

Octxel Import

Leonard Sonnenberg

September 30, 2024

1 User Guide

1.1 Installation

To install the Addin, first copy the OctxelImport folder into the AddIns folder of Fusion. Depending on the OS, this is either

%appdata%\Autodesk\Autodesk Fusion\API\AddIns for Windows or

~/Library/Application Support/Autodesk/Autodesk Fusion/API/AddIns for Mac.

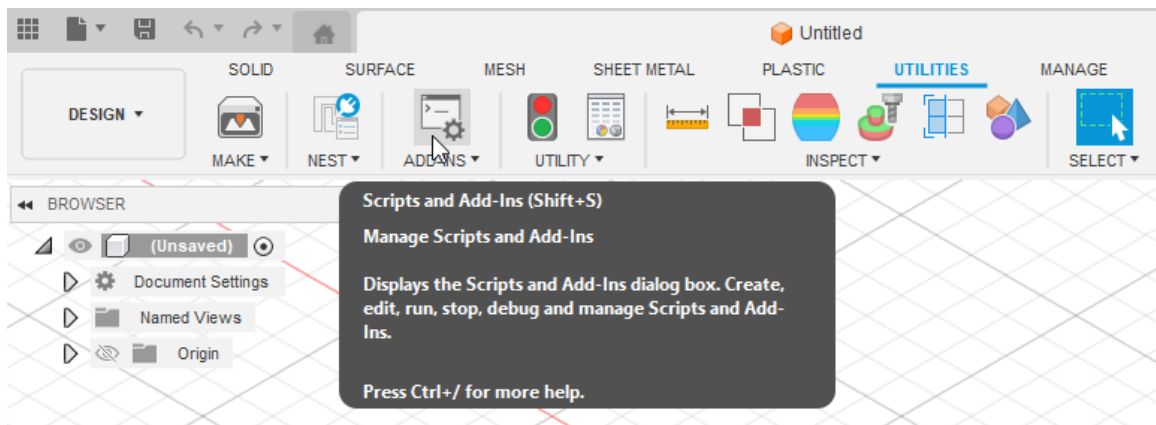


Figure 1: Location of Fusions AddIns Menu

To activate the Addin in Fusion, navigate to UTILITIES in the toolbar and click on the ADD-INS button (Figure 1). In the following menu (Figure 2), once again select Add-Ins and search the list for OctxelImport. Select it and click on the Run button to start the Addin. To automatically start the Addin whenever Fusion starts, tick the Run on Startup checkbox.

If there are issues or questions about the installation process, please visit the [Autodesk documentation](#).

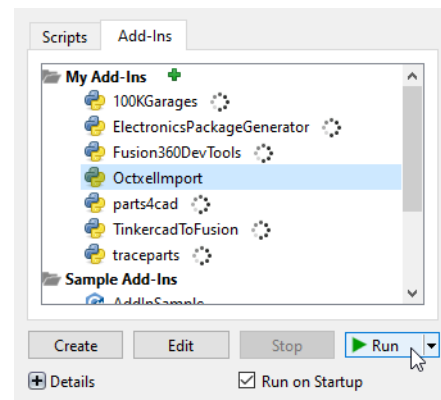


Figure 2: Fusions Addins Menu

1.2 Addin Usage

If the Addin started properly, there will now be three new options in the menu below the ADDINS button (Figure 3).

1.2.1 Import Octree json/octxel

This option opens a file dialog to choose one json or octxel file containing an octree voxel model to import into Fusion.

Be aware, the import may take a while for bigger models.

1.2.2 Json to Octxel

This option converts a selection of models in json form into the more compressed binary format octxel.

1.2.3 Octxel to Json

This option converts a selection of octxel files back into their json representation.

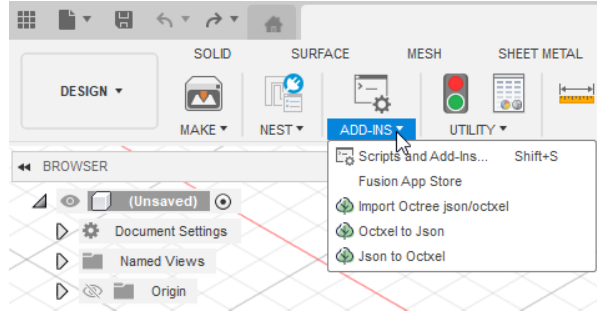


Figure 3: Octxel Menu

2 File Formats

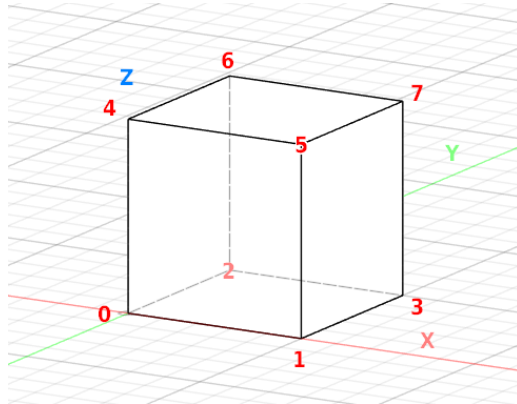


Figure 4: Octree Leaf Labels

The voxels are stored as an octree. The root of the octree is defined by a 3D coordinate and three extensions. Each voxel in the octree is described using a location String. This String is a path to follow from the root node to reach the represented voxel. The possible letters are the numbers 0 to 7, and each single letter represents the next smaller child node containing the voxel. If there is no letter left, the last selected child node is equal to the voxel. Figure 4 shows which number represents which child node.

2.1 Json Model

The json version stores everything in a single json object. The maximum depth (length of location) is stored as an integer with the key "depth". The size of the root node is stored as a String of three floating point numbers concatenated with semicolons with the key "rootExtensions". The location of the root node is stored as the center coordinates of the root node again as a String of three floating point numbers concatenated with semicolons with the key "rootCenter". The locations of the voxels are stored as a list of Strings with the key "leafNodes". The locations in the json version always start with a 0, which represents the root node.

```

1 # {
2 #   "depth": integer_depth,
3 #   "rootExtensions": "width;depth;height",
4 #   "rootCenter": "x;y;z",
5 #   "leafNodes": [
6 #     "0location_1"
7 #     "0location_2"
8 #     ...
9 #     "0location_x"
10 #   ]
11 # }

```

Listing 1: Json Model File Format

2.2 Octxel

Octxel stores a byte stream compressed using the LZMA algorithm. The stream starts with a 6 bit file version. After that, there are 6 * 64 bit double precision floating point numbers for the 3 corner coordinates and the 3 extensions of the root node in the order x, y, z, width, depth and height. Next follow blocks of voxel locations sorted by their depth, starting with 1 going up. Each block starts with a 32 bit unsigned integer that denotes the amount of voxels in that block. Since the location letters are only 8 in total, they can be stored in 3 bits each (000 = 0 to 111 = 7). So after the 32 bit amount, there are amount * depth * 3 bit of location information. Lastly, there are 0 to 7 bits of 0 padding to fill up the last byte.

```

1 # 6bit file version
2 # 64bit double root x position
3 # 64bit double root y position
4 # 64bit double root z position
5 # 64bit double root width
6 # 64bit double root depth
7 # 64bit double root height
8 # __ (depth 1)
9 # 32bit unsigned integer amount
10 # amount * 3bit voxel locations
11 # __ (depth 2)
12 # 32bit integer amount
13 # amount * 6bit voxel locations
14 # __
15 # ...
16 # __ (depth x)
17 # 32bit integer amount
18 # amount * x * 3bit voxel locations
19 # __
20 # 0-7bit padding to fill the last byte

```

Listing 2: Octxel File Format(LZMA compressed)

3 Code Documentation

3.1 OctxelImport

This is the main class of the Addin. The `cmd_definitions` list defines the commands and which classes represent those commands.

3.2 DecodeCommand

This is the main class of the Octxel to Json command. It utilizes the [Convert](#) class for user interaction and file conversion.

3.3 EncodeCommand

This is the main class of the Json to Octxel command. It utilizes the [Convert](#) class for user interaction and file conversion.

3.4 OctxelImportCommand

This is the main class for the import command. Beginning at line 22, we see the code that is executed once the import button is pressed.

```
22 app = adsk.core.Application.cast(adsk.core.Application.get())
23 # Create file browser dialog box
24 file_dialog = app.userInterface.createFileDialog()
25 file_dialog.filter = "Octxel (*.json;*.octxel);;All files (*.*)"
26 file_dialog.initialDirectory = os.path.expanduser("~/Desktop/")
27 file_dialog.isMultiSelectEnabled = False
28 file_dialog.title = 'Select Octxel file'
29
30 if file_dialog.showOpen() != adsk.core.DialogResult.DialogOK:
31     return # don't execute anything if no file was selected
32
33 file = file_dialog.filename
34 name = file.split('/')[-1].split('.')[0]
```

First, a file dialog is created and configured in the `file_dialog` variable. Then we can show the dialog with `file_dialog.showOpen()` to the user and compare the returned value to `adsk.core.DialogResult.DialogOK` to aboard the import if no file was selected. Lastly, we get the path to the selected file and extract the file name without the file ending.

```
11 def read_file(file):
12     if file.endswith(".json"):
13         return Json.decode(file)
14     else:
15         return Octxel.decode(file)
16
40 data = read_file(file)
41 if not data:
42     app.userInterface.messageBox(f"failed to open file")
43     return
44 root, voxels = data
```

In the next step, we extract the voxel data from the selected file. For this, we use the `read_file` function. This function looks at the file ending and then uses the `decode` function of the [Octxel](#) and [Json](#) classes from the [FileIO](#) package.

If the `read_file` function returns `null`, there was a problem with decoding the file, so we prompt a short notice to the user and end the import. Otherwise, we get the `root` and a list of `voxels` from the data.

```
46 progressDialog = app.userInterface.createProgressDialog()
47 progressDialog.cancelButtonText = 'Cancel'
48 progressDialog.isBackgroundTranslucent = False
49 progressDialog.isCancelButtonShown = True
50 progressDialog.show(f'importing {name}', 'Imported %v of %m Voxel (%p%)', 0, len(voxels), 0)
```

Before processing the data further, we create a progress dialog. The number of needed steps is set to the number of voxels in the voxel list.

```
52     component = futil.create_component(design.rootComponent, name).
    component
53
54     # Main Loop
55     profiles = {}
56     for voxel in voxels:
57         if progressDialog.wasCancelled:
58             break
```

We initialize a component with the name of the file to hold the new object we want to create. Additionally, we create a dictionary to hold the profiles we need to create the voxels. With those set, we can start the main loop, where we iterate over all the voxels. Before actually creating the voxel, we check whether the Cancel button of the progress dialog was pressed. If that is the case, we immediately leave the loop.

```
60         if voxel.depth not in profiles:
61             profiles[voxel.depth] = futil.create_profile(
62                 component.sketches,
63                 component.xYConstructionPlane,
64                 root.depth * (1.0 / 2 ** voxel.depth),
65                 root.width * (1.0 / 2 ** voxel.depth),
66                 voxel.depth
67             )
```

To create a body, we need to extrude a profile. Since all voxels of the same depth can use the same profile, we use the `profiles` dictionary to map the `voxel.depth` to a profile. If there is no profile associated with the depth of the current voxel, we use the `create_profile` function and add the resulting profile to the dictionary. The width and depth of the profile are calculated using the voxels depth and scaled by the roots width and depth.

```
69         futil.create_cube(
70             component.features,
71             root.depth * voxel.x + root.x,
72             root.width * voxel.y + root.y,
73             root.height * voxel.z + root.z,
74             root.height * (1.0 / 2 ** voxel.depth),
75             profiles[voxel.depth]
76         )
```

Next, we can use the `create_cube` function to create the voxel body and move it to the intended position. Since the coordinates of the voxel are relative to the root, we need to scale them with the root extensions and shift them by the root coordinates. The height of the voxel is calculated using the depth of the voxel and scaled by the height of the root.

```
78         if progressDialog.progressBarValue % 100 == 99:
79             futil.combine_all(component)
80             progressDialog.progressBarValue += 1
```

To prevent Fusion from slowing down too much, we need to keep the number of bodies low. Therefore we check the `progressValue` and every 100 steps we use the `combine_all` function to combine all bodies into one. As the last step of the loop, we increase the `progressValue` by one and then continue with the next voxel.

```
82         futil.combine_all(component)
83         for feature in component.features:
84             feature.dissolve()
85         for body in component.bRepBodies:
86             body.name = name
87         progressDialog.hide()
```

After the loop was canceled or finished naturally, we combine all resulting bodies. Then we dissolve the remaining features and give the leftover body the name of the file. With this, the import is finished, and we can hide the progress dialog.

If there are issues due to Fusion using too much RAM after an import failed, this could be due to the dissolving of leftover features. During testing, we found that dissolving a lot of features via the Fusion API can lead to a rapid increase in RAM usage. During a normal import, that should not pose an issue since, after the combine, there should only be one feature left to dissolve.

3.5 FileIO

This package takes care of all file reads and writes and the needed encodings and decodings to translate the voxel data from and to the file.

3.5.1 BitIO

Since Python only allows byte-wise reading and writing of files, this helper class buffers written and read bits until a full byte is reached. Additionally, it uses Python's lzma library to compress and decompress the binary data.

3.5.2 Convert

The `file_dialog` function of this class offers a generic way to prompt the user a file dialog and convert all selected files from one file format to another. To achieve this, the function takes a decode and an encode function as arguments. These are used to get the data from the files and then write it back to the new files.

3.5.3 Json

The class responsible for json reads and writes. It uses Python's standard json library. More information can be found in the [json model file format](#) section. Before initializing the voxels, we sort them based on the location String. This makes sure that intermediate combine steps will lead to a single body.

3.5.4 Octxel

The class responsible for octxel reads and writes. It uses the [BitIO](#) class and the struct library to read and write binary data from and into the file. More information can be found in the [octxel file format](#) section. After creating the voxels, they are sorted based on the location String. This makes sure that intermediate combine steps will lead to a single body.

3.6 Fusion360Utilities

Utility package containing helper classes to interface with Fusion.

3.6.1 Fusion360CommandBase

Base class for Addin-commands provided by Patrick Rainsberrys [Fusion360Utilities Package](#).

3.6.2 Fusion360utilities

This class contains utility functions to create and manipulate Fusion objects.

`create_component`

This function is taken from the [Fusion360Utilities Package](#) by Patrick Rainsberry. It creates a new component with a given name in a given parent component.

`combine_feature`

This function is taken from the [Fusion360Utilities Package](#) by Patrick Rainsberry. It creates a combine feature taking a target body, a list of tool bodies, and an operation type.

create_cube

We use this function to create the voxel body. It takes a feature list that is needed to create new features, the coordinates of the new body as x, y and z, the height of the cube and a profile describing the base of the cube as arguments.

To create the cube, we first create an extrude feature using the profile and the height. Then we check whether a move is necessary. If any of x, y or z are not 0 we create a move feature, moving the extrusion to its intended position.

create_profile

This function is used to create a rectangular profile. It takes a sketch list and plane needed to create new sketches, the width of the profile, the depth of the profile and a depth value used for naming the sketch as arguments.

First, a new Sketch is created with the name of the depth, and the visibility is set to invisible since it is just a helper object. Then we draw a rectangle at the origin with the given width and depth.

combine_all

This function simplifies the usage of the [combine.feature](#) for the voxel context. It takes a component as the argument and combines all bodies inside the component into one body.

For some reason, trying to combine all bodies sometimes fails. To prevent that from negatively impacting the import, we use a try-catch clause and for cases where the combine failed we first combine all bodies except the first and then combine the remaining bodies. This produces the same body the failed combine should have created, using one intermediate step. During testing, this seemed to always work as a fix.

3.7 Octxel

Package containing data classes for holding the octree voxel data.

3.7.1 Octxel

The objects of this class represent a voxel. It takes a location String when initialized and calculates the depth and relative coordinates of the voxel inside of the octree root.

3.7.2 Root

A simple class for a data object holding x, y, z coordinates and the width, depth and height of the octree root.

4 Acknowledgements

The code contains parts of the [Fusion360Utilities Package](#) by Patrick Rainsberry.

The tree icon was provided by [icons8](#).