

SD473709

More Tips, Tricks, and the Future of the Forge Model Derivative Service

Kevin Vandecar – Developer Advocate
Autodesk

Denis Grigor – Developer Advocate
Autodesk

Learning Objectives

- Learn how to compare the data sets coming from the Model Derivative service.
- Learn details about the Revit generateMasterViews flags to get room and space information.
- See a preview of coming new features in Model Derivative service.
- Learn about how and when to use the new SVF2 format.

Description

Last year we brought you “Tips, Tricks, and the Future for Forge Model Derivative Services.” We’re bringing even more goodies to this year’s presentation. We will include more details around the Revit flag to generate rooms and spaces for viewing. We will cover how you can use the model derivative data to do model version comparison. Finally, again we will preview some of the coming features in Model Derivative Service, including the improved SVF2 Forge Viewer format.

Speaker(s)



Kevin Vandecar is a Forge developer advocate and also the manager for the Media & Entertainment and Manufacturing Autodesk Developer Network Workgroups. His specialty is 3ds Max software customization and programming areas, including the new Forge Design Automation for 3ds Max service.



Denis Grigor likes to know how everything works under the hood, and is not afraid of low-level stuff like bits, buffers, pointers, stack, heap, threads, shaders and of course Math. He is interested in 3D for Web, from raw WebGL to libraries and frameworks with different levels of abstractions, as well as how virtual entities from 3D world can be linked to things from real world. In the end, "For an artificial mind, all reality is virtual" [The Animatrix]. To achieve my goals, he prefers to speak C/C++ mostly with modern dialect, Go, Python, and I have to speak Javascript/Node.js.

Contents

Re-Introduction to Model Derivative Service	3
Compare Datasets coming from Model Derivative	3
Changing property of a component in a model	6
Restructuring the model	7
Adding a new component to the model.....	8
Removing and adding component to the model.....	10
Transforming a component (translate, scale, rotate).....	11
Improvements to SVF output from AEC input types	17
Using the generateMasterViews attribute	17
September AEC Updates	18
Deprecated IFC switchLoader.....	19
New IFC -> SVF Translation Options	19
Updated Navisworks Translation Engine.....	20
Model Derivative specific Webhooks.....	20
3ds Max Physical Material support for Model Derivative SVF format.....	23
New SVF2 format.....	29
What is SVF2?	29
Enhanced Model Derivative Properties API	32
References.....	33

Re-Introduction to Model Derivative Service

As the title of this class indicates, this is a continuation of last year's "Tips, Tricks, and the Future for Forge Model Derivative Services" Autodesk University course. You can find that course and related content here: <https://www.autodesk.com/autodesk-university/class/Tips-Tricks-and-Future-Forge-Model-Derivative-Services-2019>. That course covers some of the basics of Model Derivative service and some introductory Tips and Tricks that were compiled from several sources, including the Autodesk Forge blog articles and documentation.

*Tip: The <https://forge.autodesk.com> portal has also gone through some improvements in the past year. First, there is now a specific search for blog articles. You can search by keyword, and also other tagged content. For example, to find latest news and techniques on Model Derivative service, choose it from the "API and Services" dropdown menu. Also note there is a site-wide search (find it in upper right corner) that will search for all related content on the Forge portal. Try it! Enter **generateMasterViews** to see this nice new aspect of search finding all related content including the documentation, stackoverflow questions and blog articles.*

To remind you, the Model Derivative service is fundamentally a translation service, with a priority to represent and share your designs from different formats in a cloud-based scenario. The SVF format provides additional capability that when used with the Forge Viewer enables 3D interactive viewing in cloud and mobile environments. Additionally, when using the SVF format there is also meta data coming from the source authoring systems. This functionality allows you as a software developer to build very data rich and specific workflows for your customers and users. It's main services are provided through REST based APIs, and the Forge Viewer is using a JavaScript library making it easy to consume in many environments.

The Model Derivative API is well documented, including the formats that are supported: https://forge.autodesk.com/en/docs/model-derivative/v2/developers_guide/overview/

Tip: One improvement in the past year, is to include translator specific change notes in the change history. You can find those notes here: https://forge.autodesk.com/en/docs/model-derivative/v2/change_history/changelog/

Compare Datasets coming from Model Derivative

Knowing the difference between versions of the same model or able to identify the common components of different models is a quite common need and with Forge there are several ways to address it.

In the Forge Viewer context, there is the **Autodesk.DiffTool** viewer extension that helps to visualize the changes by highlighting *added*, *removed*, and *changed* components. This extension is used by the Forge Team product to illustrate the changes between versions of the same model or assembly, but it can be easily integrated in your own instance of the Forge

Viewer, and was very well described in blogpost here:

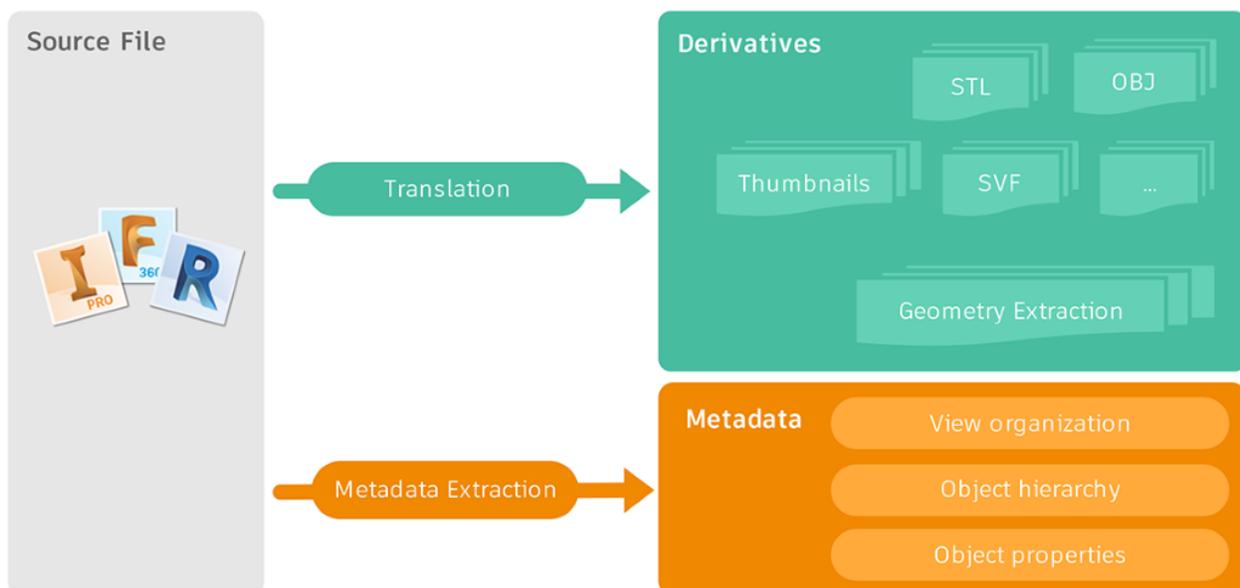
<https://forge.autodesk.com/blog/difference-3d-models-autodeskdifftool-extension>

The strength of this approach is that you are automatically presented with the visual aspect of the changes, while the main drawback is that it is an interactive UI based process running in you browser. This approach is quite difficult to automate for batch processes.

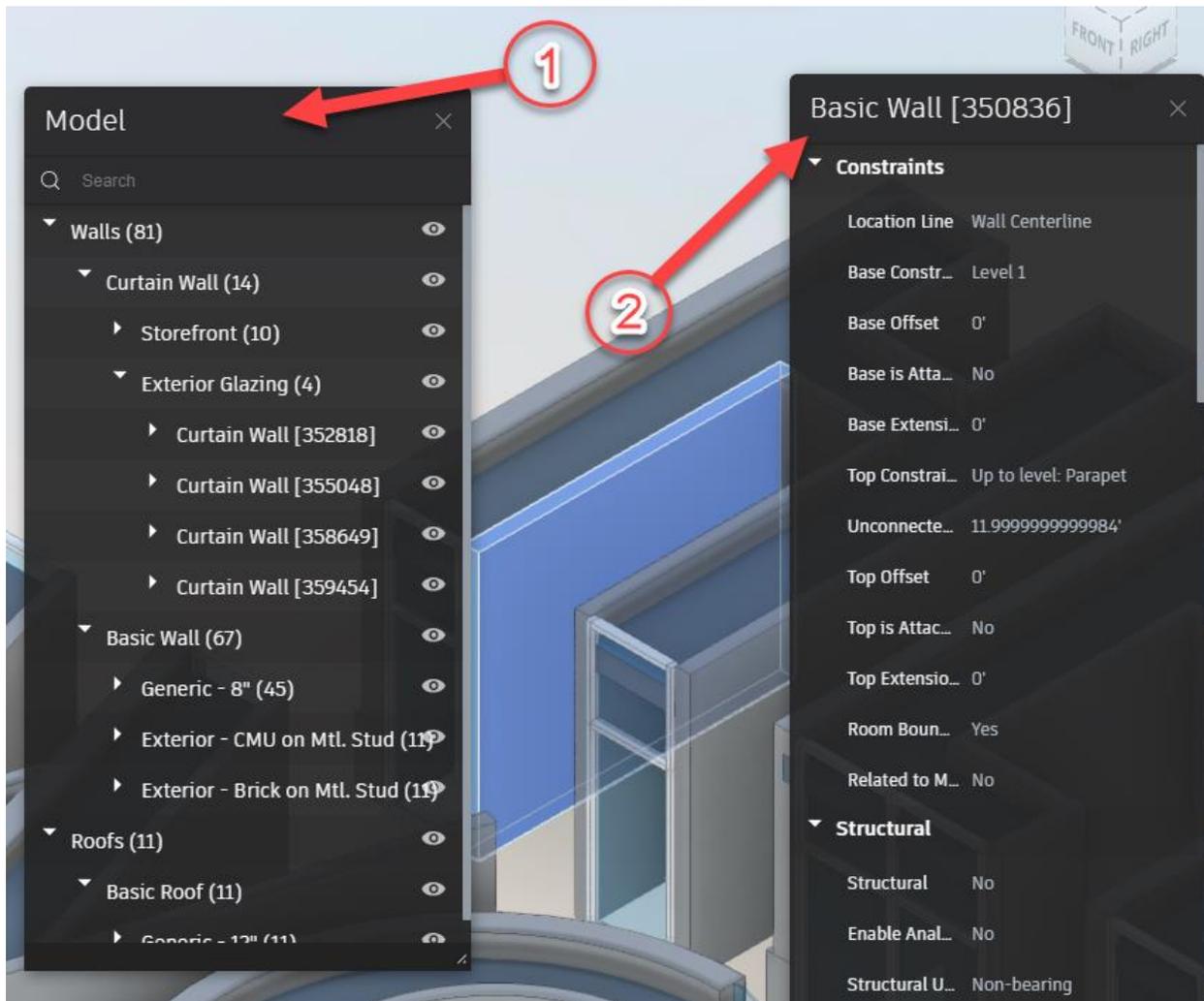
Another approach is to use the Design Automation engine to load the model and using the power of the engine (be it Revit, Inventor, AutoCAD or 3ds Max) process input scenes and identify the changes. For some engines and for some workflows this could be done with native tools/plugins/scripts (e.g. [Comparing AutoCAD Drawings with Forge Design Automation](#)), for more complex changes, it might require some plugin development.

The main strength of this approach is that once it is set, it is easy to integrate into a pipeline and excellent for batch processing, while the main drawback is that it might require some resources for plugin development, depending on the complexity of the use case and also the format of the processed files. If it is a mix of Revit and Inventor files, it will require development of separate plugins for those two engines and the format of the files you can work with is limited to what formats those engines accept and what information you are able to analyze after it was imported.

Now we arrive at the approach using the Model Derivative service. Of course this approach is based on the output of translations, so may not ideal in cases, but has proven itself in a few customer applications already. As a reminder, the Model Derivative Service accepts [over 70 formats](#) currently and it provides a common output when using the SVF format. This format includes **Metadata Extraction** and is the key to this approach.



To better understand what kind of data is extracted, it is enough to visualize the translated file in the Forge Viewer:



There are two important parts generated for any accepted format:

1. Model tree (object hierarchy) - showing the hierarchical relation of the components.
2. Object properties – generated for each component or even a group of components.

The Model Derivative service, after the translation, makes available all this metadata in the form of a json file (model tree and properties, respectively):

```

{
  "data": {
    "type": "objects",
    "objects": [
      {
        "objectId": 1,
        "name": "Model",
        "objects": [
          {
            "objectId": 2028,
            "name": "Floors",
            "objects": [
              {
                "objectId": 2029,
                "name": "Floor",
                "objects": [
                  {
                    "objectId": 2,
                    "name": "WoodMat",
                    "externalId": "[\\\"eyJlbmRpdMkiOiiIiwic2VnbWudCI6IjA1A...",
                    "properties": {
                      "Area": "6300.000 mm^2",
                      "Component Appearance": "Steel - Satin (1)",
                      "Component Name": "WoodMat",
                      "Mass": "41.213 g",
                      "Material": "Steel",
                      "Name": "WoodMat",
                      "Volume": "5250.000 mm^3",
                      "File Properties": {
                        "Part Number": "WoodMat",
                        "Title": "WoodMat"
                      }
                    }
                  }
                ]
              }
            ]
          }
        ]
      }
    ]
  }
}

```

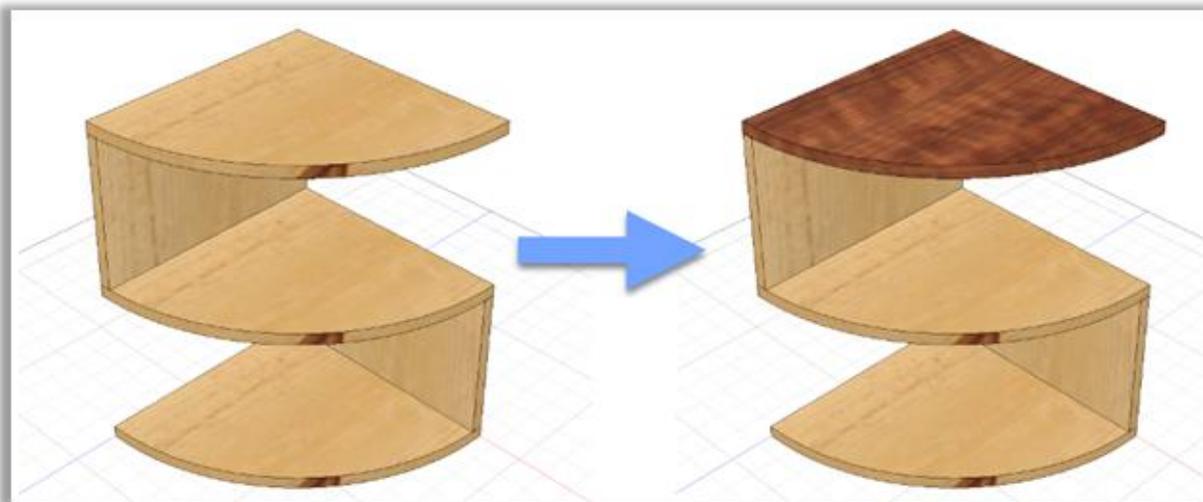
The nice bonus for BIM360 users is that usually all models available in BIM360 are already translated and you only need to use the Model Derivative API to retrieve it.

In our context, the useful part is that this metadata is made available through REST calls in form of a json and the information can be comparable and the difference easy to be identified.

However, to better understand the limitations of this approach it would be useful to know what kind of changes/differences we can capture through this method.

Changing property of a component in a model

To illustrate what we can capture when we change some properties in the model, let's look at a model of a corner shelf unit where we change the physical material of the top part:



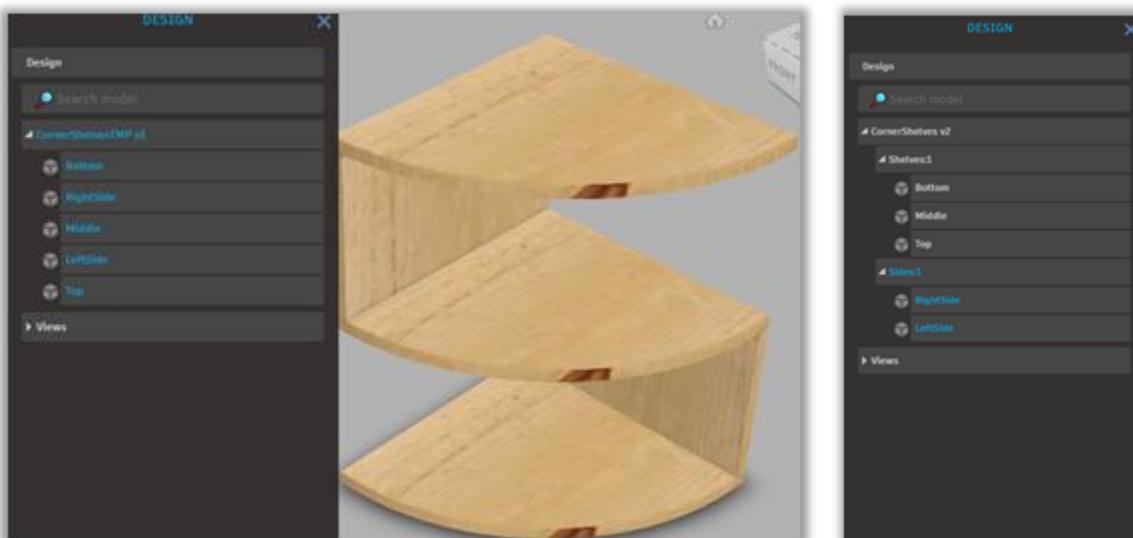
This is a Fusion model where the change of material is reflected in properties like density and consequently the mass of the part.

By checking the model hierarchy file, obviously nothing changed, but when checking the object properties file, we can easily spot the change:

<pre>{ "objectId": 6, "name": "Top", "externalId": "[\\"eyJhc3NldCI6Ij "properties": { "Appearance": "Pine", "Area": "76770.000 mm^2", "Density": "0.001 g / mm^3", "Mass": "197.515 g", "Material": "Pine", "Name": "Top", "Volume": "346400.000 mm^3" } }</pre>	73 74 75 76 77 78 79 80 81 82 83 84 85	73 74 75 76 77 78 79 80 81 82 83 84 85	<pre>{ "objectId": 6, "name": "Top", "externalId": "[\\"eyJhc3NldCI6I "properties": { "Appearance": "Cherry", "Area": "76770.000 mm^2", "Density": "0.001 g / mm^3", "Mass": "188.420 g", "Material": "Cherry", "Name": "Top", "Volume": "346400.000 mm^3" } }</pre>
---	--	--	--

Restructuring the model

In case we refactor a model/scene, the model remains the same, but the hierarchical relations between components changes.



If we retrieve the model tree for both files and compare them, we notice that due to regrouping the ids of some components changed/shifted. The “Bottom” part once had id = 3, but now since in the second model it was moved into the “Shelves” component, it definitely changed, and this is important to know and identify if we want to check later if the properties of the “Bottom” part are changed or not.

```

{
  "objectId": 2,
  "name": "CornerShelvesTMP",
  "objects": [
    {
      "objectId": 3,
      "name": "Bottom"
    },
    {
      "objectId": 4,
      "name": "RightSide"
    },
  ],
}

```

```

{
  "objectId": 2,
  "name": "CornerShelvesTMP",
  "objects": [
    {
      "objectId": 3,
      "name": "Shelves:1",
      "objects": [
        {
          "objectId": 4,
          "name": "Bottom"
        },
      ],
    },
  ],
}

```

Your logic for this concept should not rely on id parts remaining the same in different versions of the same file, regardless if in the Forge Viewer or in backend metadata processing. For this purpose it is better to rely on the *external_id* (providing it is supported and available; not all formats provide it). In our example (see below), if we investigate the object property file, we will see that the objectId changed, while the properties remain intact. But looking at the array of externalids, external id “survived” the change.

This approach is tricky, because it expects that the application that created this file relies on external id mechanism and that it also encodes the external id history for the parts/components. If it is available and matches, we can be sure that we are talking about the same part in two different models.

```

{
  "objectId": 3,
  "name": "Bottom",
  "externalId": ["eyJhc3NldCI6IjZmOTFmODdlLWEzNDgtI
    "eyJhc3NldCI6IjZmOTFmODdlLWEzNDgtI
  ],
  "properties": {
    "Appearance": "Pine",
    "Area": "76770.000 mm^2",
    "Density": "0.001 g / mm^3",
    "Mass": "197.515 g",
    "Material": "Pine",
    "Name": "Bottom",
    "Volume": "346400.000 mm^3"
  }
},

```

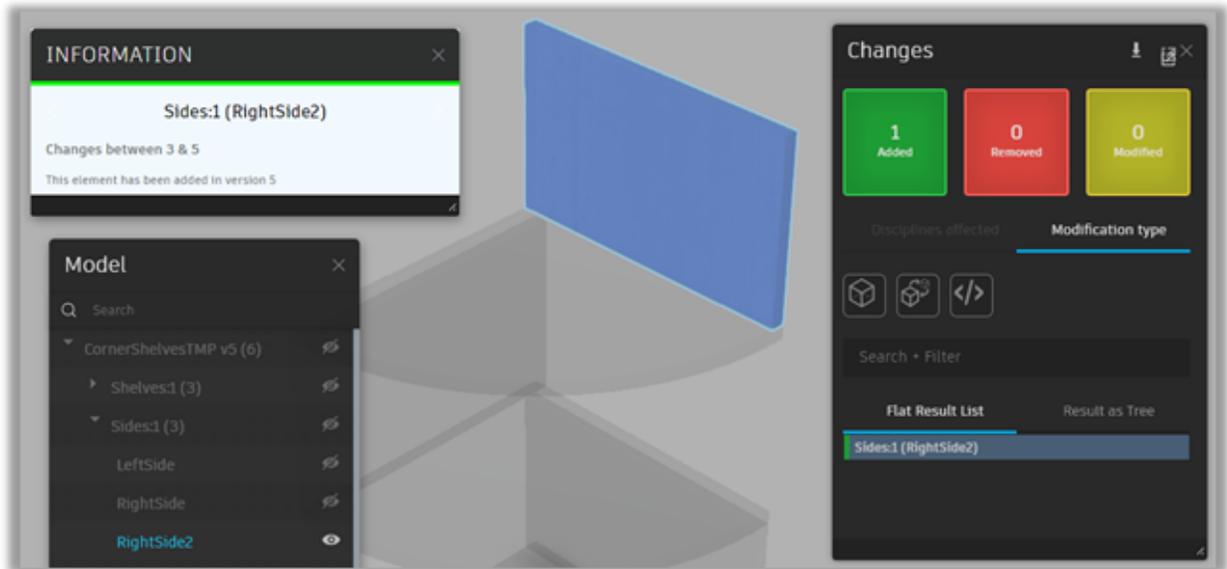
```

{
  "objectId": 4,
  "name": "Bottom",
  "externalId": ["eyJhc3NldCI6IjZmOTFmODdlLWEzNDgtI
    "eyJhc3NldCI6IjZmOTFmODdlLWEzNDgtI
  ],
  "properties": {
    "Appearance": "Pine",
    "Area": "76770.000 mm^2",
    "Density": "0.001 g / mm^3",
    "Mass": "197.515 g",
    "Material": "Pine",
    "Name": "Bottom",
    "Volume": "346400.000 mm^3"
  }
},

```

Adding a new component to the model

To continue with the same sample file, let's add another part, in this case just a side panel. Visually it will be identified as:



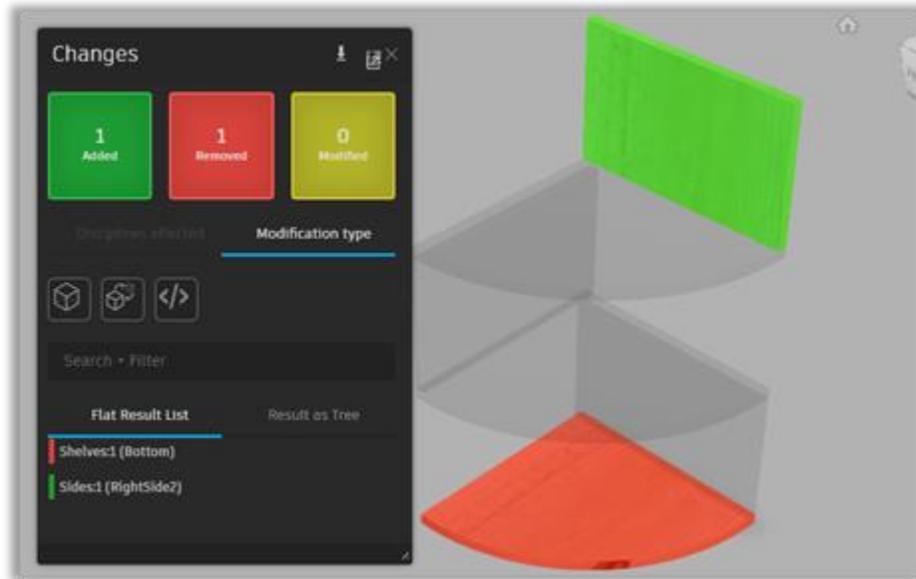
If we think again in terms of the model tree, to identify that this part was added, for us it would be enough to look at the model tree:

<pre>{ "objectId": 7, "name": "Sides:1", "objects": [{ "objectId": 8, "name": "LeftSide" }, { "objectId": 9, "name": "RightSide" }] }</pre>	31 31 32 32 33 33 34 34 35 35 36 36 37 37 38 38 39 39 40 40 41 41	<pre>{ "objectId": 7, "name": "Sides:1", "objects": [{ "objectId": 8, "name": "LeftSide" }, { "objectId": 9, "name": "RightSide" }, { "objectId": 10, "name": "RightSide2" }] }</pre>
<pre>] }]</pre>	42 42 43 43 44 44 45 45 46 46 47 47	<pre>], { "objectId": 10, "name": "RightSide2" } }</pre>

It looks simple enough to spot the newly added element and for simple cases it is fine. However, once you start to have a mix of added and removed components, things get complicated.

Removing and adding component to the model

When removing or do a combination of adding and removing a part like in the following example:



```

{
  data: {
    objects: [
      0: {
        objects: [
          0: {
            objects: [
              0: {
                objectid: 4,
                name: "Bottom"
              },
              1: {
                objectid: 5
              },
              2: {
                objectid: 9,
                name: "RightSide2"
              }
            ]
          },
          1: {
            objectid: 6,
            objects: [
              0: {
                objectid: 7
              },
              1: {
                objectid: 8
              }
            ]
          }
        ]
      }
    ]
  }
}

```

we can expect that the model tree will be somehow scrambled, but even in this situation, filtering the common things and concentrating on changes we can easily spot them.

Note: As it was mentioned before, the objectid is not a reliable identifier as it might “move” in case components are added/removed/reordered. For this very reason it is mandatory to find

another unique identifier. Sometimes the name could be enough, but better to reinforce it by having something like a pair of names like in the below example “Sides:1+LeftSide” created by concatenating “parentName” and “partName”.

```

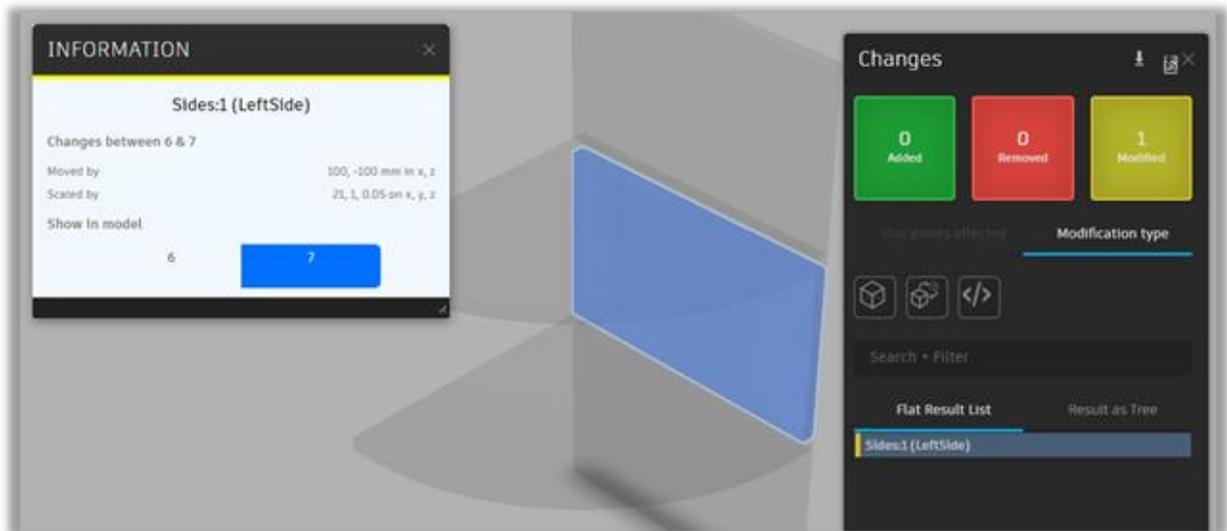
"objectId": 7,
"name": "Sides:1",
"objects": [
  {
    "objectId": 8,
    "name": "LeftSide"
  }
]

```

In this case even if the id of “LeftSide” changes, it can easily be identified that it was not removed, or the hierarchy relocated. Nevertheless, as mentioned before the bulletproof identifier remains the external id, but in case the application does not support it, this workaround can help solve the problem:

Transforming a component (translate, scale, rotate)

When we do not change component hierarchy, and change no properties, but do a slight transformation, be it moving, or rotating a part, like in the example below where a side panel was rotated and moved to align with other side panels, it is expected that the model tree json will show no difference with the previous model.



Almost same picture we see when inspecting the properties file. The only thing that changed is the external id of the root node:

```

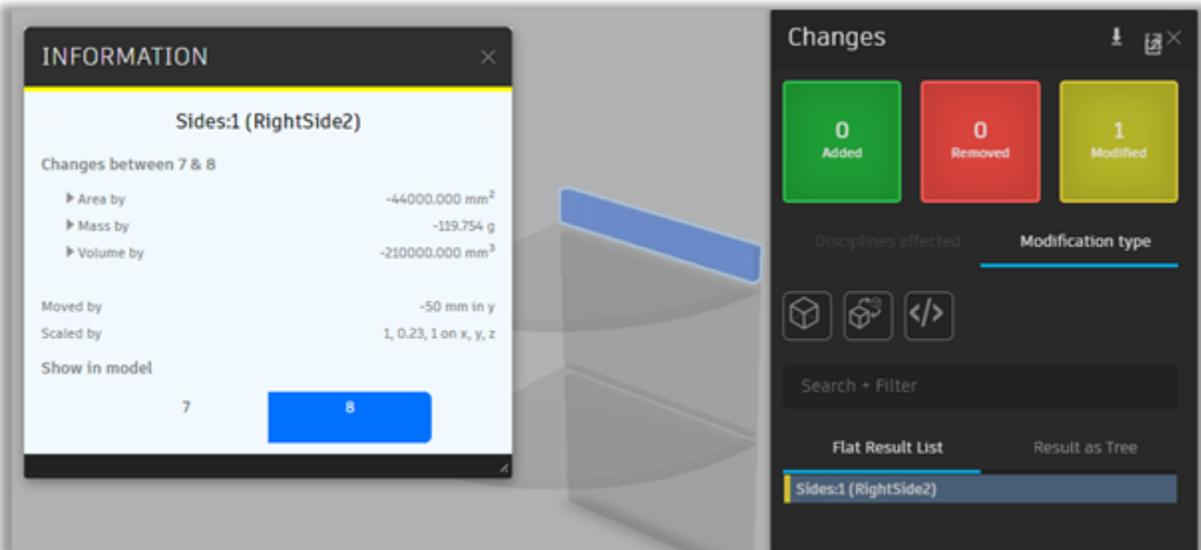
{
  "data": {
    "type": "properties",
    "collection": [
      {
        "objectId": 1,
        "name": "CornerShelvesTMP",
        "externalId": "\\Q29ybmVyu2h1bHZ1c1RyYW5zZm9yYV9BQ55mM2Q",
        "properties": {
          "Name": "CornerShelvesTMP"
        }
      }
    ]
  },
}

```

This might not look too useful, but from this we know that:

- No new component was added/removed/reordered, otherwise the model tree file would have changed.
- No properties were changed, otherwise the object property file would have changed.

As a side note, if it is a “parametric type application” (ie. Fusion 360 in this case), we also know that no part was resized, otherwise it would have reflected in change of its properties. For example, in our case, if we resize a side panel, it automatically affects the mass, area and volume property:



INFORMATION

Sides:1 (RightSide2)

Changes between 7 & 8

- Area by: -44000.000 mm²
- Mass by: -119.754 g
- Volume by: -210000.000 mm³

Moved by: -50 mm in y

Scaled by: 1, 0.23, 1 on x, y, z

Show in model

7 **8**

Changes

0 Added, 0 Removed, 1 Modified

Disciplines affected: [Icons]

Modification type: [Icons]

Search + Filter

Flat Result List, Result as Tree

Sides:1 (RightSide2)

```

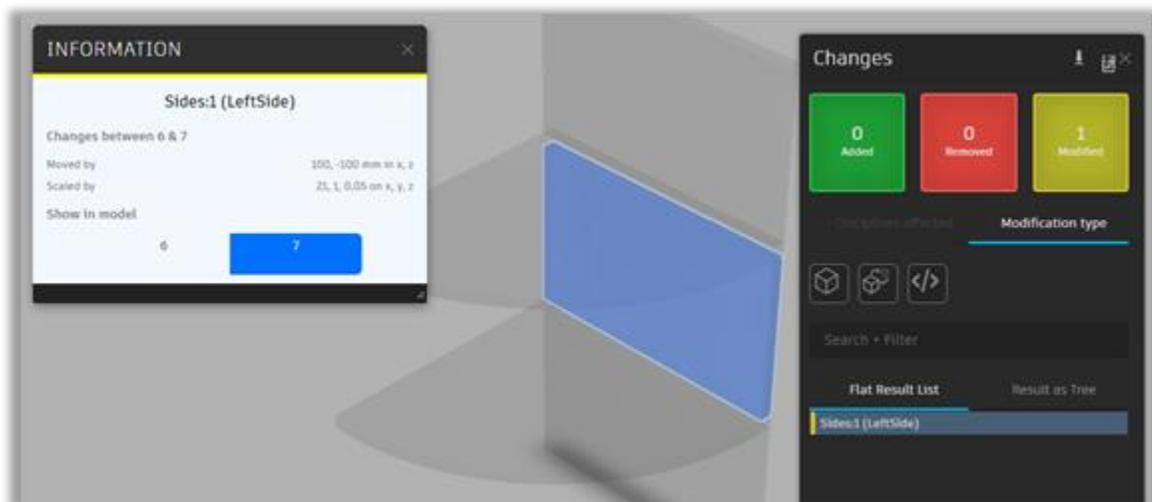
{
  "objectId": 9,
  "name": "RightSide2",
  "externalId": "\\eyJhc3NldCI6IjZi",
  "properties": {
    "Appearance": "Pine",
    "Area": "61400.000 mm^2",
    "Density": "0.001 g / mm^3",
    "Mass": "155.680 g",
    "Material": "Pine",
    "Name": "RightSide2",
    "Volume": "273000.000 mm^3"
  }
}

```

Unfortunately, with only these 2 files we cannot be sure if any components were moved or rotated, or the file was just saved with another version (because this external id is regenerated upon file change).

Nevertheless, with Model Derivative service we still can drill down and identify the if the geometry of the part was changed. It might look cumbersome and an overkill, but if we want to capture the geometry changes, we must analyse the geometry itself.

In our case if we go back to rotated and translated part:



We know from the model tree that it has not changed and by comparing the object properties files, we are sure that nothing changed from id perspective.

The Model Derivative API allows extracting the OBJ for any part, or group, providing the id of the needed parts:

POST <https://developer.api.autodesk.com/modelderivative/v2/designdata/job>

```

{
  "input": {
    "urn": "{{ENCODED_URN}}"
  },
  "output": [
    {
      "formats": [
        {
          "type": "obj",
          "advanced": {
            "modelGuid": "{{GUID}}",
            "objectIds": [7]
          }
        }
      ]
    }
  ]
}

```

By passing the id of the needed part, we can trigger another translation job and ask for the OBJ of that very part. The good thing of the OBJ format is that it is readable, and if it is readable, it means that it can be compared:

```
# WaveFront *.obj file (generated by Autodesk ATF)
g Obj.7
v 21.000000 28.000000 -8.000000
v -8.000000 28.000000 -8.000000
v 21.000000 28.000000 1.000000
v -8.000000 28.000000 1.000000
v 21.000000 15.000000 -8.000000
v 21.000000 15.000000 1.000000
v -8.000000 15.000000 -8.000000
```

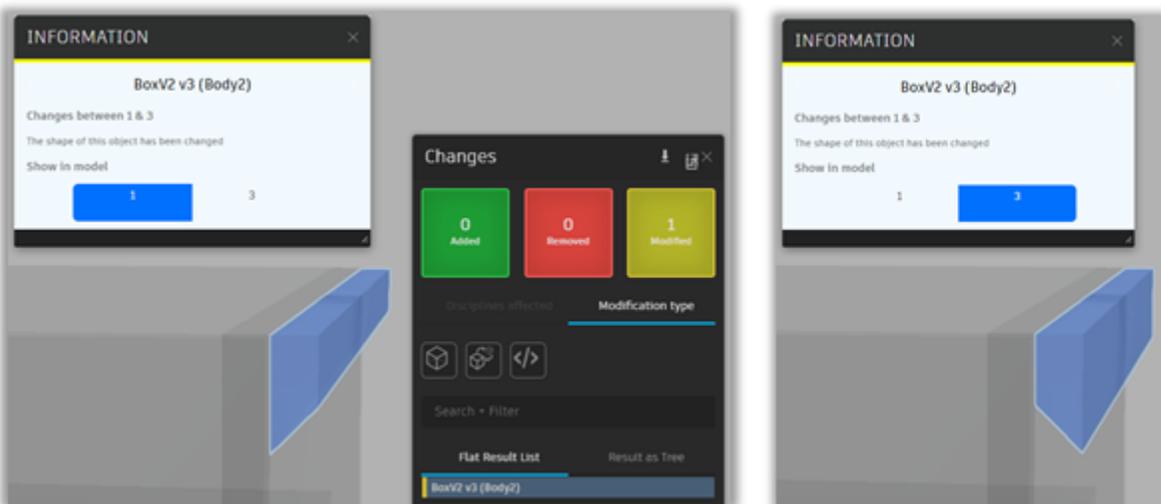
For our case, it is enough just to know if the geometry is the same or not, by comparing the ascii content of the OBJ file originating from different model – if it matches, then it is the same, if it doesn't than, most probably it was transformed.

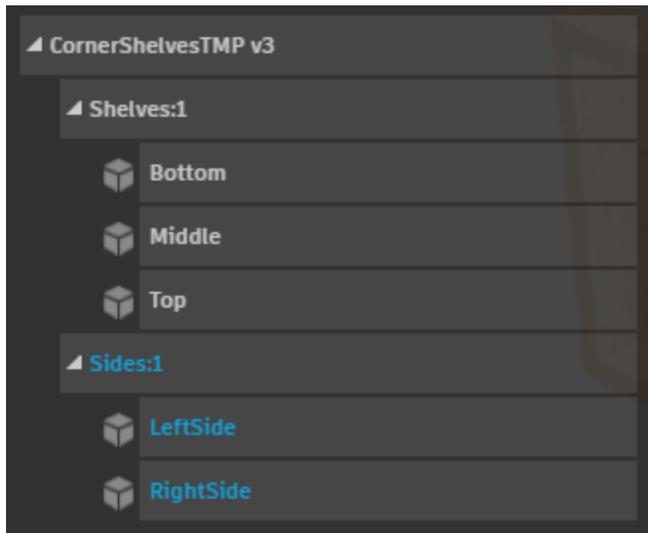
Thus, analysing the file until we find first mismatch should be enough for our purpose:

# WaveFront *.obj file (generated by Autodesk ATF)	1	2	3	4	5	6	7	8	9	10	11
g Obj.7											
v 1.000000 28.000000 21.000000	>>	5	<<	v 21.000000 28.000000 -0.000000							
v 1.000000 28.000000 0.000000		6		v -0.000000 28.000000 -0.000000							
v -0.000000 28.000000 21.000000		7		v 21.000000 28.000000 1.000000							
v -0.000000 28.000000 0.000000		8		v -0.000000 28.000000 1.000000							
v 1.000000 15.000000 21.000000		9		v 21.000000 15.000000 -0.000000							
v -0.000000 15.000000 21.000000		10		v 21.000000 15.000000 1.000000							
v 1.000000 15.000000 0.000000		11		v -0.000000 15.000000 -0.000000							

But if there is a need to go further and identify the type of transformation – was it translated, rotated or both, then this is a good starting point.

Note: This approach is not limited to identify only moves and rotation, in fact, since we analyse the geometry at the vertex level, it can easily identify the part that has been tapered, twisted, bent, mirrored etc. Once the shape is changed, it will be spotted.





Tip: For optimization purposes, instead of checking each id individually, a divide and conquer approach could be employed. In this tree relation, the Sides:1 node has no geometry (the non-leaf's nodes have no geometry), but it is a parent of the LeftSide and RightSide nodes with geometry. The important thing to know is that if we ask for an OBJ and provide a parent node, then the geometry of all its children will be provided (merged) in the resulted OBJ. Thus, by comparing the geometry at branch level, it is enough to identify if the geometry of the leafs are the same or not. With just a true/false analysis, it should be easy to identify the id of the components with changed geometry. At the same time, this could be applied at the root level. Asking the OBJ of the root will provide information not only on vertices, but also on id of the part to which the list of vertices belong. It involves some parsing, but this information is there to be retrieved.

For our case, by isolating just the changes in the OBJ for id=1 (usually the root), and seeing “g Obj.7” we can identify that it belongs to part with id 7.

f 93//54 50//54 91//54	440	440	f 93//54 50//54 91//54
	441	441	
g Obj.7	442	442	g Obj.7
	443	443	
v 1.000000 28.000000 21.000000	>> 444	444 <<	v 21.000000 28.000000 -0.000000
v 1.000000 28.000000 0.000000	445	445	v -0.000000 28.000000 -0.000000
v -0.000000 28.000000 21.000000	446	446	v 21.000000 28.000000 1.000000
v -0.000000 28.000000 0.000000	447	447	v -0.000000 28.000000 1.000000
v 1.000000 15.000000 21.000000	448	448	v 21.000000 15.000000 -0.000000
v -0.000000 15.000000 21.000000	449	449	v 21.000000 15.000000 1.000000
v 1.000000 15.000000 0.000000	450	450	v -0.000000 15.000000 -0.000000
v -0.000000 15.000000 0.000000	451	451	v -0.000000 15.000000 1.000000
vn 0.000000 10.000000 0.000000	452	452	vn 0.000000 10.000000 0.000000
vn -0.000000 0.000000 10.000000	>> 453	453 <<	vn 10.000000 0.000000 0.000000
vn 0.000000 -10.000000 0.000000	454	454	vn 0.000000 -10.000000 0.000000
vn 0.000000 0.000000 -10.000000	>> 455	455 <<	vn -10.000000 0.000000 0.000000
vn -10.000000 0.000000 -0.000000	>> 456	456 <<	vn 0.000000 0.000000 10.000000
vn 10.000000 0.000000 0.000000	457	457	vn 0.000000 0.000000 -10.000000
f 97//55 98//55 99//55	458	458 <<	f 97//55 98//55 99//55
f 99//55 98//55 100//55	459	459	f 99//55 98//55 100//55
f 101//56 97//56 102//56	460	460	f 101//56 97//56 102//56
f 102//56 97//56 99//56	461	461	f 102//56 97//56 99//56

In summary, based on use cases discussed above we can conclude that Model Derivative service provides the necessary information, to be able to determine the most common changes. It excels when it comes to identifying the changes in metadata and to some extent, it provides the necessary data to spot the geometry changes. However, when it comes to comparing models where components were rearranged, added, and removed – it requires some work to properly identify the common parts and added parts, and this is where a visual review may prove more useful (for example using the Forge Viewer extension)

The Model Derivative approach as compared to the Forge Viewer approach with its **Autodesk.DiffTool** extension, offers almost the same level of information, but in this case it runs “headless” (without UI), which makes it ideal for automation and batch processing of large number of files.

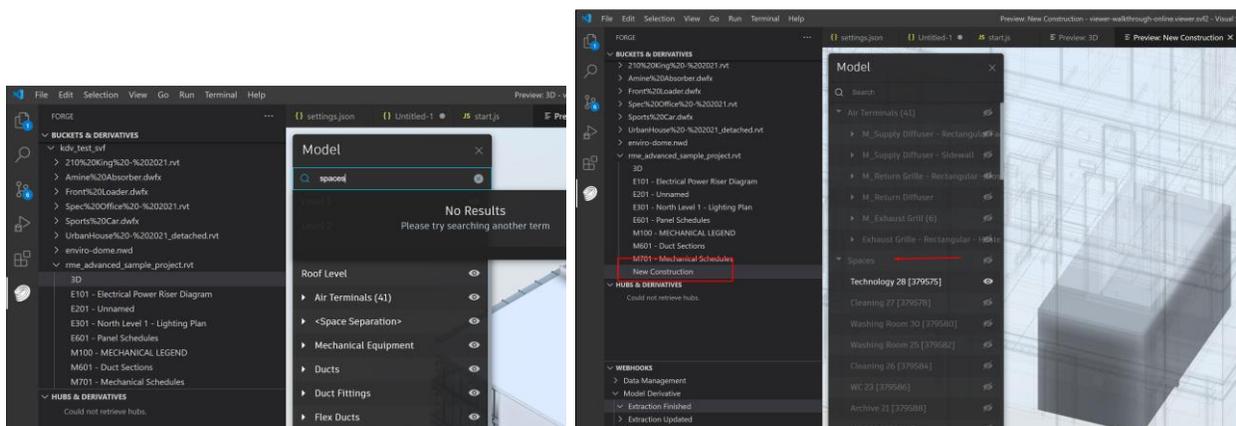
The Model Derivative approach as compared to the Design Automation approach is less powerful, because we are not working with the native files and API functionality (much richer in the context of that specific Design Automation Engine).

Hopefully the options are now clearer, and you can see that Model Derivative supports many formats and requires less resources (time and effort) to work with the metadata. These ideas make Model Derivative a valid option for automating model comparisons.

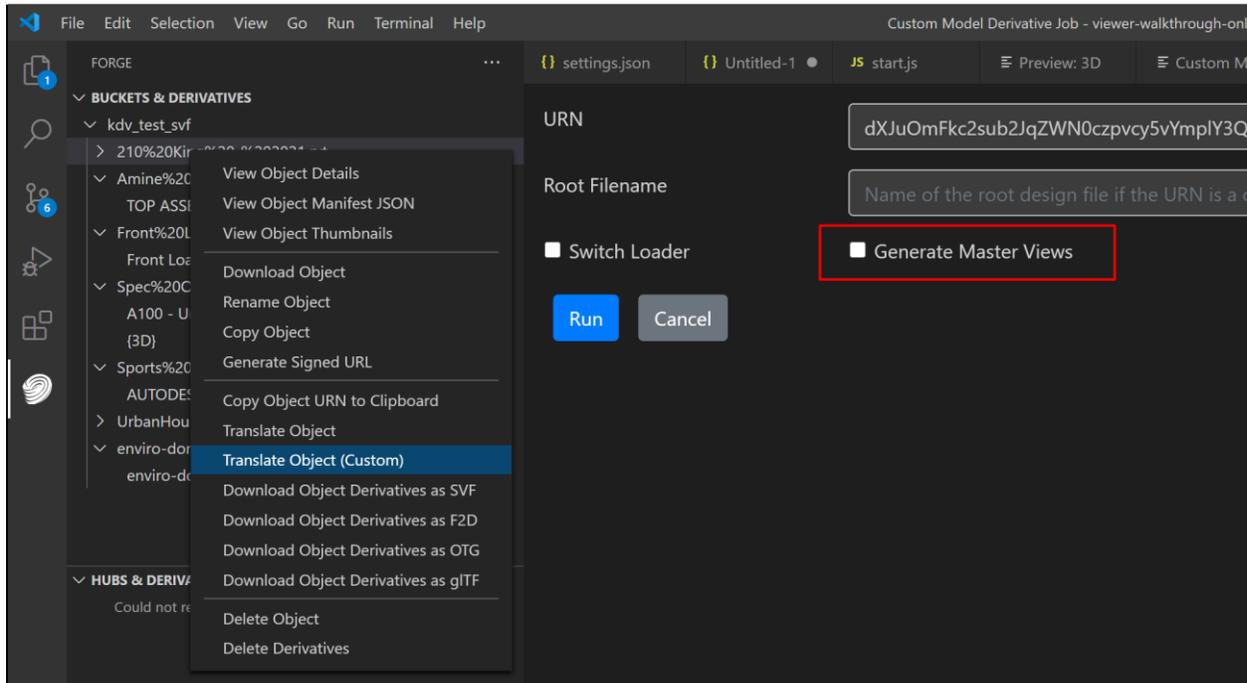
Improvements to SVF output from AEC input types

Using the generateMasterViews attribute

Last year we announced a new attribute to allow Revit models to also translate the room/space and zone information called **generateMasterViews**. A master view is a Viewable that is generated for each phase of the Revit model. It contains all the elements, including room elements that are present in the source model for that phase. After translation is complete, you will have extra viewables that are named from the phasing information. Within these viewables you will find the extra room/space and zone details. Here is a screen shot of the *rme_advanced_sample_project.rvt* file showing the differences of the **generateMasterViews** attribute. On the left is default behavior (without **generateMasterViews**) and on the right is using the **generateMasterViews** attribute. Notice there is a new Viewable called “New Construction” that now contains the “spaces” category of objects.



During the past year we have added support for **generateMasterViews** in the Visual Studio Code Extension (see here for introduction: <https://forge.autodesk.com/blog/forge-visual-studio-code>) and also fixed a few problems with the functionality. The original blog I made is here: (<https://forge.autodesk.com/blog/new-rvt-svf-model-derivative-parameter-generates-additional-content-including-rooms-and-spaces>). Here is a screenshot of its uses in the VS Code extension:



The `generateMasterViews` is now documented here: <https://forge.autodesk.com/en/docs/model-derivative/v2/reference/http/job-POST/> (See “Case 2: Input file type is Revit” in the “advanced” section).

We also have a tutorial that is specific to this workflow: <https://forge.autodesk.com/en/docs/model-derivative/v2/tutorials/prep-roominfo4viewer/>

One caveat we found with this attribute is that not all spaces were present in a small number of cases. This problem also happened to occur in the BIM 360 Docs automatic translation. The problem was that if the exporter could not make sense of an oddly or malformed shaped space it would stop translating all further spaces. This was reported and has since been fixed. However, note that the “erroneous” space is skipped and will still be missing. If needed for graphics in the Viewer, it is recommended to reshape the space object into a more regular shape. However, remember then that the shape itself is not accurate in size and volume. In the future we are looking at options to handle this better. One option being considered is to list the object ids for the skipped items into the manifest so you can determine which ones are missing and then make decision on how to resolve. Keep an eye on the Forge blog for improvements here.

September AEC Updates

The next sections describe updates that went into production on 09/07/2020. These affect how translations are done with IFC, Navisworks, and Revit files.

Deprecated IFC switchLoader

Last year we also introduced the switchLoader attribute that was for IFC files. This basically switched between using the Revit or Navisworks engine to translate an IFC file. See the next section about IFC->SVF for details on all the changes.

New IFC -> SVF Translation Options

There are now several new attributes for the “advanced” section to control IFC output to SVF format. This is fully documented here: <https://forge.autodesk.com/en/docs/model-derivative/v2/reference/http/job-POST/> (see “Case 1: Input file type is IFC”)

Let’s take a look at these options:

conversionMethod replaces switchLoader and is clearer in name than switchLoader. switchLoader is deprecated but still supported for older workflows. When both are present conversionMethod overrides. It is recommended to update to conversionMethod as soon as possible to avoid losing switchLoader later. conversionMethod has two options:

- legacy uses the Navisworks IFC translator and is slower
- modern uses the Revit IFC translator and provides more options now.

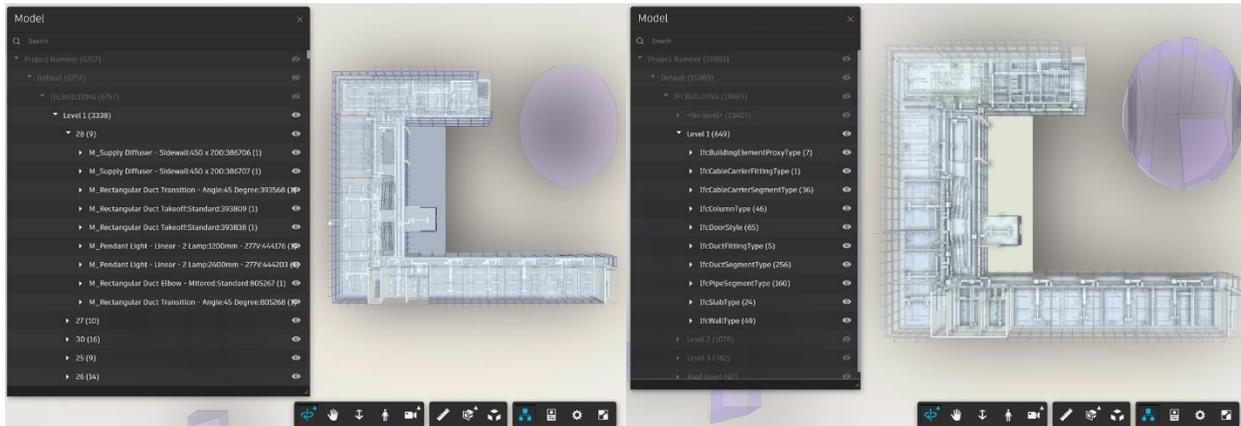
Also note that when conversionMethod is set to modern, there are several more options available to help optimize the IFC conversion for your needs.

- buildingStoreys allows you to hide, show or skip storeys (note the spelling is matching the IFC specification)
- spaces allows you to hide, show or skip spaces
- openingElements allows you to hide, show or skip openings

First of all, if you do nothing (ie. no attributes) you will get the legacy Navisworks extractor. The Navisworks extractor is faster and will provide similar results to the Revit extractor. However the Revit extractor has certain benefits. First, it honors the IFC standard more closely and accurately. It also is more modern and being improved regularly. The Revit extractor also handles “swept surfaces” better than Navisworks. Finally, using the Revit extractor is the only way to get the additional benefits of controlling the aspects of buildingStoreys, spaces, and openingElements.

How can this affect your translations? Let’s consider the Revit sample *rme_advanced_sample_project.rvt* as an IFC dataset (exported from Revit as IFC). Because raw IFC is in textual form, the same dataset is much larger (over 100MBs in this case).

The default Navisworks extractor took 172.33 seconds. The Revit IFC extractor took 340.05 seconds. After discussing with a few others, it is agreed that currently the modern (Revit) extractor generally takes on average about 2-4 times longer. But the results, especially model structure can be much better in the “modern” Revit translator.



conversionMethod = legacy (Navisworks) on left and conversionMethod = modern (Revit) on right

Updated Navisworks Translation Engine

First of all, the performance has been improved, and also eliminates many problems when converting large files when there are time-out error messages.

The render settings are also being considered now. In the past this was completely ignored and gave Forge Viewer results that looked much different than the Navisworks settings.

Additional settings have been added to handle a variety of things that improve conversion times by default, but you can now control and use at will. These settings are: materialMode, hiddenObjects, basicMaterialProperties, autodeskMaterialProperties, and timelinerProperties.

For details, see <https://forge.autodesk.com/en/docs/model-derivative/v2/reference/http/job-POST/> and scroll down to “Case 3: Input type Is Navisworks”.

Model Derivative specific Webhooks

The Webhook system came later than Model Derivative and in the prior presentation we did not discuss it, so I believe it is important to point out. Currently none of the documentation tutorials or samples are showing it and it is documented under “Webhooks” section so you may have missed it. It’s very convenient way to work with Model Derivative jobs.

For example, in the Tutorial for “Translating a Source file” we show using the manifest to check progress. You can find that example here: <https://forge.autodesk.com/en/docs/model-derivative/v2/tutorials/translate-to-obj/task3-translate-source-file/> and see the end section where they repeatedly request the manifest to see the “status” and “progress”. In fact most of the samples do this.

There are two Model Derivative webhooks which can help you to avoid this repetitive request for the manifest to get the status. For example, if all you really care about is that the job is finished, you can use the `extraction.finished`. If you also want to know a regular status, you can use `extraction.updated`.

The Webhooks documentation starts here:

https://forge.autodesk.com/en/docs/webhooks/v1/developers_guide/overview/

The specific event details:

https://forge.autodesk.com/en/docs/webhooks/v1/reference/events/model_derivative_events/

There is a tutorial here:

<https://forge.autodesk.com/en/docs/webhooks/v1/tutorials/create-a-hook-model-derivative/>

Here is some simple example code to setup the web hook (this is added to the Online Viewer App tutorial that uses Express and Axios, a library that allows you to call the REST APIs directly and clearly). The tutorial is here: <https://forge.autodesk.com/developer/learn/viewer-app/overview>.

Step 1. Setup the webhook and callback. Notice that this is using the `extraction.finished` event and sets-up the work flow scope to be `'workflow-extraction-complete'`.

```
Axios({
  method: 'POST',
  url: 'https://developer.api.autodesk.com/webhooks/v1/' +
    'systems/derivative/events/extraction.finished/hooks',
  headers: {
    'content-type': 'application/json',
    Authorization: 'Bearer ' + access_token
  },
  data: JSON.stringify({
    // This would be a real location on your sever when deployed.
    // For local server ngrok redirects to your localhost.
    'callbackUrl': 'http://8a1524b9b9cd.ngrok.io/callback/jobfinished',
    // Scope is important to identify the specific job callback you want.
    // See the job API call.
    'scope' : {
      'workflow': 'workflow-extraction-complete'
    }
  })
})
```

Step 2. Create the Callback function. Notice in previous code we indicated it would be `/callback/jobfinished`. In the following example, we can access the

```
app.post('/callback/jobfinished', jsonParser, async (req, res, next) => {
  // Best practice is to tell immediately that you got the call
  // so return the HTTP call and proceed with the business logic
  // From Augusto Goncalves
  res.status(202).end();

  // Get the hookId... You can get from the response when creating, too.
  var mdFinishedHookId = req.body.hook.hookId;
  // Was the job successful at end of the job?
  var mdFinishedStatus = req.body.payload.Payload.status;

  console.log('hit...' + urn);
  console.log('status...' + mdFinishedHookId + ' : ' + mdFinishedStatus);
});
```

```
});
```

Step 3. When using the job API we can specify the workflow scope for this job in the body misc attribute:

```
Axios({
  method: 'POST',
  url: 'https://developer.api.autodesk.com/modelderivative/v2/designdata/job',
  headers: {
    'content-type': 'application/json',
    Authorization: 'Bearer ' + access_token
  },
  data: JSON.stringify({
    'input': {
      'urn': urn
    },
    'output': {
      'formats': [
        {
          'type': 'svf',
          'views': ['2d', '3d']
        }
      ]
    },
    "misc": { // webhook callback
      "workflow": "workflow-extraction-complete"
    }
  })
})
```

Step 4. Finally, when you are done with the webhook/callback functionality, you should delete it:

```
// delete the webhook...
Axios({
  method: 'DELETE',
  url: 'https://developer.api.autodesk.com/webhooks/v1/' +
    'systems/derivative/events/extraction.finished/hooks/' + mdFinishedHookId,
  headers: {
    'content-type': 'application/json',
    Authorization: 'Bearer ' + access_token
  }
})
```

3ds Max Physical Material support for Model Derivative SVF format

3ds Max is one of Autodesk's main desktop visualization software, and is leading the online Autodesk Viewing fidelity experience. Recently with 3ds Max Model Derivative improvements, and enhancements in the Large Model Viewing technology, it has enabled the support of Physical Materials (PBR) that are translated into a format where the Viewer can render them.

To use this feature, you only need to have Physical Materials setup in your 3ds Max scene, and then send to the Model Derivative job for translation to SVF. Here are the two recent Changelog entries that bring awareness to these features:

- https://forge.autodesk.com/en/docs/model-derivative/v2/change_history/changelog/#release-date-2020-08-27
- https://forge.autodesk.com/en/docs/model-derivative/v2/change_history/changelog/#release-date-2020-06-02

See the Figure 1 screen shot showing the 3ds Max viewport render of the model. The Figure 2 image is same model that was translated using 3ds Max *.MAX format and then rendered in the Forge Viewer. The Figure 3 screen shot is showing was previously available using FBX format (where the translator does not support Physical Materials). Finally Figure 4 shows another example of a PBR model that was translated in MD service and being viewing using Forge Viewer.



FIGURE 1



Forge Viewer showing 3ds Max format with PBR materials

FIGURE 2



Forge Viewer showing 3ds Max scene exported
to FBX with materials embedded

FIGURE 3



FIGURE 4

Credits for these models come from artists publishing on <https://www.turbosquid.com/>

Digital Camera SLR Generic - by 3d_molier International
<https://www.turbosquid.com/3d-models/digital-camera-slr-generic-3d-model/934661>

Male Sci-Fi Suit - by Dyasharuku
<https://www.turbosquid.com/FullPreview/Index.cfm/ID/1121012>

As mentioned in the previous class, the easiest way to process a 3ds Max scene file with materials and xrefs, etc. is to use the Archive format, that is using a ZIP output. For example, using the Camera scene from above screen shot examples, we choose the “Archive...” option. In Figure 5 you can see this on the 3ds Max 2021 menu.

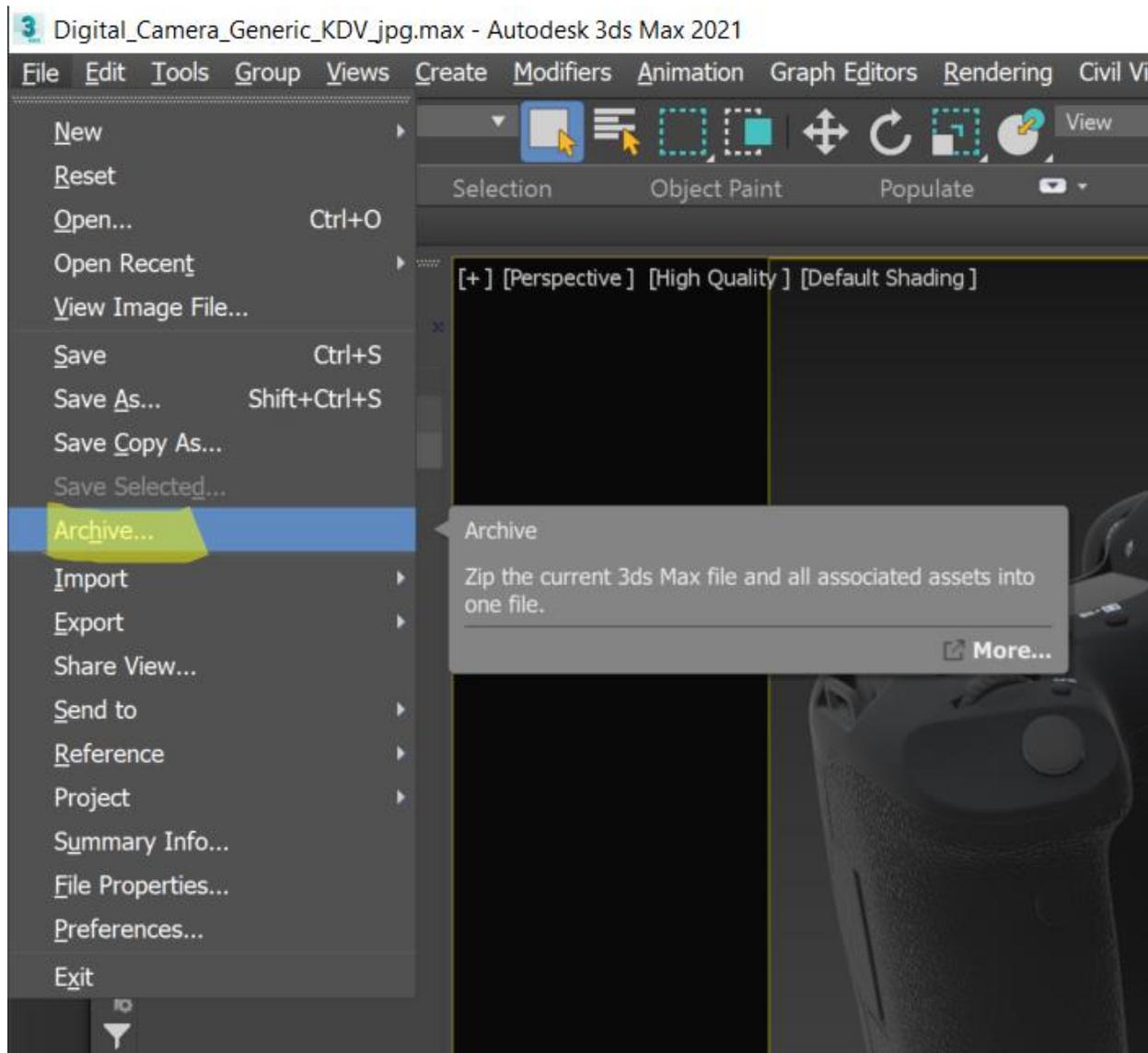


FIGURE 5

Then to send that job to the Model Derivative POST <https://developer.api.autodesk.com/modelderivative/v2/designdata/job> end point, the request body JSON payload would look like Figure 6. Pay attention to the `compressedUrn` attribute being set to true, and also the `rootFilename` attribute indicating the main scene name. If the scene is setup for Physical Materials, they should appear in the

Forge Viewer. Remember to use Forge Viewer version 7.13.1 at minimum, and to use all the current features of viewer, it is suggested to use the latest version.

```
{
  "input": {
    "urn": "{{Base64URN}}",
    "compressedUrn": true,
    "rootFilename": "camera.max"
  },
  "output": {
    "formats": [
      {
        "type": "svf",
        "views": [
          "2d",
          "3d"
        ]
      }
    ]
  }
}
```

New SVF2 format

Last year at Autodesk University 2019 we announced a new format that was code named OTG that was being researched to improve Forge Viewer performance. Several demos were made and you can find a short mention about this initiative in the Forge Keynote discussion here (<https://youtu.be/c8-AxaoHDlk?t=1147>). In this video segment, Susanna Holt discusses a strategic partner's desire to handle much larger models, and this is where the OTG format was born. Since then, Autodesk has turned the OTG format into the SVF2 format and on track to be available as a public beta before AU 2020. Check the <https://forge.autodesk.com/blog> for details.

What is SVF2?

SVF itself is already an optimized format for WebGL. The Forge Viewer, using the Three.js library is able to display very large designs in browser and mobile applications. On the user side of the Autodesk viewing technology, it is often referred to as LMV (Large Model Viewer) and is used in many other products including the BIM 360 Docs and Fusion 360 hubs for viewing. The same technology is what the Model Derivative service and Forge Viewer is using to give the same viewing experience to custom apps and workflows.

SVF2 is optimizing the SVF format to reduce loading time of models containing repetitive geometry shapes. Large models typically produced by the AEC industry (Architecture, Engineering & Construction) fall into this category. Currently, the Model Derivative service produces SVF2 derivatives by first producing SVF and further optimizing it by sharing meshes with the same viewable, and even across multiple viewables when possible. Because of this optimization, SVF2 format greatly reduces the viewable storage size. It also enables operations like incremental loading, fast switching among multiple versions, and comparing different versions in a finer granularity. Because of the extra processing, translating to SVF2 will take longer. Furthermore, all the post translation operations possible with SVF are possible with SVF2 as well. For example, extraction of metadata and extraction of geometry. What's more, the process of extracting metadata and geometry is identical for both formats.

At the moment, the SVF2 format is produced directly from the SVF formats when requested. The SVF2 generation is basically a post-process/optimization.

See below for examples of performance increase of SVF2 as compared to SVF. As you will see, in most cases the SVF2 format outperforms the SVF format and takes only a small additional amount of time. But be aware, it does take additional time, so ideally, it's benefit vs. cost should be used where a model is quite large.

Three APIs will be updated to support SVF2:

GET <https://developer.api.autodesk.com/modelderivative/v2/designdata/formats>

POST <https://developer.api.autodesk.com/modelderivative/v2/designdata/job>

GET <https://developer.api.autodesk.com/modelderivative/v2/designdata/:urn/manifest>

- or -

GET <https://developer.api.autodesk.com/modelderivative/v2/regions/eu/designdata/:urn/manifest>

For example, using the job endpoint, you can see only the format is different:

```

Axios({
  method: 'POST',
  url: 'https://developer.api.autodesk.com/modelderivative/v2/designdata/job',
  headers: {
    'content-type': 'application/json',
    Authorization: 'Bearer ' + access_token
  },
  data: JSON.stringify({
    'input': {
      'urn': urn
    },
    'output': {
      'formats': [
        {
          'type': 'svf2',
          'views': ['2d', '3d']
        }
      ]
    }
  })
})

```

You can also call `viewer.model.isSVF2()` to check if the loaded viewable is an SVF2 viewable. This is already documented here:

<https://forge.autodesk.com/en/docs/viewer/v7/reference/Viewing/Model/#issvf2>

Here are a few example statistical differences between SVF and SVF2. You will notice a large difference in number of meshes, especially in the Revit and Navisworks examples.

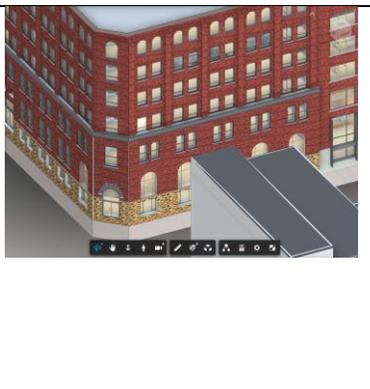
Model	Front Loader.dwfx	
Model File Size	10.3 MB	
Translation Type	SVF2	SVF
Translation time	66.29 seconds	20.78 seconds
Loading Viewable	0.57 second	1.03 seconds
Total geometry size:	27.179 MB	30.630 MB
Number of meshes:	1701	1919
Num Meshes on GPU:	1701	1919
Net GPU geom memory used:	27928096	31473282



Model	engine_type_01_2_liter_asm.zip	
Model File Size	62.8 MB	
Translation Type	SVF2	SVF
Translation time	83.04 seconds	53.97 seconds
Loading Viewable	0.77 seconds	1.16 seconds
Total geometry size:	24.084 MB	34.559 MB
Number of meshes:	307	549
Num Meshes on GPU:	307	549
Net GPU geom memory used:	25150610	36053286



Model	210 King - 2021.rvt	
Model File Size	92.8 MB	
Translation Type	SVF2	SVF
Translation time	488.49 seconds	385.35 seconds
Loading Viewable	0.60 seconds	1.51 seconds
Total geometry size:	22.589 MB	166.515 MB
Number of meshes:	3362	29646
Num Meshes on GPU:	3362	10000
Net GPU geom memory used:	22556672	101428240



Model	UrbanHouse - 2021_detached.rvt	
Model File Size	28.4 MB	
Translation Type	SVF2	SVF
Translation time	149.58 seconds	110.2 seconds
Loading Viewable	0.19 seconds	0.86 seconds
Total geometry size:	8.414 MB	17.413 MB
Number of meshes:	776	3090
Num Meshes on GPU:	776	3090
Net GPU geom memory used:	8561832	17221052



Model	ice stadium.nwd	
Model File Size	2.4 MB	
Translation Type	SVF2	SVF
Translation time	60.62 seconds	40.49 seconds
Loading Viewable	0.54 seconds	1.26 seconds
Total geometry size:	4.481 MB	12.342 MB
Number of meshes:	3854	13333
Num Meshes on GPU:	3854	10050
Net GPU geom memory used:	3403678	6572896



It is expected that SVF2 beta availability will be ready for Autodesk University. You can check the forge.autodesk.com/blogs for details. Search for Model Derivative tag.

Enhanced Model Derivative Properties API

We are planning to improve the performance on existing get properties by objectid API (/modelderivative/v2/designdata:/urn/metadata/:modelGuid/properties?objectid=:id). We have done some significant work here to help when working with very large datasets, and also the performance behind retrieving that data. We will also improve the metadata APIs to allow more granular access to the hierarchy and properties of a model. These changes will be coming soon in the form of a beta. Watch the forge blog (<https://forge.autodesk.com/blog>) for the latest information.

References

<https://forge.autodesk.com/> - Developer Portal

<https://forge.autodesk.com/en/support/get-help> - Forge help

<https://forge.autodesk.com/blog> - Forge Blog

https://forge.autodesk.com/en/docs/model-derivative/v2/developers_guide/overview/ -
Documentation for Model Derivative Service