or make your Blueprint class derive from an Editor-only base class, such as **EditorUtilityActor**. The examples below show Blueprint graphs triggered from buttons in an Editor Utility Widget.

Your destination folder must start with `/Game/`.

Importing a .udatasmith File

In this example, the **Get Options** node requests the **DatasmithImportOptions** class, which contains the basic import settings inside its **Base Options** variable.

Copy code

Click the top right of the image to copy the node graph.

Importing a CAD File

In this example, the first **Get Options** call requests the same **DatasmithImportOptions** class as the previous example. The second call requests the **DatasmithCommonTessellationOptions** class, which contains the tessellation settings inside its **Options** variable. The third call requests the **DatasmithCADImportOptions** class, which contains additional CAD-specific settings that are generally intended for use in special circumstances.

Copy code

Click the top right of the image to copy the node graph.

If you want to use Python to customize the import process, your main starting point will be the `unreal.DatasmithSceneElement` class. This class offers you all the functions you need to construct a scene from a file, work with the elements in that scene (through the functions defined in the base `unreal.DatasmithSceneElementBase` class), and finalize the import.

Importing a .udatasmith File

```

import unreal

ds_file_on_disk = "C:\\scenes\\building.udatasmith"
ds_scene_in_memory =
unreal.DatasmithSceneElement.construct_datasmith_scene_from_file(ds_file_on_disk)

if ds_scene_in_memory is None:
```

```
print "Scene loading failed."
quit()

# Modify the data in the scene to filter out or combine elements...

# Remove any mesh whose name includes a certain keyword.
remove_keyword = "exterior" # we'll remove any actors with this string in their names.
meshes_to_skip = set([]) # we'll use this set to temporarily store the meshes we don't need.

# Remove from the scene any mesh actors whose names match the string set above.
for mesh_actor in ds_scene_in_memory.get_all_mesh_actors():
actor_label = mesh_actor.get_label()
if remove_keyword in actor_label:
print("removing actor named: " + actor_label)
# add this actor's mesh asset to the list of meshes to skip
mesh = mesh_actor.get_mesh_element()
meshes_to_skip.add(mesh)
ds_scene_in_memory.remove_mesh_actor(mesh_actor)

# Remove all the meshes we don't need to import.
for mesh in meshes_to_skip:
mesh_name = mesh.get_element_name()
print("removing mesh named " + mesh_name)
ds_scene_in_memory.remove_mesh(mesh)

# Set import options.
import_options = ds_scene_in_memory.get_options(unreal.DatasmithImportOptions)
import_options.base_options.scene_handling = unreal.DatasmithImportScene.NEW_LEVEL

# Finalize the process by creating assets and actors.

# Your destination folder must start with /Game/
```

```
result = ds_scene_in_memory.import_scene("/Game/MyStudioScene")

if not result.import_succeed:
print "Importing failed."
quit()

# Clean up the Datasmith Scene.
ds_scene_in_memory.destroy_scene()
print "Custom import process complete!"


```

Copy full snippet

Importing a CAD File

```

import unreal

# Construct the Datasmith CAD Scene from a file on disk.
# Your destination folder must start with /Game/
ds_file_on_disk = "C:\\designs\\Clutch assembly.SLDASM"
ds_scene_in_memory =
unreal.DatasmithSceneElement.construct_datasmith_scene_from_file(ds_file_on_disk)

if ds_scene_in_memory is None:
print "Scene loading failed."
quit()

# Modify the data in the scene to filter out or combine elements.
remove_keyword = "_BODY" # we'll remove any actors with this string in their names.
```

```
meshes_to_skip = set([]) # we'll use this set to temporarily store the meshes we don't need.

# Remove from the scene any mesh actors whose names match the string set above.
for mesh_actor in ds_scene_in_memory.get_all_mesh_actors():
actor_label = mesh_actor.get_label()
if remove_keyword in actor_label:
print("removing actor named: " + actor_label)
# add this actor's mesh asset to the list of meshes to skip
mesh = mesh_actor.get_mesh_element()
meshes_to_skip.add(mesh)
ds_scene_in_memory.remove_mesh_actor(mesh_actor)

# Remove all the meshes we don't need to import.
for mesh in meshes_to_skip:
mesh_name = mesh.get_element_name()
print("removing mesh named " + mesh_name)
ds_scene_in_memory.remove_mesh(mesh)

# Set import options.

# Main import options:
import_options = ds_scene_in_memory.get_options(unreal.DatasmithImportOptions)
import_options.base_options.scene_handling = unreal.DatasmithImportScene.NEW_LEVEL

# CAD-only surface tessellation options:
tessellation_options = ds_scene_in_memory.get_options(unreal.DatasmithCommonTessellationOptions)
tessellation_options.options.chord_tolerance = 0.1
tessellation_options.options.max_edge_length = 0
tessellation_options.options.normal_tolerance = 30
tessellation_options.options.stitching_technique =
unreal.DatasmithCADStitchingTechnique.STITCHING_SEW

```

```
# Additional CAD-only options:
cad_import_options = ds_scene_in_memory.get_options(unreal.DatasmithCADImportOptions)
cad_import_options.uv_generation = unreal.CADUVGeneration.KEEP
cad_import_options.num_threads = 8

# Finalize the process by creating assets and actors.
ds_scene_in_memory.import_scene("/Game/MyCADScene")

# Clean up the Datasmith Scene.
ds_scene_in_memory.destroy_scene()
print "Custom import process complete!"


```

Copy full snippet

Accessing Import Options

When you work with different types of Datasmith source files, the only difference in the API is that different types of files give you different options for controlling the import operation.

- For *.udatasmith* files, the first example above shows how you can access a **DatasmithImportOptions** object that provides access to destination Asset and path names, what types of data you want to import, lightmap resolution settings, and so on. This works for all types of files you can import with Datasmith.
- For CAD files, the second example shows how you can also access a **DatasmithCommonTessellationOptions** object that controls the settings used in the tessellation process, and a **DatasmithCADImportOptions** object that offers access to other internal debug settings that you may find useful.
- Other file types supported by Datasmith encapsulate their import settings in other classes, such as **DatasmithC4DImportOptions** for Cinema 4D.

Apart from accessing these import options, the rest of the scripts for working with the Datasmith Scene and the elements inside the scene are exactly the same.

For more detail on these import option classes and the settings they offer, see their entries in the Python API Reference or the Blueprint API Reference.

When you call `DatasmithSceneElement.get_options()` in Python or use the **Get Options** node in Blueprint, if you specify a class of Datasmith options that doesn't apply to the type of source file you've constructed, you'll get a null return. In Python, you can test for a `None` return. In Blueprint, you can pass the returned value to the **Utilities > IsValid** node to check that it's a valid instance of the class you requested.

As an alternative to using `DatasmithSceneElement.get_options()` in Python or **Datasmith > Scene > Get Options** in Blueprint, you can use `DatasmithSceneElement.get_all_options()` or **Datasmith > Scene > Get All Options** to retrieve *all* classes of options for the Datasmith scene that you've constructed in a single map. The key value of each entry in the map is a class of options owned by the Datasmith Scene Element that you constructed from your file, and the value of the entry is the instance of that class owned by the Datasmith Scene Element. If you don't know what kind of file your script will be handling, and therefore what kinds of option classes the `DatasmithSceneElement` should contain, you can use this approach to retrieve them all and iterate through them.

About the Datasmith Scene

In order to explore what you're able to do with the Datasmith scene during the pre-import phase, it helps to know a little about how the scene is constructed.

Scene Contents

A Datasmith Scene is mostly a container for different types of *elements*. Each of these elements represents either an *Asset* that will be created in your Content Browser after import, or an *Actor* that will be spawned in your Level with its own particular transform in 3D space.

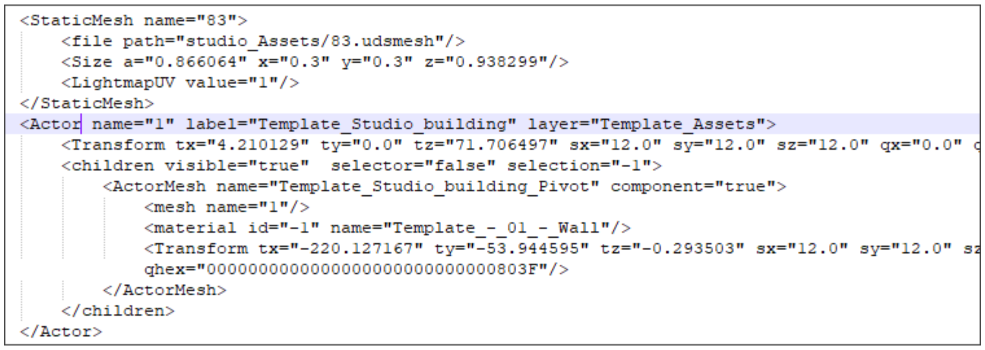
The main asset element types include:

- Meshes: Each mesh element represents a block of 3D geometry. When you complete the import, each mesh element ends up as a separate Static Mesh object under the Geometry folder. Each mesh element has a number of material slots, each of which is associated by name with material elements.
- Materials: Each material element represents a distinct type of surface that is needed for your geometries. When you complete the import, each material element ends up as a separate Material object under the Materials folder.
- Textures: Each texture element represents a single 2D image that is used by at least one of your Materials. When you complete the import, each texture element ends up as a separate Texture object under the Textures folder.

The main actor element types include:

- Mesh actors: Each mesh actor element represents an instance of a mesh geometry in your Level. When you complete the import, each mesh actor element ends up as a Static Mesh Actor in your World Outliner.
- Light actors: Each light actor element represents a light emitter in your scene. When you complete the import, each light actor element ends up in your Level as an instance of a base Unreal light type, such as a Point Light or Spot Light, or as a custom Datasmith Actor that simulates an Area light. You can get and set a number of properties on these lights, such as their intensity, color, IES profile texture files, and so on.
- Camera actors: Each camera represents a point of view that is set up in your source scene. When you complete the import, each camera actor element ends up in your Level as a CineCameraActor. You can get and set some basic properties on these camera actors, such as their aspect ratios.

The data contained in the Datasmith Scene in memory is very similar to what you see in a *.udatasmith* file if you open it up. If you're using 3ds Max or Sketchup, you can open any exported *.udatasmith* file to get an idea of how the Datasmith Scene object is constructed:



```xml
<StaticMesh name="83">
    <file path="studio_Assets/83.udsmesh"/>
    <Size a="0.866064" x="0.3" y="0.3" z="0.938299"/>
    <LightmapUV value="1"/>
</StaticMesh>
<Actor name="1" label="Template_Studio_building" layer="Template_Assets">
    <Transform tx="4.210129" ty="0.0" tz="71.706497" sx="12.0" sy="12.0" sz="12.0" qx="0.0"
    <children visible="true"  selector="false" selection="-1">
        <ActorMesh name="Template_Studio_building_Pivot" component="true">
            <mesh name="1"/>
            <material id="-1" name="Template_-_01_-_Wall"/>
            <Transform tx="-220.127167" ty="-53.944595" tz="-0.293503" sx="12.0" sy="12.0" sz
            qhex="0000000000000000000000000000803F"/>
        </ActorMesh>
    </children>
</Actor>
```

Working with the Datasmith Scene

You'll mainly interact with the Datasmith Scene in order to retrieve lists of the elements outlined above. To do this in Python, you'll use the functions defined in the `DatasmithSceneElementBase` class that your `DatasmithSceneElement` derives from. In Blueprint, you'll use the **Datasmith > Scene** nodes as shown in the examples above.

Once you have a list of elements, you can iterate through the list to retrieve a particular element. Then, use the Python API for that element (such as `DatasmithMeshActorElement`), or the **Datasmith > Element** nodes in Blueprint, to get and set information about that particular element. If your element is an Actor type, you can also get its child actor elements, which lets you browse downward through the scene hierarchy.

If you want to remove existing elements from the scene (as shown in the examples above), or add new elements, you can also do this through the functions exposed in the same places. For example, you could re-shuffle the hierarchy of Actors by removing them and re-adding them under different parents, using functions exposed by your `DatasmithSceneElement` class in Python, or nodes from the **Datasmith > Scene** category in Blueprint.