SD473715

3ds Max Design Automation: Add Beautiful Renders to Your Web Site

Denis Grigor – Developer Advocate
Autodesk

Kevin Vandecar – Developer Advocate
Autodesk

Learning Objectives

- Review Design Automation and create a simple workflow
- Create still renderings using Design Automation for 3ds Max
- Use Forge Viewer to setup scene and send jobs to Design Automation for 3ds Max
- Understand the strengths and limitations of using Design Automation for 3ds Max as a rendering solution

Description

Generally, the Design Automation API enables you to integrate an automated batch processor into a pipeline to process files in the cloud. In the Design Automation context, Revit software, AutoCAD software, and Inventor software generally work with native file formats; but in the case of 3ds Max software, the list of file formats that we can import and work with is more extensive. This traditional strength of 3ds Max—of importing and working with a huge number of file formats—opens the way for multiple workflows, including visualization. In this class, we will present an overview of some possible Design Automation rendering workflows, as well as discuss the strengths and weaknesses of each workflow. We will explore the benefits with a pros-and-cons approach, including costs (time and credits), which will help you decide the ultimate approach and benefits.

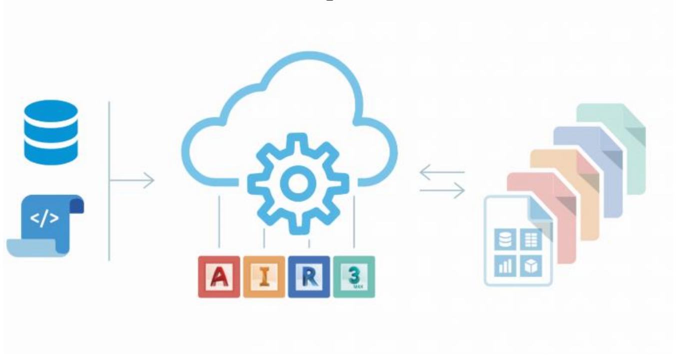
Speaker(s)

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.

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.

Review Design Automation and create a simple workflow

The Design Automation service makes possible to run automated jobs in the cloud, using CAD engines. At this moment, the available CAD engines are AutoCAD, Revit, Inventor and 3ds Max.



The important thing about Design Automation API is that it was made generic enough, so that the workflow of using it is almost similar, no matter what engine is used, making it easy to switch engines. Also, since the service is exposed through a REST API, it facilitates integration into your project, no matter what programming language you are using.

In what follows, we will go through the process of creating a simple workflow using the Design Automation for 3ds Max. This is not a replacement of the excellent learnforge.autodesk.io tutorial and some familiarity with Design Automation API is expected, but bellow we will trye to concentrate on the process itself and tips and tricks which are specific to 3ds Max area of Design Automation.

The main ingridient in mastering Design Automation API (and any api in general) is the terminology. Here we have 4 of them:

1. **Engine –** this can be seen as the environment in which our code will run;
2. **Activity –** this can be seen as a function definition, where we state the number and type of inputs, what should be done with input (using the engine) and the type of input;
3. **AppBundle –** this can be considered as the library used within the function (in our case Activity);
4. **Workitem –** this can be seen as the function call, where we specify the input and where should output be stored.

With this paradigm in mind, to use Design Automation, no matter with which engine, it resumes to following steps:

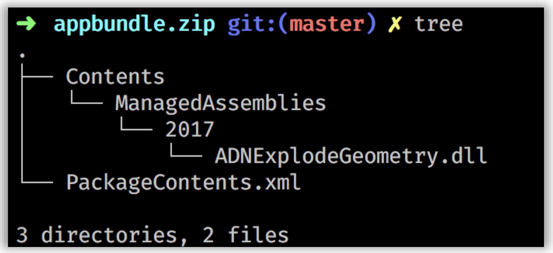
1. Create the appbundle and test it locally.
2. Upload the appbundle to the cloud.
3. Create the activity.
4. Create the work item (or more).
5. Start the job and download the results.

1. Create the appbundle and test it locally

In Design Automation, a bundle is the container that could hold custom code, plugin, scripts and any other content that is needed during every workitem execution.

The appbundle is a zip archived folder and should finish with .bundle (e.g. myFiles.bundle). Inside this folder, the first thing that should be present is PackageContents.xml, which describes to the used engine (g.e. 3ds Max) what to load on startup. For more documentation on this xml file check here.

For example, an AppBundle containing a 3ds Max plugin written using 3ds Max .Net SDK, could the following structure:



```
➜ appbundle.zip git:(master) ✗ tree
.
├── Contents
│   └── ManagedAssemblies
│       └── 2017
│           └── ADNExplodeGeometry.dll
└── PackageContents.xml

3 directories, 2 files
```

and the minimal amount of specification needed inside PackageContents.xml could be the following:

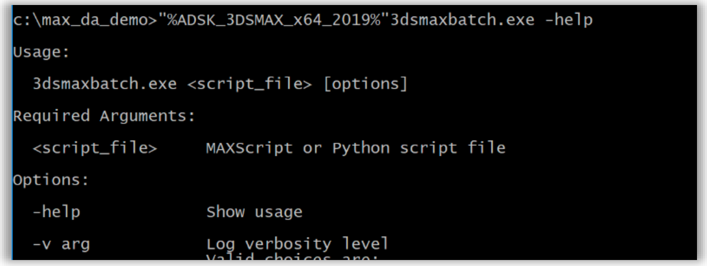
```
<?xml version="1.0" encoding="utf-8"?>
<ApplicationPackage
        AutodeskProduct="3ds Max"
        ProductType="Application"
        ProductCode="3ds Max Explode geometry plug-in.">

        <CompanyDetails />

        <Components Description="pre-start-up plugin part">
                <RuntimeRequirements
                                        OS="Win64"
                                        Platform="3ds Max"
                                        SeriesMin="2017"
                                        SeriesMax="2019" />
                <ComponentEntry
                                AppName="ADNExplodeGeometry"
                                Version="2.1.0"

        ModuleName="./Contents/ManagedAssemblies/2017/ADNExplodeGeometry.dll" />
        </Components>
</ApplicationPackage>
```

Note: The above xml is just the minimum necessary, but the schema for the ***PackageContents.xml*** can accept a lot of information. For the full list, check here.
Warning: The minimum amount of information necessary to be present within ***PackageContents.xml*** is increasing with each release of 3ds Max. What was as optional field in 3ds Max 2019, might become mandatory field in 3ds Max 2020. The back compatibility is

assured, but for forward compatibility, when using a newer engine of 3ds Max, make sure to check the Autodesk Application Plug-in Package Format Specification for that version.

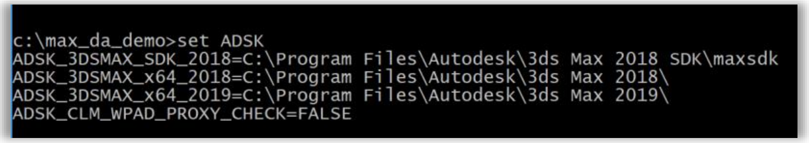
Testing the appbundle locally

Before using the appbundle in Design Automation workflow, it is a good practice to test it locally. For this purpose, you can use 3dsmaxbatch.exe located in the root of the 3ds Max installation folder:



```
c:\max_da_demo>"%ADSK_3DSMAX_x64_2019%"3dsmaxbatch.exe -help
Usage:
  3dsmaxbatch.exe <script_file> [options]
Required Arguments:
  <script_file>      MAXScript or Python script file
Options:
  -help              Show usage
  -v arg             Log verbosity level
```

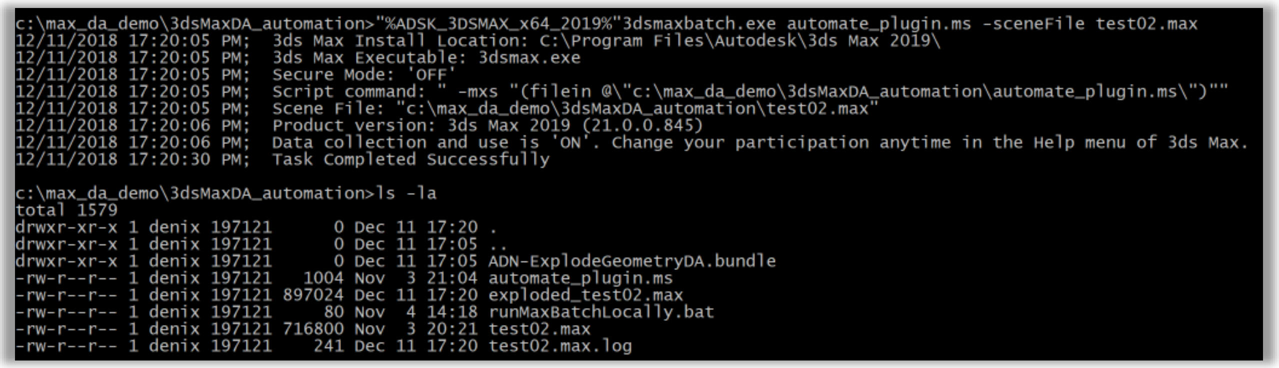
Tip: Note the use of environment variable to specify the 3ds installation folder. To get these environment variables, just type in Command Line set ADSK:



```
c:\max_da_demo>set ADSK
ADSK_3DSMAX_SDK_2018=C:\Program Files\Autodesk\3ds Max 2018 SDK\maxsdk
ADSK_3DSMAX_x64_2018=C:\Program Files\Autodesk\3ds Max 2018\
ADSK_3DSMAX_x64_2019=C:\Program Files\Autodesk\3ds Max 2019\
ADSK_CLM_WPAD_PROXY_CHECK=FALSE
```

The workflow of using the ***3dsmaxbatch.exe*** is based on passing to it an execution script and the scene file to work with. If needed, there is also the option of passing arguments for values (numbers) and strings.To understand the use of ***3dsmaxbatch.exe***, please consult Using 3ds Max Batch.

The simplest use of ***3dsmaxbatch.exe*** is passing a script (let us call it work script) and the scene file ***3dsmaxbatch.exe myScript.ms -sceneFile myMaxFile.max***:



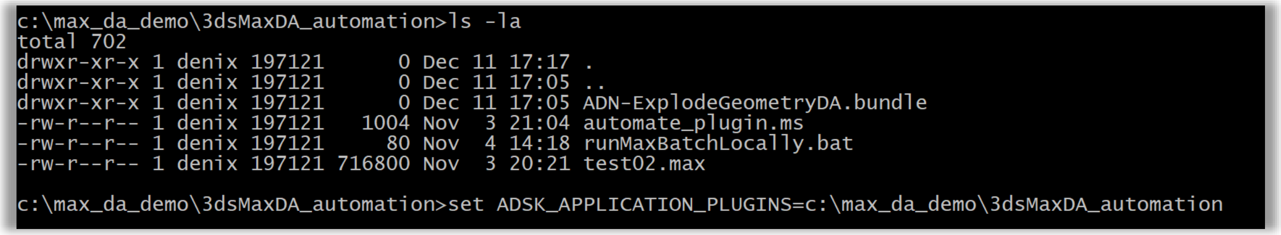
```
c:\max_da_demo\3dsMaxDA_automation>"%ADSK_3DSMAX_x64_2019%"3dsmaxbatch.exe automate_plugin.ms -sceneFile test02.max
12/11/2018 17:20:05 PM;  3ds Max Install Location: C:\Program Files\Autodesk\3ds Max 2019\
12/11/2018 17:20:05 PM;  3ds Max Executable: 3dsmax.exe
12/11/2018 17:20:05 PM;  Secure Mode: 'OFF'
12/11/2018 17:20:05 PM;  Script command: " -mxs "(filein @\"c:\max_da_demo\3dsMaxDA_automation\automate_plugin.ms\")""
12/11/2018 17:20:05 PM;  Scene File: "c:\max_da_demo\3dsMaxDA_automation\test02.max"
12/11/2018 17:20:06 PM;  Product version: 3ds Max 2019 (21.0.0.845)
12/11/2018 17:20:06 PM;  Data collection and use is 'ON'. Change your participation anytime in the Help menu of 3ds Max.
12/11/2018 17:20:30 PM;  Task Completed Successfully

c:\max_da_demo\3dsMaxDA_automation>ls -la
total 1579
drwxr-xr-x 1 denix 197121      0 Dec 11 17:20 .
drwxr-xr-x 1 denix 197121      0 Dec 11 17:05 ..
drwxr-xr-x 1 denix 197121      0 Dec 11 17:05 ADN-ExplodeGeometryDA.bundle
-rw-r--r-- 1 denix 197121   1004 Nov  3 21:04 automate_plugin.ms
-rw-r--r-- 1 denix 197121 897024 Dec 11 17:20 exploded_test02.max
-rw-r--r-- 1 denix 197121     80 Nov  4 14:18 runMaxBatchLocally.bat
-rw-r--r-- 1 denix 197121 716800 Nov  3 20:21 test02.max
-rw-r--r-- 1 denix 197121    241 Dec 11 17:20 test02.max.log
```

The complexity of the script file can vary and either contain the MAXScript code to do some operation on the file or invoke plugin operations.

In case the work script is invoking plugin operations, we must make sure that before the 3ds max is running, the needed plugin was located and loaded.

3ds Max has special location for plugins, but in our case, we are more interested in ways of informing 3ds Max of our custom location of the self contained plugin.

The bundle format is assuring that the plugin is self-contained and will be picked up by the 3ds Max and properly installed. The location of the bundle can be specified to 3ds Max by setting the **ADSK_APPLICATION_PLUGINS** environment variable:

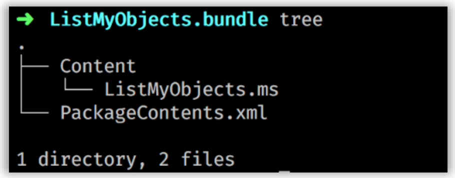


```
c:\max_da_demo\3dsMaxDA_automation>ls -la
total 702
drwxr-xr-x 1 denix 197121      0 Dec 11 17:17 .
drwxr-xr-x 1 denix 197121      0 Dec 11 17:05 ..
drwxr-xr-x 1 denix 197121      0 Dec 11 17:05 ADN-ExplodeGeometryDA.bundle
-rw-r--r-- 1 denix 197121   1004 Nov  3 21:04 automate_plugin.ms
-rw-r--r-- 1 denix 197121     80 Nov  4 14:18 runMaxBatchLocally.bat
-rw-r--r-- 1 denix 197121 716800 Nov  3 20:21 test02.max

c:\max_da_demo\3dsMaxDA_automation>set ADSK_APPLICATION_PLUGINS=c:\max_da_demo\3dsMaxDA_automation
```

For example, suppose that you create a MAXScript script (or C++, .Net plugin) that exposes just a simple function of listing the objects and saving the results into list.txt file:

```
fn ListMyObject =
(
  dir = sysInfo.currentdir + "/list.txt"
  f = (createFile dir)
  allObjects = $*
  for OBJ in allObjects do(
        val1 = OBJ as string
        val1 += "\n"
        format val1 to:f
  )
)
```

we can create a folder with the following structure



```
➜ ListMyObjects.bundle tree
.
├── Content
│   └── ListMyObjects.ms
└── PackageContents.xml

1 directory, 2 files
```

where the PackageContents.xml will have following content:

```
<?xml version="1.0" encoding="utf-8"?>
<ApplicationPackage
                                  SchemaVersion="1.0"
                                  AutodeskProduct="3ds Max"
                                  ProductType="Application"
                                  ProductCode="3ds Max Design Share Plugin">
<CompanyDetails />
  <Components Description="pre-start-up scripts parts">
        <RuntimeRequirements
                         OS="Win64"
                         Platform="3ds Max"
```

```
                          SeriesMin="2018"
                          SeriesMax="2019" />
        <ComponentEntry
                          AppName="ListMyObjects"
                          Version="1.0.0"
                          ModuleName="./Content/ListMyObjects.ms" />
  </Components>
</ApplicationPackage>
```

After we zip it, if our current bundle is located at C:\my_bundle, then before invoking the **3dsmaxbatch.exe**, we should write in **Command Prompt**:

```
set ADSK_APPLICATION_PLUGINS=c:\my_bundle
```

This will make sure that before **3dsmaxbatch.exe** loads the scene and executes the work script, it will look into this folder and load the bundle located there.
Next, we could write a simple working script (let us call it ***run.ms***) that just invokes that function:

```
ListMyObject()
```

all what is left is just make sure that we are in the same folder as the running script and the scene (or specify the absolute/relative path) and run
`3dsmaxbatch.exe run.ms -sceneFile myMaxFile.max`
After the task is complete, we should have a ***list.txt*** in your folder, listing all the objects from the scene.
This covered the simple use-case with no passed params. In case we need to pass some configuration params, the documentation on 3dsmaxbatch.exe provides more details on this, but the essential are the following arguments:

```
 -mxsString arg    MAXScript String Parameter option, where arg=<key>:<value>
                    <value> will be accessible as a MAXScript string in
                    <script_file> through the maxOps.mxsCmdLineArgs dictionary
                    using <key> as the dictionary key.
                    Can be specified multiple times.

  -mxsValue arg     MAXScript Value Parameter option, where arg=<key>:<value>
                    <value> will be accessible as a MAXScript value in
                    <script_file> through the maxOps.mxsCmdLineArgs dictionary
```

As the documentation mentions, the arguments will be available to the work script through ***maxOps.mxsCmdLineArgs*** dictionary:

For example, consider the following Command Prompt invocation:

```
>3dsmaxbatch demo_mxscmdline.ms -mxsString string1:"Hello"  -mxsValue
value1:"[3,12,22]" -mxsValue value2:"Point3 3 8 2.3" -listenerLog "test.log"
```

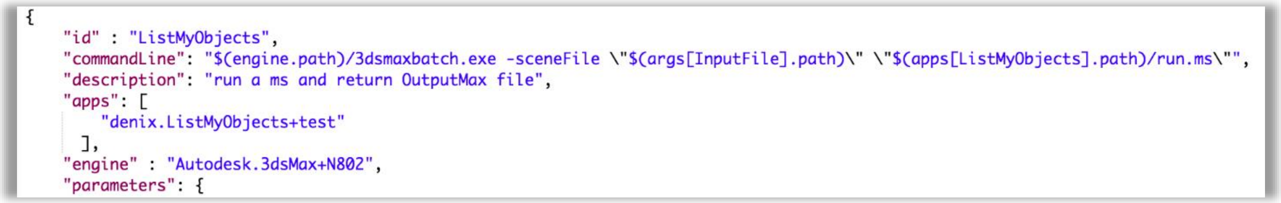
and the work script:

```
opts = maxOps.mxsCmdLineArgs
for k in opts.keys do (
  format "% == % class: %\n" k opts[k] (classOf opts[k])
)
```

the test.log file (we specified this file as the last argument) will contain:

```
#string1 == Hello class: String
#value2 == [3,8,2.3] class: Point3
#value1 == [3,12,22] class: Point3
```

All this is an important step in understanding the Design Automation workflow because when creating an activity, you have to specify the commandLine param, which will look like:



```
{
    "id" : "ListMyObjects",
    "commandLine": "$(engine.path)/3dsmaxbatch.exe -sceneFile \"$(args[InputFile].path)\" \"$(apps[ListMyObjects].path)/run.ms\"",
    "description": "run a ms and return OutputMax file",
    "apps": [
        "denix.ListMyObjects+test"
     ],
    "engine" : "Autodesk.3dsMax+N802",
    "parameters": {
```

and by now, this line should look familiar to you and we will get back to it later when creating an activity.

2.Upload the appbundle to the cloud

Once we have everything set and tested, we can prepare moving this workload to the cloud and make use of Design Automation API.

If locally, we can set **ADSK_APPLICATION_PLUGINS** environment variable, for the local 3ds Max to know where to find our appbundle, in context of Design Automation API, we will have to upload it somewhere in the cloud. This is done in 2 steps:

1. Create an App through call to **https://developer.api.autodesk.com/da/us-east/v3/appbundles** by specifying:

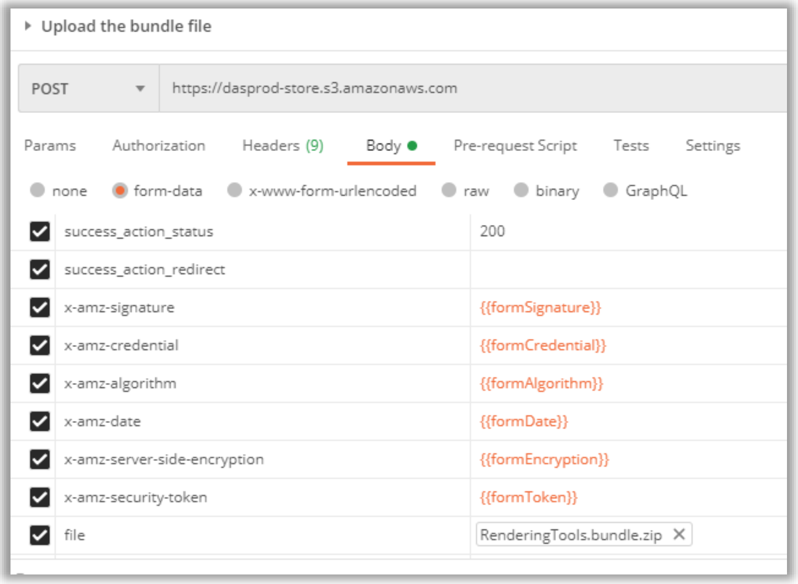
   ```
   {
     "engine" : "Autodesk.3dsMax+2018",
     "id": "myDemoApp"
   }
   ```

   the reply from this code will contain a bunch of data, including the security parts:

   ```
   {
       "uploadParameters": {
           "endpointURL": "https://dasprod-store.s3.amazonaws.com",
           "formData": {
   ```

```
                "key": "apps/Denis/myDemoApp/1",
                "content-type": "application/octet-stream",
                "policy": "ey...X0=",
                "success_action_status": "200",
                "success_action_redirect": "",
                "x-amz-signature": "01d...292",
                "x-amz-credential": "AS...I/20181212/us-east-1/s3/aws4_request/",
                "x-amz-algorithm": "AWS4-HMAC-SHA256",
                "x-amz-date": "20181212T232929Z",
                "x-amz-server-side-encryption": "AES256",
                "x-amz-security-token": "FQ...4AU="
            }
        },
        "engine": "Autodesk.3dsMax+2018",
        "version": 1,
        "id": "Denis.myDemoApp"
}
```

2. Upload the appbundle to ***https://dasprod-store.s3.amazonaws.com*** by passing as **form-data** the **key-values** received in the pervious request and specified as ***formData*** object. You will have to add the bundle file as value to the file key of this form.
   An example using Postman is presented below:



POST https://dasprod-store.s3.amazonaws.com

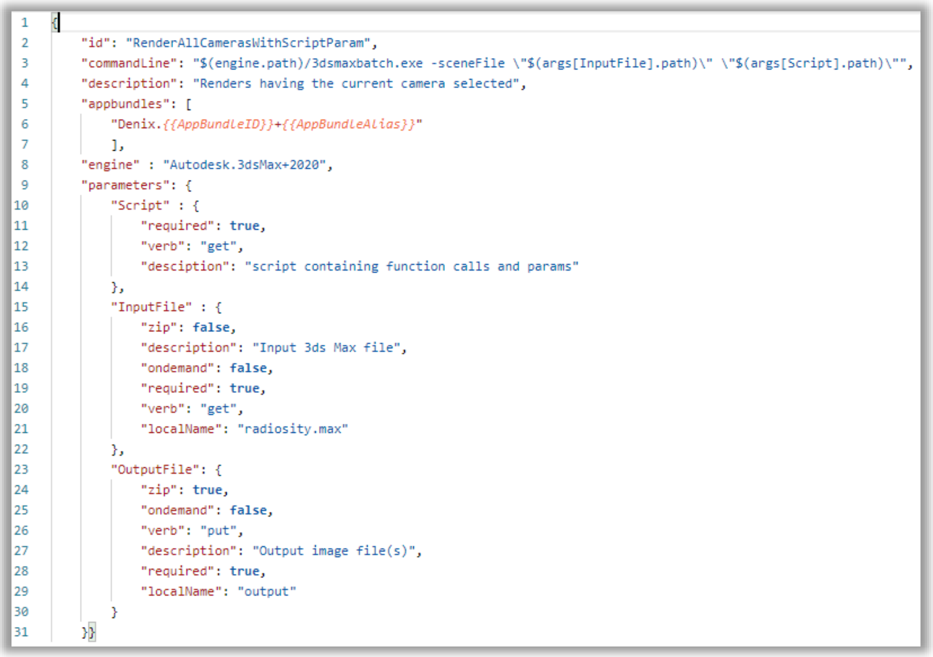
none [✓ form-data] x-www-form-urlencoded raw binary GraphQL

✓	success_action_status	200
✓	success_action_redirect	
✓	x-amz-signature	{{formSignature}}
✓	x-amz-credential	{{formCredential}}
✓	x-amz-algorithm	{{formAlgorithm}}
✓	x-amz-date	{{formDate}}
✓	x-amz-server-side-encryption	{{formEncryption}}
✓	x-amz-security-token	{{formToken}}
✓	file	RenderingTools.bundle.zip

So far, all this was the preparation part, maybe a bit annoying, but a necessary step. From this point things get more interesting.

3. Create the activity

As mentioned before, creating an activity could be seen as defining a function. For example in the following activity:



```
{
    "id": "RenderAllCamerasWithScriptParam",
    "commandLine": "$(engine.path)/3dsmaxbatch.exe -sceneFile \"$(args[InputFile].path)\" \"$(args[Script].path)\"",
    "description": "Renders having the current camera selected",
    "appbundles": [
        "Denix.{{AppBundleID}}+{{AppBundleAlias}}"
        ],
    "engine" : "Autodesk.3dsMax+2020",
    "parameters": {
        "Script" : {
            "required": true,
            "verb": "get",
            "desciption": "script containing function calls and params"
        },
        "InputFile" : {
            "zip": false,
            "description": "Input 3ds Max file",
            "ondemand": false,
            "required": true,
            "verb": "get",
            "localName": "radiosity.max"
        },
        "OutputFile": {
            "zip": true,
            "ondemand": false,
            "verb": "put",
            "description": "Output image file(s)",
            "required": true,
            "localName": "output"
        }
    }}
```

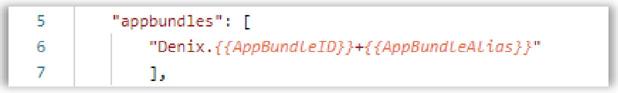
It might look daunting at first look, but once used, it gets pretty simple. The parameters part defines the inputs and outputs, where the "verb" field is the one distinguishing them. All parameters which have "get" as a verb, can be seen as params of a function and so on. In other words, this could be translated as something like:

```
func RenderAllCamerasWithScriptParam(Script, InputFile) => OutputFile {
}
```

as a pseudo function call.
The **commandLine** can be seen as the function body, and indeed if we look at it

```
"commandLine": "$(engine.path)/3dsmaxbatch.exe
                                      -sceneFile \"$(args[InputFile].path)\"
                                      \"$(args[Script].path)\""
```

First, it should look very familiar, because it is the same thing we did when we were testing the AppBundle in the local environment. Secondly, it collects all input parameters (along with engine version) and passes it to the cloud instance of 3dsmaxbatch.
All this parametrization makes switching 3ds Max version, input files and so on a very simple task, like a function definition is an abstraction and parametrization of a repetitive chunk of code.
To complete the analogy, in this picture, the appbundles field:

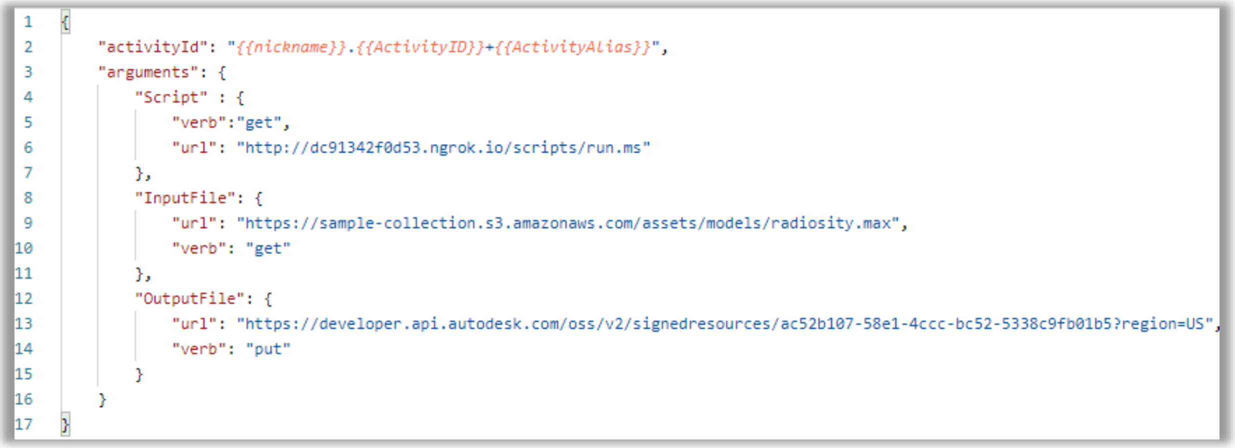


```
    "appbundles": [
        "Denix.{{AppBundleID}}+{{AppBundleAlias}}"
        ],
```

can be seen as importing external libraries, which in it's essence is, because it can be freely reused within other activities.

4. Create the work item (or more)

To continue the above analogy, a workitem is like a function call and it's body content:



```
{
    "activityId": "{{nickname}}.{{ActivityID}}+{{ActivityAlias}}",
    "arguments": {
        "Script" : {
            "verb":"get",
            "url": "http://dc91342f0d53.ngrok.io/scripts/run.ms"
        },
        "InputFile": {
            "url": "https://sample-collection.s3.amazonaws.com/assets/models/radiosity.max",
            "verb": "get"
        },
        "OutputFile": {
            "url": "https://developer.api.autodesk.com/oss/v2/signedresources/ac52b107-58e1-4ccc-bc52-5338c9fb01b5?region=US",
            "verb": "put"
        }
    }
}
```

could be translated into a pseudo code as:

```
outputFile := NameOfActivity(Script = “the url”, InputFile = “the other url”)
```

and intuitively, this kind of function calls can be made in numbers and independent of each other, which we will explore in more details in next section.

Create still renderings using Design Automation for 3ds Max

In previous part we listed the necessary ingredients to be able to use Design Automation APi in general and for 3ds Max environment in special.

Now let us take a simple task of creating a rendered image out of given scene. In desktop environment we would load a scene, trigger the rendering, and save the result, all this by clicking the necessary buttons in menus, Render Dialog etc.

For Design Automation workflow, we must think on how we would do that in headless mode, without any human interaction and this is where the scripting power comes in play.
In context of 3ds Max, we could use Python and the pymxs library, but the most simple and straightforward way is just to use MAXScript. To trigger the rendering task and save the image the following line should do the job:

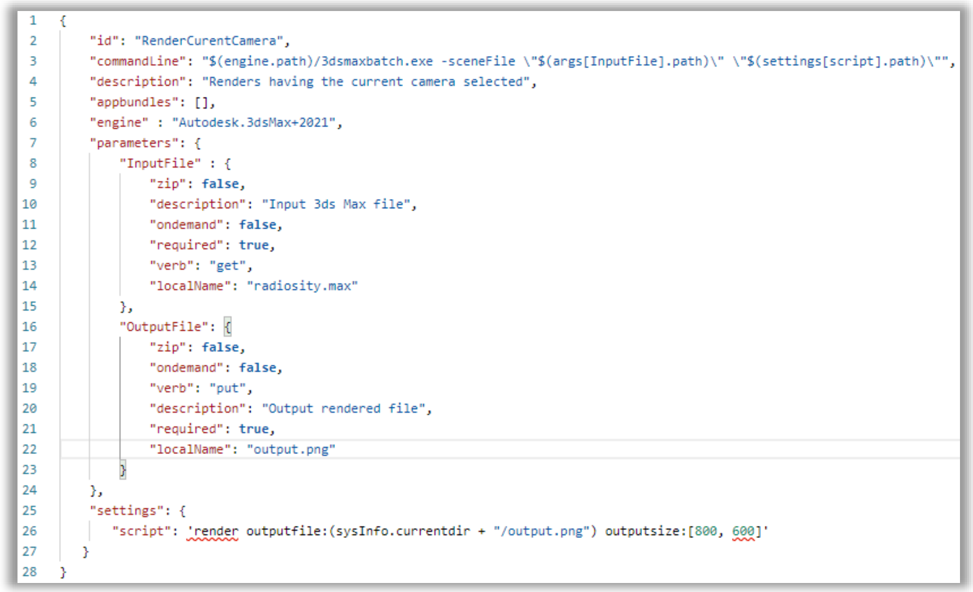
```
render outputfile:(sysInfo.currentdir + "/output.png") outputsize:[800, 600]
```

Now, if we want to transfer this instruction to Design Automation environment, we have a choice here:

- Either we wrap this MAXScript command into a function (it could be a function which accepts width and height as its params) and invoke when defining the Activity.
- Or call this line directly within the Activity.

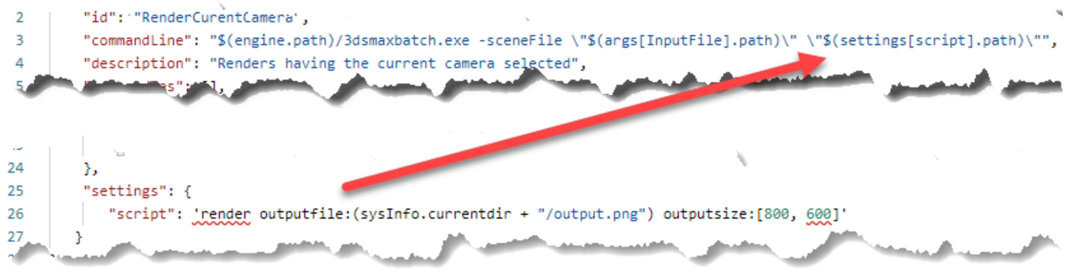
At first sight, since this is a simple line, going with the first choice might look as an overkill, but later we will see that this is not the case, especially when you want to reuse it in different activities. For now, we will proceed with second option, but later I will explain why this is not always a good choice.

With second choice, the body of the activity could look like this:



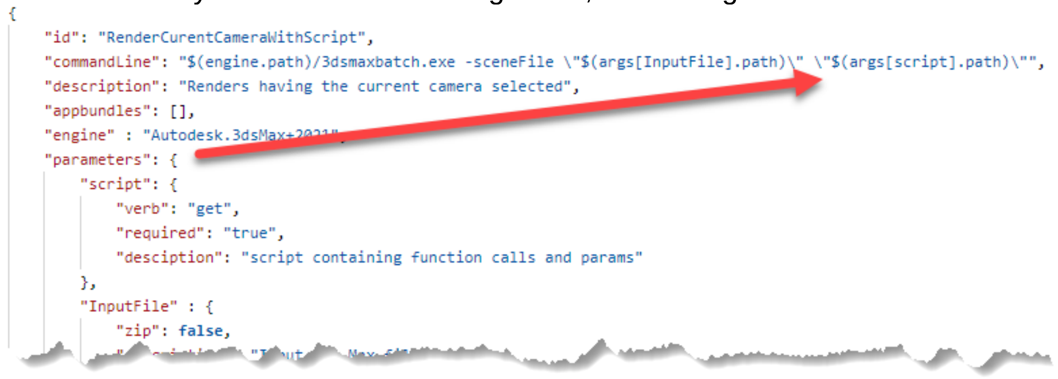
```
{
    "id": "RenderCurentCamera",
    "commandLine": "$(engine.path)/3dsmaxbatch.exe -sceneFile \"$(args[InputFile].path)\" \"$(settings[script].path)\"",
    "description": "Renders having the current camera selected",
    "appbundles": [],
    "engine" : "Autodesk.3dsMax+2021",
    "parameters": {
        "InputFile" : {
            "zip": false,
            "description": "Input 3ds Max file",
            "ondemand": false,
            "required": true,
            "verb": "get",
            "localName": "radiosity.max"
        },
        "OutputFile": {
            "zip": false,
            "ondemand": false,
            "verb": "put",
            "description": "Output rendered file",
            "required": true,
            "localName": "output.png"
        }
    },
    "settings": {
      "script": 'render outputfile:(sysInfo.currentdir + "/output.png") outputsize:[800, 600]'
   }
}
```

By now, the content of the activity should be familiar to you, but in this case the new part is the **settings** part where we state the body of the script and in **commandLine** field, we reference it:



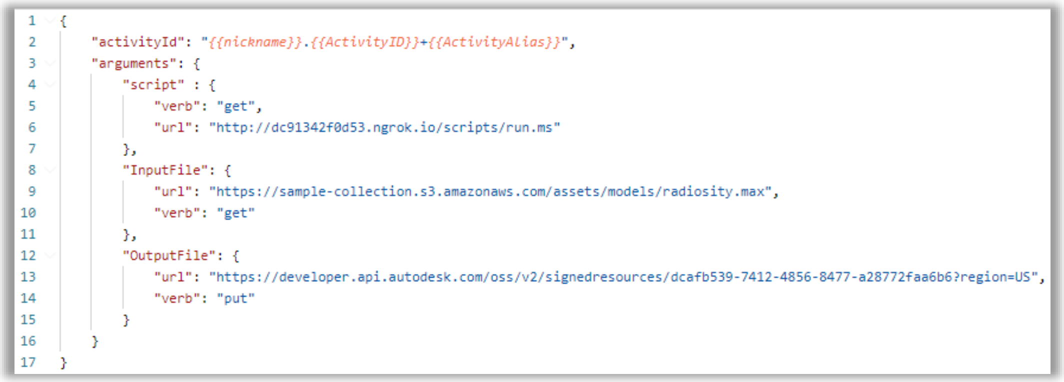
```
    "id": "RenderCurentCamera",
    "commandLine": "$(engine.path)/3dsmaxbatch.exe -sceneFile \"$(args[InputFile].path)\" \"$(settings[script].path)\"",
    "description": "Renders having the current camera selected",

    },
    "settings": {
      "script": 'render outputfile:(sysInfo.currentdir + "/output.png") outputsize:[800, 600]'
   }
```

This is the way to go only if you have that script constant. If we want to slightly change something (like the output size to 1920x1080), you will have to create another activity and in the end you will end up having a bunch of activities for different resolutions and other tiny settings. From this perspective, a better approach would be to have this script as an input parameter, then the activity would become more generic, if not too generic:



```
{
    "id": "RenderCurentCameraWithScript",
    "commandLine": "$(engine.path)/3dsmaxbatch.exe -sceneFile \"$(args[InputFile].path)\" \"$(args[script].path)\"",
    "description": "Renders having the current camera selected",
    "appbundles": [],
    "engine" : "Autodesk.3dsMax+2021",
    "parameters": {
        "script": {
            "verb": "get",
            "required": "true",
            "desciption": "script containing function calls and params"
        },
        "InputFile" : {
            "zip": false,
```

Too generic, because even if we named this activity *"RenderCurentCameraWithScript"*, nothing prevents us, upon workitem creation, to pass a script file, where we do some totally different things, like enumerating and creating a list of all lights in the scene. This is “bug attracting spot”, but it gives the necessary flexibility of changing the needed params from our script.

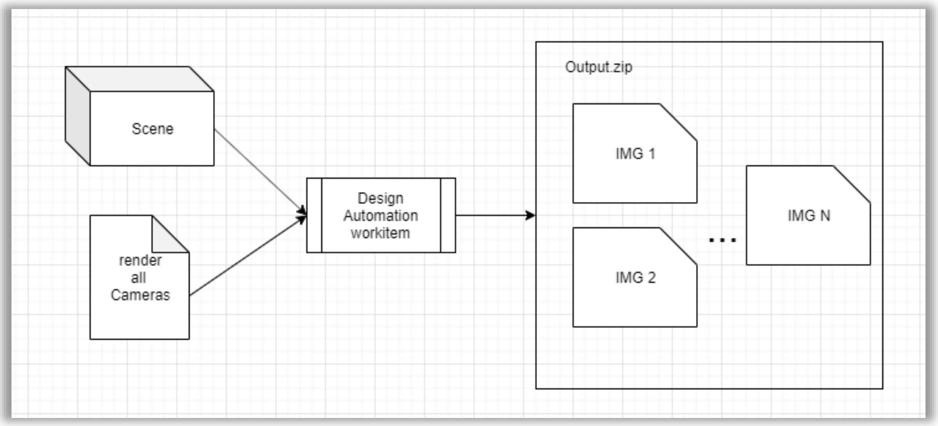


```
{
    "activityId": "{{nickname}}.{{ActivityID}}+{{ActivityAlias}}",
    "arguments": {
        "script" : {
            "verb": "get",
            "url": "http://dc91342f0d53.ngrok.io/scripts/run.ms"
        },
        "InputFile": {
            "url": "https://sample-collection.s3.amazonaws.com/assets/models/radiosity.max",
            "verb": "get"
        },
        "OutputFile": {
            "url": "https://developer.api.autodesk.com/oss/v2/signedresources/dcafb539-7412-4856-8477-a28772faa6b6?region=US",
            "verb": "put"
        }
    }
}
```

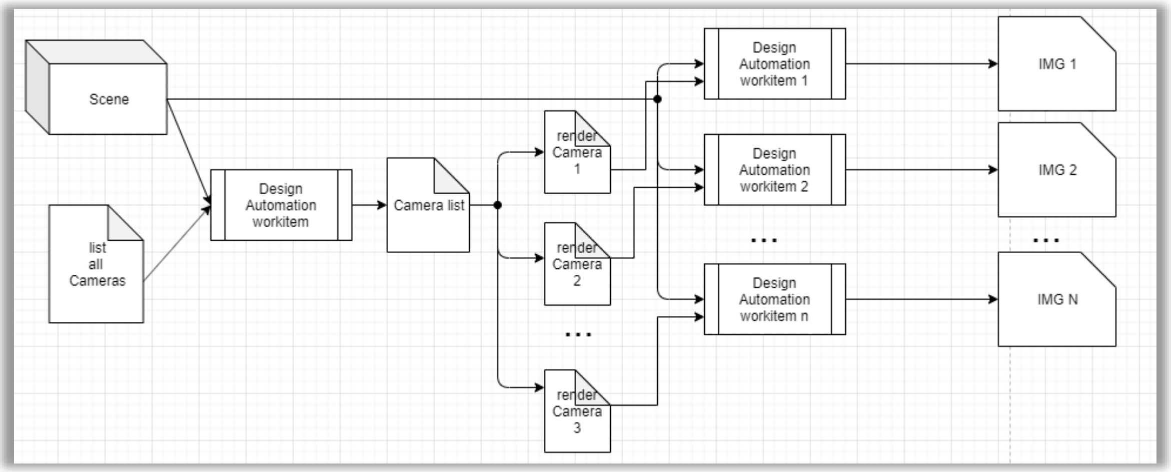
Having this approach, we can submit multiple workitems, with different InputFile url, allowing us to automate a task like render in parallel all scenes present in a given folder or set of folders. A classic example of a job needing parallelism.

Now, armed with this workflow, we can apply it to a more difficult task. A scene can have multiple cameras and in case we would like to automate creation of some sort of a gallery, we would like to get a still rendering for each camera. At this point, we are presented with following choices:

- We can send the task of rendering all cameras to a single workitem. This being a serial task, the time between submitting the job and getting the result is the sum of each rendering.



- We can send a task of getting the name of all cameras in the scene, then using the output of that job, to create a rendering task for each camera we have in the scene. In this case the time between submitting the job and getting the result is the time of processing the task of getting all cameras plus the longest rendering task.



Scene
list all Cameras
Design Automation workitem
Camera list
render Camera 1
render Camera 2
...
render Camera 3
Design Automation workitem 1
Design Automation workitem 2
...
Design Automation workitem n
IMG 1
IMG 2
...
IMG N

We will get the result faster with second approach than the first one due to parallelism, but this does not mean that it will be more cost effective. Here starts to sneak one of the main drawbacks of using Design Automation API as a rendering solution. Note in second approach that each workitem, when working on rendering a camera, it will copy the scene to its environment. This happens because (for different reasons, including security one) each workitem is running in a sandbox environment, with no communication with other workitems and this is why they have to make their own copy of data, which for some tasks is not efficient. In our example, if in a scene file we have 100 cameras and we want to render each camera in parallel, that scene file will be copied 100 times. In this case the rendering is a very good example, but it is not limited to it.

Design Automation service is good for automating tasks, but we must be careful in what we automate, understand the limitations and weight the benefits of using it for certain tasks.

Up till now, we explored the use of Design Automation service by feeding files and scripts, while in what follows, we will try to increase the complexity and add some interactivity through use of Forge Viewer.

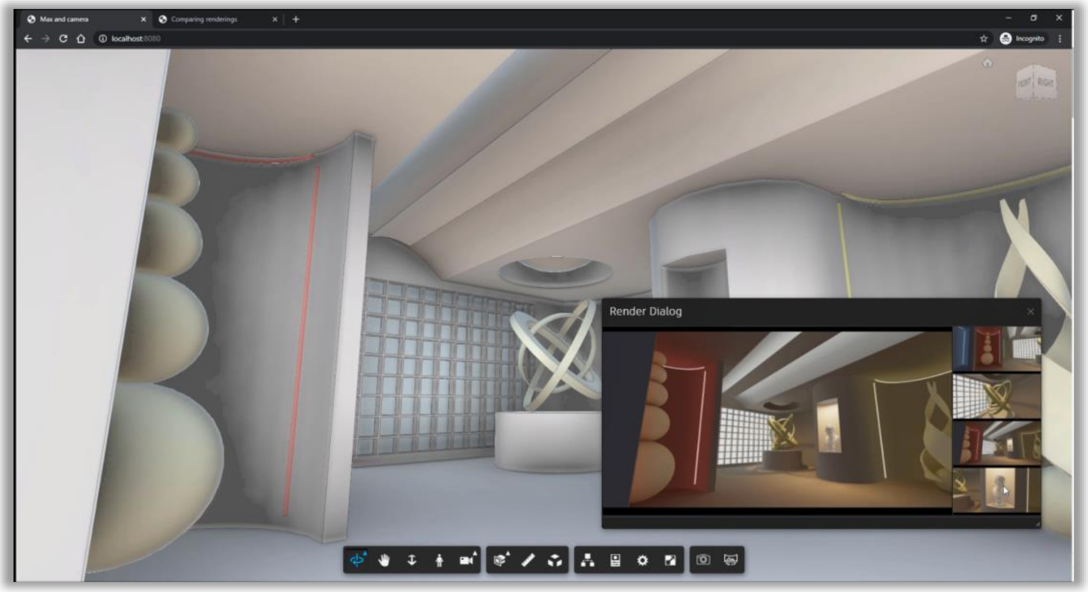
Use Forge Viewer to setup scene and send jobs to Design Automation for 3ds Max

By now we know how to render a singe image and even a pack of images in one shot. All this can be easily transferred to code and integrated into a pipeline, with task like “Create me a gallery out of bunch of scenes I have in this folder”. I will not go into details on how to achieve that, but it is on my to-do list as a nice and simple sample.

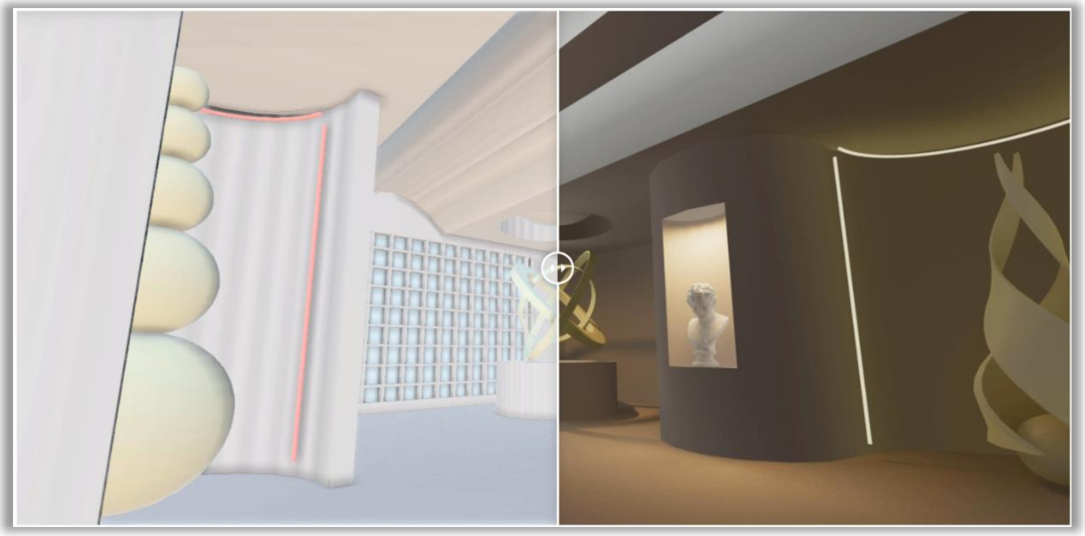
What I propose next is to look at integration of several Forge services in one project (including Design Automation) and create a “sort of” interactive experience. “Sort of”, because a rendering task is not an easy task and can take from couple of minutes to couple of hours. Nevertheless, the plan is simple:

- Use the Forge Viewer to visualize and navigate the scene
- Allow user, after selecting the needed pan-view-rotate position, to submit a rendering job to render the original scene from the same angle/view/camera.
- Upon completion, display the rendering results and allow to compare with the original view.

In other words, we want something like this:



and be able to compare what you see with what was rendered:

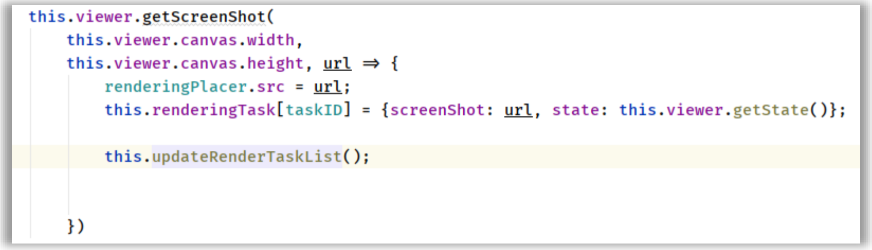


Note that the Forge Viewer is a very good option here as UI, because it allows to previsualize the rendering.

Introduction to Forge Viewer and writing extensions is beyond scope of this class and deserves a class on its own, of not several. A good source for hands-on introduction to Viewer is https://learnforge.autodesk.io/ in general and specifically the parts View your models and Viewer extension. For more advanced information on how to create an extension, please check the Advanced Techniques for Using the Forge Viewer class. Once you get familiar with these topics, you will quickly understand that writing an extension like in this demo is not a rocket science. At the end you will find the link to the GitHub repo and explore the code in more detail, but in what follows I will go with a quick overview and concentrate on one “secret souse” which makes this work smoothly.

The skeleton of an extension is more or less generic and usually the business logic is the one which distinct one extension from another.
In this extension the workflow is simple. After the user opens the rendering dialog, once he presses the Render button, the following will happen:

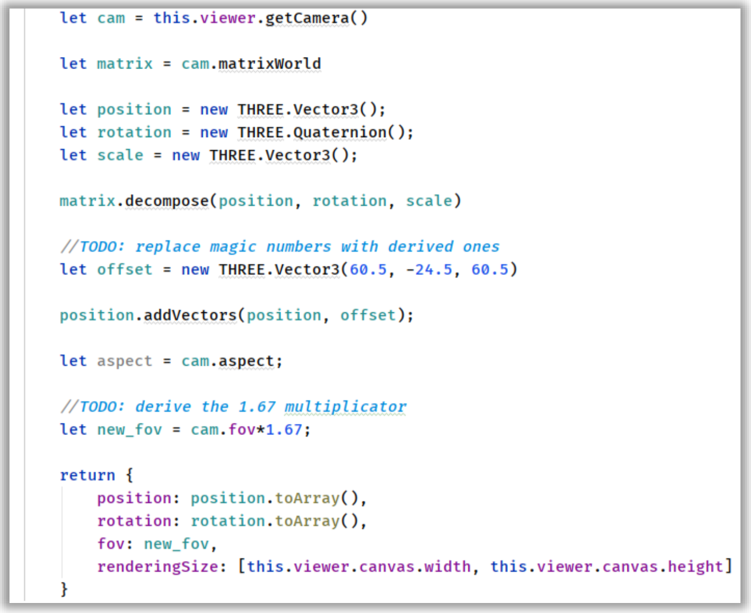
1. We get a screenshot through the viewer:



```
this.viewer.getScreenShot(
    this.viewer.canvas.width,
    this.viewer.canvas.height, url ⇒ {
        renderingPlacer.src = url;
        this.renderingTask[taskID] = {screenShot: url, state: this.viewer.getState()};

        this.updateRenderTaskList();


    })
```

   and save the screenshot url, as well as the viewer state. Saving viewer state allows us, upon click on the rendering task image, to navigate to same camera params (position, lookAt etc) to the moment when we set the renderingJob.

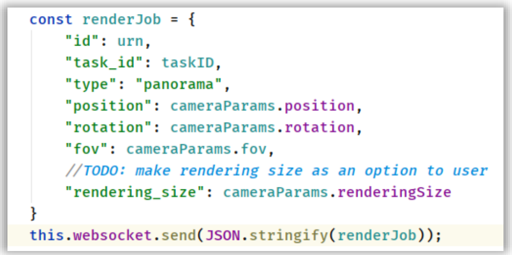
2. We get the camera params:



```
let cam = this.viewer.getCamera()

let matrix = cam.matrixWorld

let position = new THREE.Vector3();
let rotation = new THREE.Quaternion();
let scale = new THREE.Vector3();

matrix.decompose(position, rotation, scale)

//TODO: replace magic numbers with derived ones
let offset = new THREE.Vector3(60.5, -24.5, 60.5)

position.addVectors(position, offset);

let aspect = cam.aspect;

//TODO: derive the 1.67 multiplicator
let new_fov = cam.fov*1.67;

return {
    position: position.toArray(),
    rotation: rotation.toArray(),
    fov: new_fov,
    renderingSize: [this.viewer.canvas.width, this.viewer.canvas.height]
}
```

And here is the “secret souse” and the main secret here are those magic numbers. We will talk in detail later about these magic numbers and methods on how they can be derived. But in essence, we decompose the camera matrix for position and rotation, as well as the aspect and fov (field of view).

3. We create a render job, providing the params we got from above camera disassembling and submit it through websocket session to our backend. The web socket is not really necessary for submitting the job – a POST REST call could be ok, but anyway we keep a websocket session alive, to be notified when the rendering is done, get the rendered image and replace the thumbnails and the main screenshot.



```
const renderJob = {
    "id": urn,
    "task_id": taskID,
    "type": "panorama",
    "position": cameraParams.position,
    "rotation": cameraParams.rotation,
    "fov": cameraParams.fov,
    //TODO: make rendering size as an option to user
    "rendering_size": cameraParams.renderingSize
}
this.websocket.send(JSON.stringify(renderJob));
```

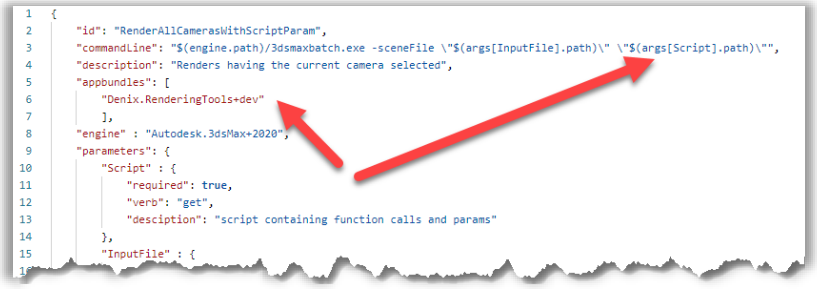
4. On the server side, it accepts this payload and using a template:

```
scriptTemplate, err = template.New( name: "rendering").Parse( text: "renderAtView  [{{.Posx}}, {{.Posy}}, {{.Posz}}] " +
    "[{{.Rotx}}, {{.Roty}}, {{.Rotz}}, {{.Rotw}}] {{.Fov}} \"{{.TaskID}}\" {{.Width}} {{.Height}}")
```

it renders it to simple one line MAXScript file, something like:

```
renderAtView  [121.56692, -0.39253235, 71.706505] [0.322742, 0.5246158, 0.67098564, 0.4127882] 98.0549 "1591290894488" 1920 937
```

If you remember, in one of above explanations, I mentioned that passing a script as an input, gives room for flexibility. This is the case of that flexibility. For each rendering task we assemble this MAXScript file and submit is an input:



```
{
    "id": "RenderAllCamerasWithScriptParam",
    "commandLine": "$(engine.path)/3dsmaxbatch.exe -sceneFile \"$(args[InputFile].path)\" \"$(args[Script].path)\"",
    "description": "Renders having the current camera selected",
    "appbundles": [
        "Denix.RenderingTools+dev"
        ],
    "engine" : "Autodesk.3dsMax+2020",
    "parameters": {
        "Script" : {
            "required": true,
            "verb": "get",
            "desciption": "script containing function calls and params"
        },
        "InputFile" : {
```

This script in its essence, just calls one our functions I packaged in an appbundle

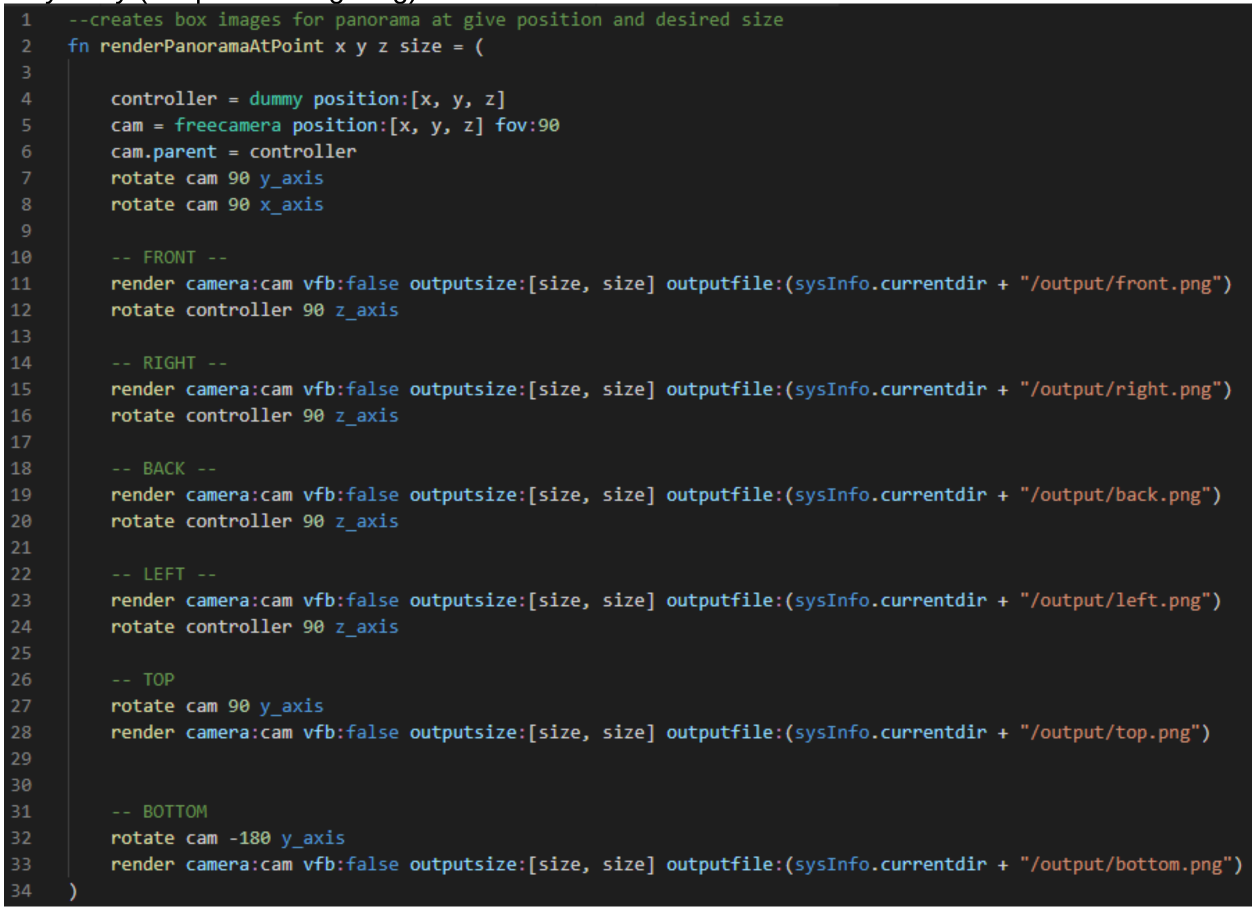
```
--render a view given position, rotation and FOV
--used to match Viewer camera and get rendering of that camera
fn renderAtView pos rot fov name w h = (
    cam = freecamera rotation:(quat rot.x rot.y rot.z rot.w) position:[pos.x, pos.y, pos.z] fov:fov
    render camera:cam outputsize:[w, h] outputfile:(sysInfo.currentdir + "/output/" + name + ".png")

)
```

and now it is reused in different context, including this one.

What we have is an activity which defines an input script, an input file, the used appbundle and the output. Each rendering task is just changing the input script, and the output – the bundle stays the same and the scene stays the same (because we are working with very same scene).

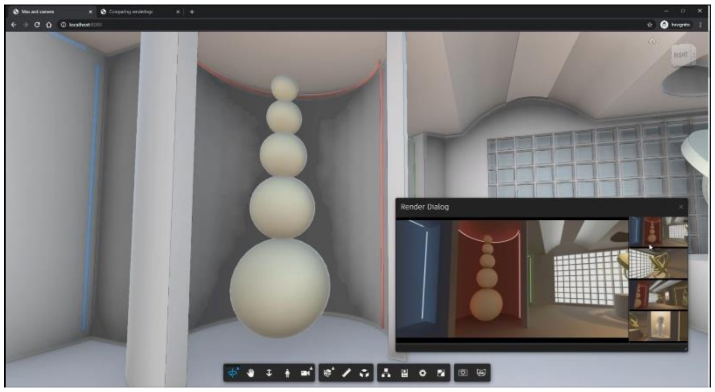
5. We submit the workitem with these inputs, along with OnComplete callback param, to be notified when the job is done.
6. Once job is done, we copy the results, save it as static content, through websocket session we inform about job completion and provide the static content address.

The task of rendering panorama is exactly the same except the input script we assemble has the following template: ***"renderPanoramaAtPoint {{.Posx}} {{.Posy}} {{.Posz}} {{.Width}}"*** For the panorama job, since we use the cubemap type, we send the camera position and just the width, which will be used to render a square image. Actually, for panorama, we have to render 6 images, for each side of the cube map. The render function from the bundle is also very easy (despite it being long):



```
--creates box images for panorama at give position and desired size
fn renderPanoramaAtPoint x y z size = (

    controller = dummy position:[x, y, z]
    cam = freecamera position:[x, y, z] fov:90
    cam.parent = controller
    rotate cam 90 y_axis
    rotate cam 90 x_axis

    -- FRONT --
    render camera:cam vfb:false outputsize:[size, size] outputfile:(sysInfo.currentdir + "/output/front.png")
    rotate controller 90 z_axis

    -- RIGHT --
    render camera:cam vfb:false outputsize:[size, size] outputfile:(sysInfo.currentdir + "/output/right.png")
    rotate controller 90 z_axis

    -- BACK --
    render camera:cam vfb:false outputsize:[size, size] outputfile:(sysInfo.currentdir + "/output/back.png")
    rotate controller 90 z_axis

    -- LEFT --
    render camera:cam vfb:false outputsize:[size, size] outputfile:(sysInfo.currentdir + "/output/left.png")
    rotate controller 90 z_axis

    -- TOP
    rotate cam 90 y_axis
    render camera:cam vfb:false outputsize:[size, size] outputfile:(sysInfo.currentdir + "/output/top.png")


    -- BOTTOM
    rotate cam -180 y_axis
    render camera:cam vfb:false outputsize:[size, size] outputfile:(sysInfo.currentdir + "/output/bottom.png")
)
```

With all these ingredients it should be a breeze to assemble the needed workflow and even if a rendering task can take some time, it is done asynchronously, without overloading the user machine with this task.

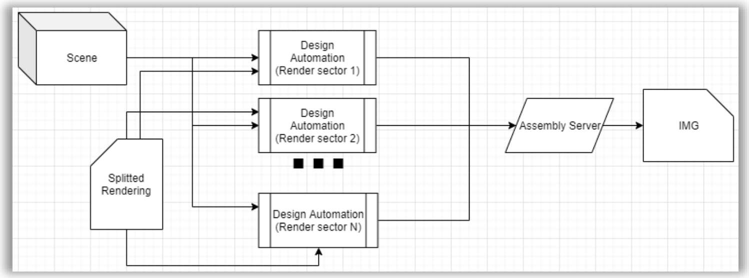


The user can continue navigate and explore the scene and come back when the results are ready, or you can decide to send him later by email a gallery of renderings he requested along his exploration.

Understand the strengths and limitations of using Design Automation for 3ds Max as a rendering solution

All this looks good on paper, but how about the reality and is it cost effective? The answer is a clear no.
This solution will never compete with a dedicated cloud rendering solution. In a dedicated cloud rendering solution, the workflow is optimized to balance the load among multiple workers. To simplify a lot, when a rendering task is submitted the job is split and distributed to dozens of workers. Some worker were lucky enough to finish faster maybe because there was not much to render in that chunk, then it gets another job from the pool and so on till all workers finish and then from this bunch of rendered results, the final rendered image is completed.

We could mimic this workflow with use of Design Automation service:


Scene
Splitted Rendering
Design Automation (Render sector 1)
Design Automation (Render sector 2)
Design Automation (Render sector N)
Assembly Server
IMG

but here comes the same problem we mentioned before when talking about rendering multiple images. This is not efficient because if we split the job in 100 chunks and use 100 of workers to do the rendering, each worker will need the copy of the scene to work with, so the scene will be copied 100 times and that’s a lot of traffic and a lot of time for just moving things around.

Another problem is the fact that in Design Automation, each worker is working in a sandbox environment, meaning that it cannot communicate with outside world and it means it cannot reuse some computations (some heuristics) made by another worker, so it will have to do it by itself, again. And all this just described in general lines.

A rendering task is a heavy job and a very good example of worst-case scenario, but for other scenarios it might be a perfect match. I hope that by now I convinced you that it is easy to setup and integrate into a pipeline, it allows you to offload your machine from repetitive tasks and the most important thing is that it is scalable. Be it dozens or hundreds of jobs, it can run all in parallel and you continue your work till the job is done and you need it.

The main conclusion is that, the Design Automation service is good for automating tasks, but we must be careful in what we automate, understand the limitations and weight the benefits of using it for certain tasks.