

Session 3.3

The Architect's Dynamo: Getting Non-Revit Users Involved!

Paul F. Aubin, www.paulaubin.com

Zach Kron, Autodesk

Class Description

This class is designed to show architectural offices how they can utilize Dynamo to involve their non-Revit users in a Revit environment. The class will cover how Revit can extract data from external software for use in manipulating an architectural building design. Let's do some Dynamo!

About the Speakers:

Paul F. Aubin is the author of many Revit book titles including the widely acclaimed: The Aubin Academy Series, Renaissance Revit and Revit video training at www.lynda.com/paulaubin. Paul is an independent architectural consultant providing Revit® for Architecture implementation, training, and support services. Paul's involvement in the architectural profession spans over 25 years, with experience in design, production, CAD management, mentoring, coaching and training. He is an active member of the Autodesk user community, an Expert Elite and is a top-rated repeat speaker at Autodesk University, Revit Technology Conference and Midwest University. His diverse experience in architectural firms, as a CAD manager, and an educator gives his writing and his classroom instruction a fresh and credible focus. Paul is an associate member of the American Institute of Architects and lives in Chicago with his wife and three children.

Zach Kron is a Senior Product Manager at Autodesk, focused on Dynamo. Since 2007 he has researched, strategized and helped implement parametric design tools and workflows. In addition to internal teaching and curriculum development at Autodesk, Zach has helped create and run workshops at Massachusetts Institute of Technology, the Association for Computer Aided Design in Architecture, Autodesk University, and many other venues. Before joining Autodesk, Zach worked as a designer in several Boston-area architectural offices on projects ranging in scale from furniture to bridges. He has more than 16 years of professional experience in design.

Introduction

In this handout, I have detailed one of the workflows that we will be discussing in the live session. There will be other workflows discussed that will not appear here in the paper. We are treating the paper as a follow-up resource. This will allow you to revisit the logic as you read through this detailed workflow and either attempt to recreate it, or simply apply the logic to your own needs and workflows back at the office.

Data Sharing with Excel

In the following scenario, we will look at using Dynamo to access a collection of data stored in an Excel file and use that data to create Revit elements. Specifically, the Excel file contains a building program with a list of each type of room required, the quantity of rooms needed for each type and their required areas. We will read this list into Dynamo, process the data as required by Dynamo and use it to create Revit elements (both 3D space-planning blocks and Revit room elements).

Read an Excel file

The first step required is to access our Excel file. Here is a look at what the file contains:

Name	Qty.	SF Each	Space Type
Enclosed Executive Office	0	180	Office Spaces
Enclosed Large Office	0	150	Office Spaces
Enclosed Small Office	0	120	Office Spaces
Open Large Office	4	180	Office Spaces
Open Small Office	15	120	Office Spaces
Open Workstation	100	80	Office Spaces
Reception Desk	1	80	Office Spaces
Reception Seating	1	120	Support Spaces
Conference Large	1	600	Support Spaces
Conference Small	5	150	Support Spaces
Informal Breakout Center	12	80	Support Spaces
Printer/Copier/Fax Center	3	80	Support Spaces
Break Room Service Unit	1	340	Support Spaces
Information Reference Center	3	180	Support Spaces
Supply Center	4	40	Support Spaces
Work Center	1	200	Support Spaces
File Area	2	144	Support Spaces
Documents Room	1	240	Support Spaces
Server Room	1	176	Support Spaces

Dynamo – the “basic” basics

If you have never used Dynamo before, here are the bare essentials: you use the library pane on the left to locate nodes (browse or search). These nodes contain predefined snippets of code that perform a single discreet action. Nodes have inputs on the left and outputs on the right. Click on any port (input or output) to create a “wire.” Attach this wire to other ports to wire up a graph. Graphs and data flow and read from left to right with the output of one node feeding the input(s) of the next. Use your wheel mouse to navigate.

We will access the Excel information using the **Excel.ReadFromFile** node. This node is on the *Office* branch in the library (see Figure 1). It requires three inputs. The **file** input points to the actual file on the hard drive. The **sheetName** input tells Dynamo what sheet in your Excel file to read and the **readAsStrings** input is a toggle switch. If you toggle it on, it will convert all of the incoming data to text (strings), if you leave it toggled off, it will not change the data at all and read it in as-is.

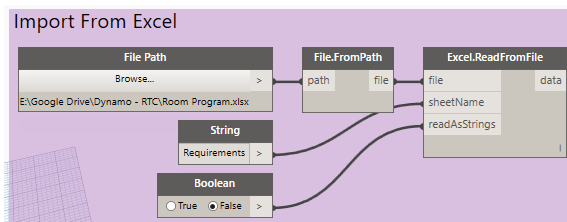


Figure 1

To pass in the path name, we need two nodes: **File Path** has a browse button that you can use to locate the Excel file. To convert this to the proper format required by the **Excel.ReadFromFile** node, use a **File.FromPath** node. (I don't know why converting the file path is required, but if you don't want to get an error, make sure you use both nodes).

The **sheetName** input accepts string (text) data. So a **String** node is appropriate. You can type anything you want into a **String** node. Simply type in the name of the Excel sheet that contains your data. This is case-sensitive, so type carefully.

*Many would argue that a **Code Block** node would be a better choice here, but please don't get me started on that...*

Many node inputs have default values. The **readAsStrings** input is one such input. If you don't want the data converted to text as it is input, leave this toggle set to its default of "False." To force all input data to be converted to text (strings), use a **Boolean** node and choose the "True" option. As you can see, you can also use the **Boolean** node regardless and leave it set to "False" as well if you wish.

When you finish wiring up these nodes, if execution of your graph is set to Automatic (see Figure 2), it will immediately launch Excel and load up the spreadsheet you indicated. (If Excel launches and loads your document, then you wired everything up correctly).

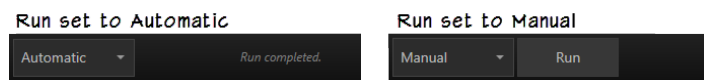


Figure 2

If this does not happen, check your wires and the spelling of the Worksheet name in the **String** node.

It is not a bad idea at this point to select all of the nodes, press CTRL + G to group them and then double-click on the name and type in a description. This helps keep the graph organized and provides a degree of documentation for future reference.

Process The Data

Now that we have the data loaded into Dynamo, it is time to begin processing it. For starters, most Excel data will contain headers. You will typically want to strip those out. This is easy to do with the **List.Deconstruct** node. This node will remove the first item from a list and output it to one port, and then output the rest of the items on the list separately on another port (see Figure 3).

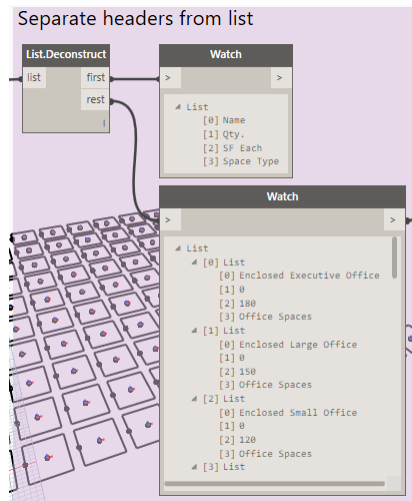


Figure 3

There is a single input: **list**. This gets connected to the **Excel.ReadFromFile** node output. To check the data coming out of the **List.Deconstruct** node, connect a **Watch** node to each output. **Watch** nodes display the results of any output. Take a look at the format of the data coming out of the Excel file in the **Watch** nodes.

First, these lists are numbered using a zero-based numbering system. So the first item on the list is number 0. This is how all lists in Dynamo are formatted. So get used to it! Second, if we study how the data is organized, we see that we have a list which contains other lists. Each sub-list contains one row of data from Excel. So the data is read row by row, *not* column by column. Sometimes you will prefer the data to be formatted by columns instead of by rows. This is easy to accomplish with the **List.Transpose** node (see Figure 4).

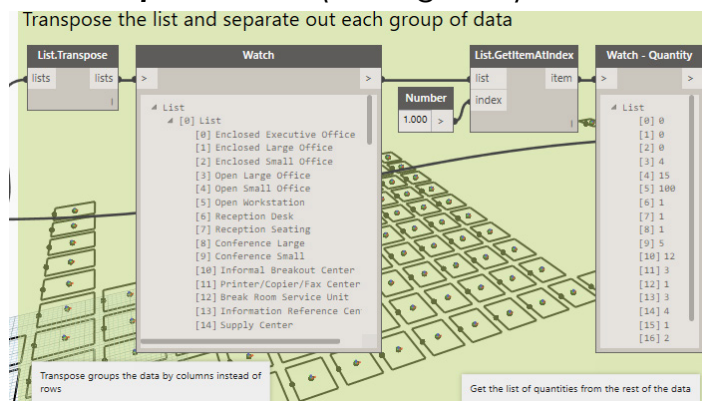


Figure 4

So feed the output of the **rest Watch** node into the input of the **List.Transpose** node and the result is still a list of lists, but they are now reversed to show the data by column instead of by row. From these lists, we want to grab the quantities of

rooms required. Scrolling through the **Watch** node we see this is the second list (which is index [1]).

A **List.GetItemAtIndex** node will let us pull out a single item from the list. This is done by feeding in a **Number** node (yes **Number**, but if you really want you can use a **Code Block**) into the **index** port. Feeding this to another **Watch** node gives us a nice tidy list of just the required quantities.

Repeat Items

So now that we have separated out our quantities, we can use them to create a new list that contains the desired quantity of each type of space. The node for this is called: **List.OfRepeatedItem** (see Figure 5). It has two inputs: **item** (this is the item you want repeated) and **amount** (this is how many times you want to repeat it).

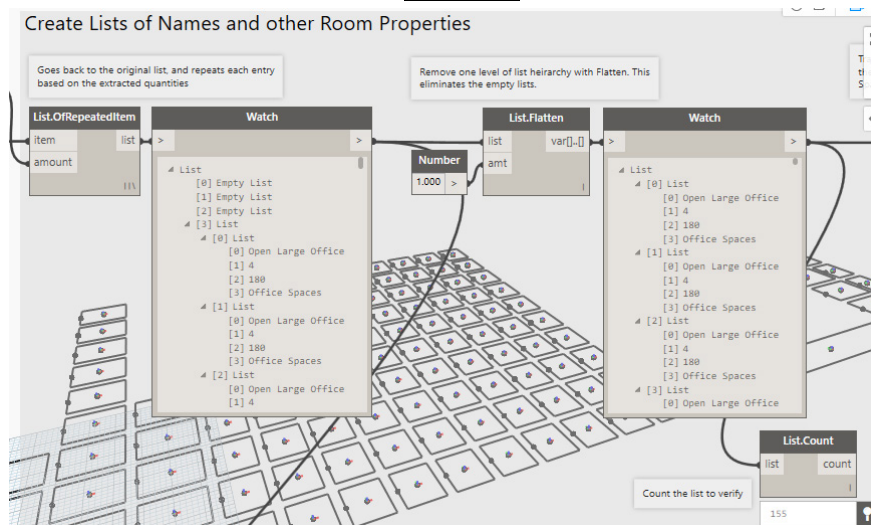


Figure 5

So we'll feed the **rest** output from the original list of items (from Figure 3 above) into this node. Then we'll feed the quantities we extracted in Figure 4 into the **amount** input. A **Watch** node here will reveal a list of lists. However, we now run into an issue of data matching. When you have two datasets coming together, Dynamo must decide how to interweave the information. This is called: "Lacing." The default is: "Shortest" which matches the first item with the first item and on down the list throwing out any extras. "Longest" will match up the extra items by matching some items more than once. There is also "Cross Product" which finds every possible combination. In this case, Longest will give us the result we need. So, right-click on the **List.OfRepeatedItem** node and choose: **Lacing > Longest**. Once complete, we will have a list for each type of space required, and then nested sub-lists for the repeated items within each list (see the left side of Figure 5).

Take a close look at this data. Notice that there are some “Empty Lists.” If you look back at the quantities, these are the items whose quantity was zero. We can remove these and the top level of lists with a **List.Flatten** node. Here you input the list and a **Number** (with value of: 1) to tell it how many steps of hierarchy to remove (see the right side of Figure 5).

Now we have a nice tidy list containing each required space. A **List.Count** reveals that there are 155 total items. Sounds reasonable from the original data... Cool!

Extract Required Values

We will eventually want to create rooms and place families that have the properties from the original Excel file. So using the same technique covered above in Figure 4, we can extract the: Room Names, Required Areas and Space Types from the Excel data with **List.Transpose** and **List.GetItemAtIndex** (see Figure 6).

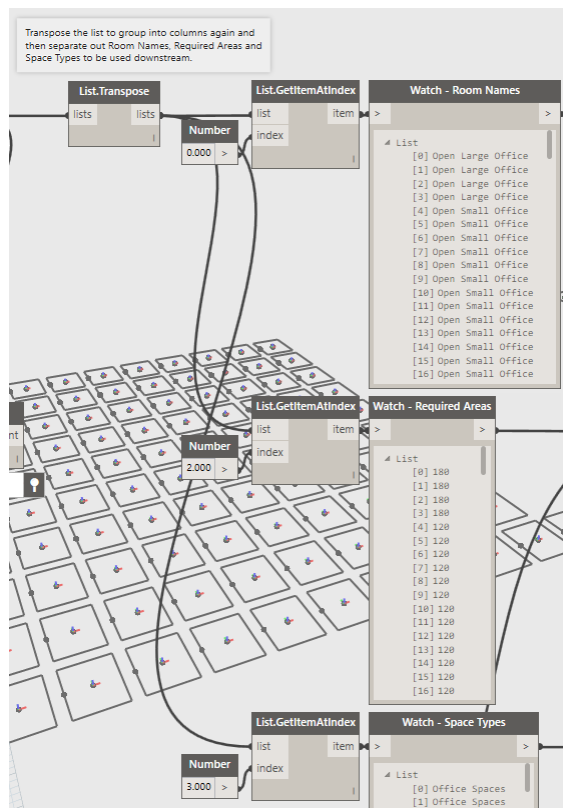


Figure 6

You will notice that the names of the Watch nodes have been edited. To do this, simply double-click on a node title and edit the name. I like to leave the original title in the new name just to remind myself what kind of node it was originally. But this is just my preference. You can name the nodes anything you like. Other ways to document your graph include grouping the nodes (as suggested above) and adding notes. To group nodes, select several, and then press **CTRL + G**. To add a note, press **CTRL + W**.

Chop the lists

Let's go back to processing our list. Above we created a list of all of the rooms required based on the quantities from the Excel file. Our next step is to break this list up into more manageable pieces. Ultimately, we want to have a series of shorter lists that will be used to place these elements in a Revit file in a logical way. For example, how about placing items in a grid pattern where the columns are the kind of room required and the rows are the quantities. However, since some of our quantities are large, we also want some logic that says if the quantity exceeds a certain amount, to break it into more columns (see Figure 7).

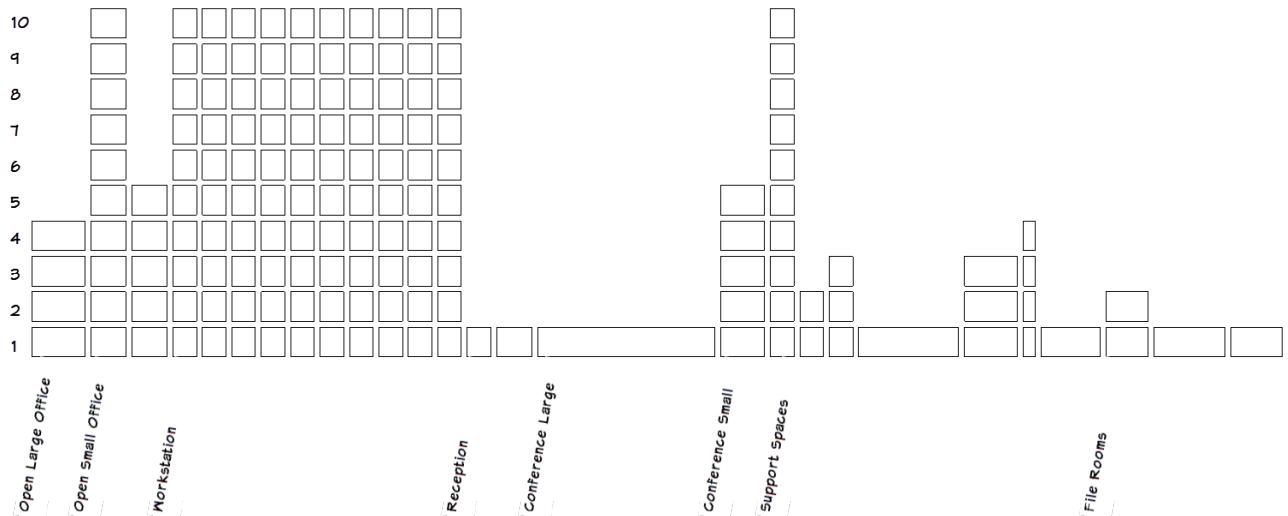


Figure 7

I settled on a quantity of 10 as the maximum number of rows. This gets fed into a **Number** node. (Note that I renamed the node to indicate that it controls the Maximum List Size, see the left side of Figure 8). To make the lists, we use **List.Chop**. This is sort of like the opposite of Flatten that we used above. **Flatten** reduces the

hierarchy in the lists where **List.Chop** adds hierarchy to the list creating a list of lists. Here the main list will be a list for each column and then the sub-lists will contain each of the rows.

You will notice that in Figure 8 there is also a **List.Map** after the **List.Chop**. So what does this do and why do we need it?

List.Map allows you run an operation on the nested list instead of the top level list. So why not just run the **List.Chop** on the flattened list from above? Well remember that we want to preserve the hierarchy and have a column of items for each type of room required. If we use the flattened list, we would just get 10 items per column with no break per room type. With the **List.Map**, we get a column of rooms of the same type up to ten. If there are more than ten, then **List.Chop** will give us a second column of up to ten and so on. For example, back in the diagram shown in Figure 7, there are several columns of Workstations, each with a maximum of ten items.

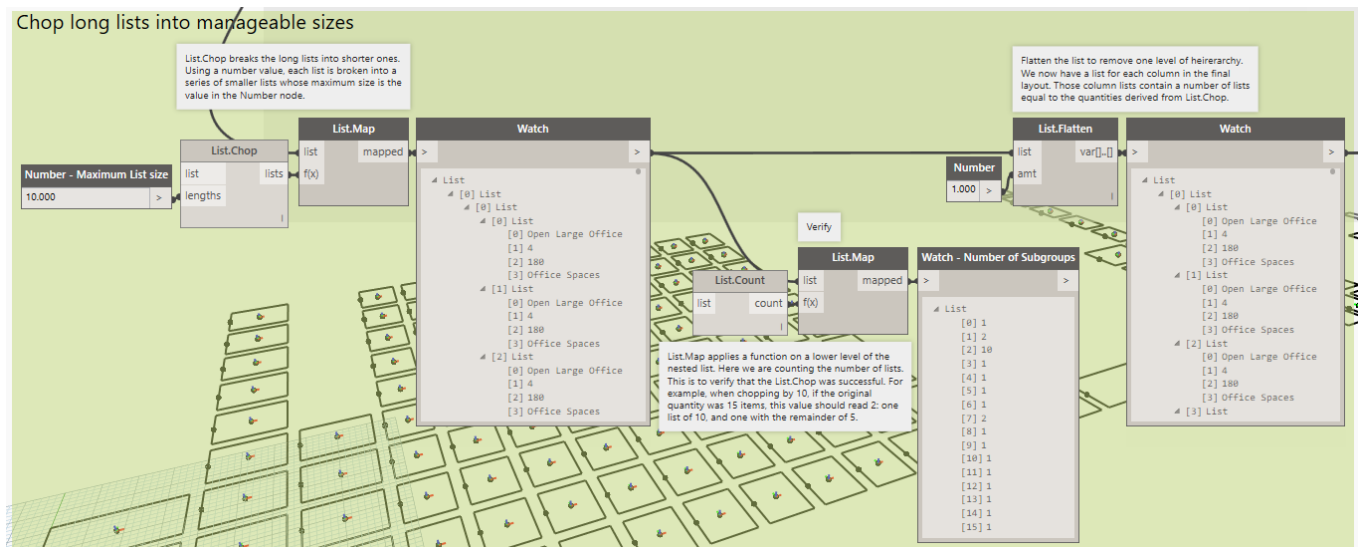


Figure 8

Almost there. On the right side of Figure 8 we first have a **List.Count** just for verification purposes. This is applied to the nested list using another **List.Map**. This helps to ensure that **List.Chop** did its job as expected. (So, item [1] has a value of 2 and item [3] a value of 10. This means that we will end up with two columns of Open Small Offices and ten columns of Workstations). Beyond this, there is another flatten because List.Chop gave us an extra level in the hierarchy that we don't need. There are two flatten nodes in Dynamo: **Flatten** and **List.Flatten**. **Flatten** will remove *all* hierarchy from the list giving a completely flat list. **List.Flatten** has an amt input (which takes a number) and allows you to designate how many levels of hierarchy to remove. This is more flexible and works well in this case.

Determine how many columns and rows in the Layout

The final list in the previous step contains 27 lists of lists. The quantity 27 was determined by **List.Chop**. So we need 27 columns. The number of rows in each column will vary by how many sub-lists each column list contains. If all we needed was this quantity, a simple **List.Count** would do the trick. But we will also be setting parameters on Revit elements, so we need to extract other information such as the required area of each room. So we'll take a different approach.

List.FirstItem takes the first item from a list. If we use it with another **List.Map**, we can apply it to the nested lists and get the first item from each of the 27 nested lists! That first item in this case will be another list that contains the name, required quantity, required area and space type for each kind of room. (This is data that came from the original Excel file).

Using one more **List.Map** combined with a **List.GetItemAtIndex**, we can ask for a list of all of the required areas for each of our 27 columns. All we need to do is use a **Number** node to feed in the appropriate index value; [2] in this case (see Figure 9).

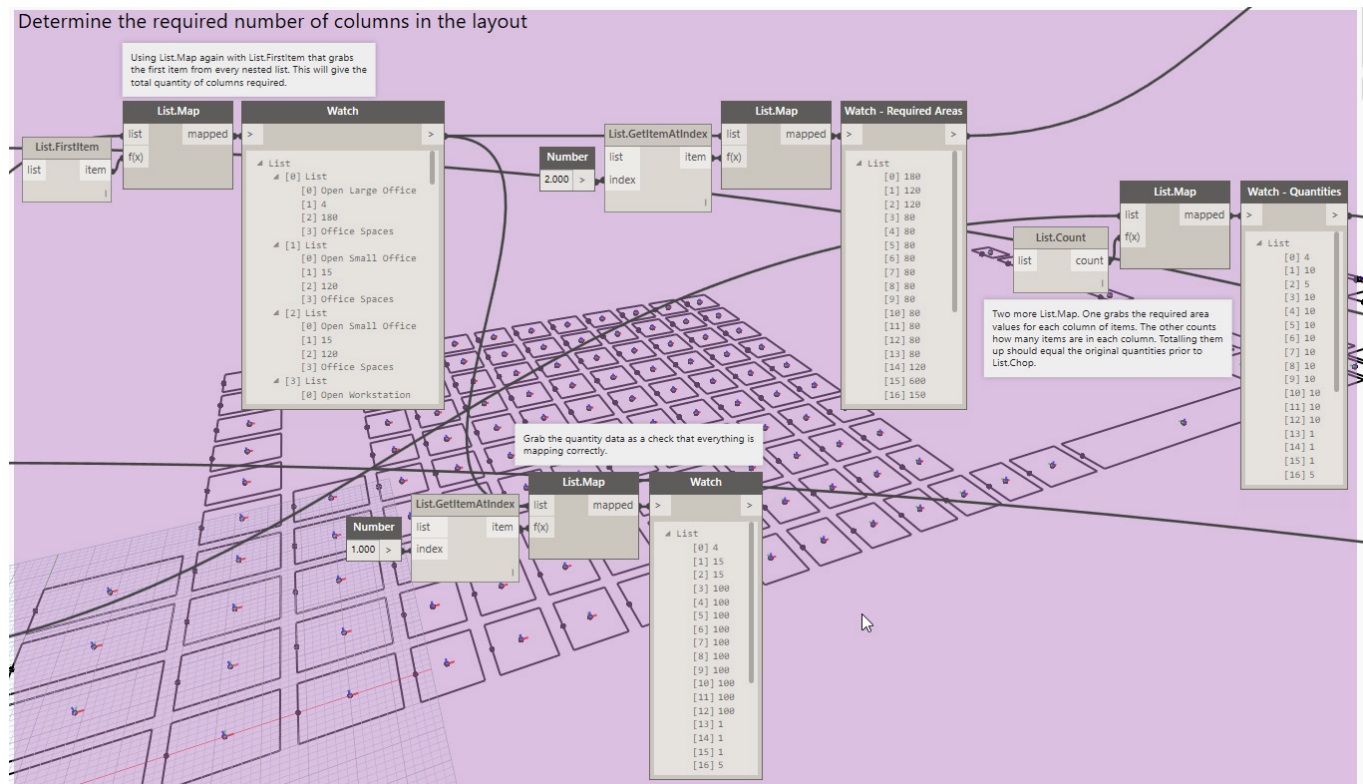


Figure 9

As you can see in the figure, I also grabbed index [1] as a check to make sure that everything is mapping properly. This is optional of course, but it is helpful to do these checks along the way to be sure everything is working. The final group of nodes on

the right side in the figure goes back to the original chopped list and using another **List.Map** with **List.Count** tells us how many rows are in each column.

Create the Layout

Now that we have imported, sliced and diced the Excel data, we are ready to use it to begin making Revit elements. In this example we will place a series of families using the data from above and also, taking advantage of some custom nodes we will place room elements as well. Let's begin with some inputs and an assumption. The original Excel file gave us the required area, but did not say the width and depth of the rooms. So for simplicity, we will simply divide each required area by a constant value. This will be one of our inputs. The other input will be just a constant value to add to each item to provide a little padding between the elements as they are created (see Figure 10).

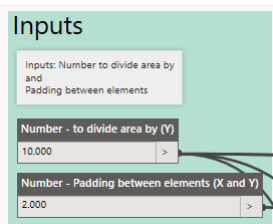


Figure 10

I used **Number** inputs for both of these. If you want something more interactive, try a **Number Slider**. The number that I am dividing into the areas I am using as the Y dimension. That means that all elements will be the same height and we will calculate the widths (X). The padding value will be used in both directions.

Performing Calculations

Simple math is easily performed with nodes in the Operators branch (see Figure 11).

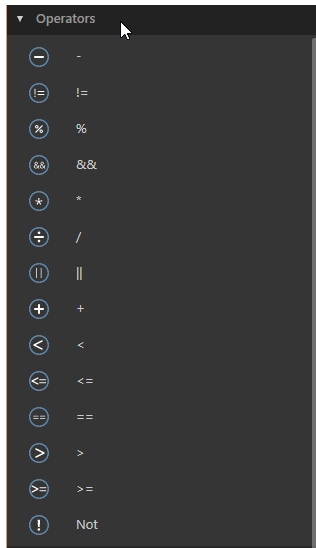


Figure 11

For example, to calculate our X value, we go back to the list of required areas (Figure 9) and feed it into the **x** input of a / (division) node and feed the **Number** node containing our constant from Figure 10 above into the **y** input. Then using a + node, feed this result into the first port and the padding value constant into the other port. A **Watch** will help verify the results (see the left side of Figure 12).

*Notice that the operator nodes use the generic names of **x** and **y** for the inputs. These can take any kind of input. The names "x" and "y" are just variable names like you had in algebra class...*

oops, sorry for the painful flashback...

Calculate the X Values

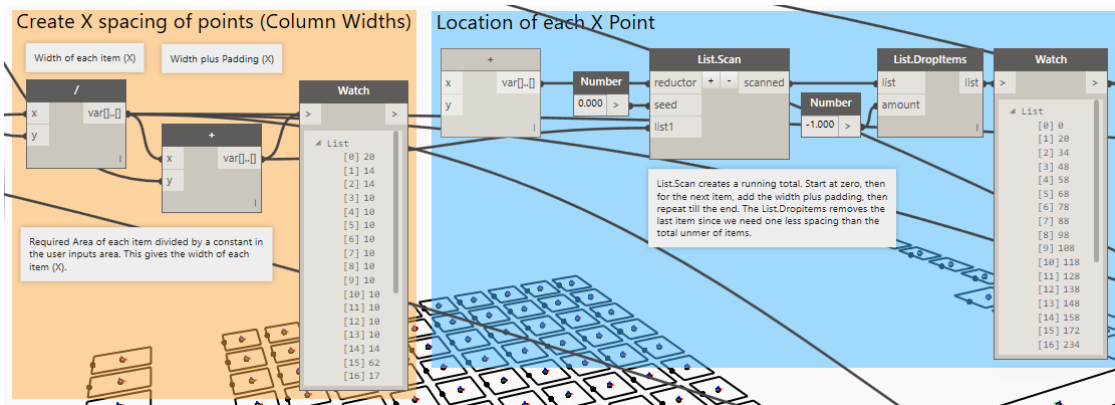


Figure 12

In the X direction, we need a “running total” of all the X values. Remember that above we determined that we needed 27 columns. The first column will start at zero. The next column needs to start at a distance equal to the width of the first column plus the padding constant. Then the column after that needs to add the next width plus the padding and so on. This can be accomplished with the **List.Scan** node. **List.Scan** will start at a number fed into the seed port. Then using whatever operator you feed into the reducer port, it will build a running list from the values fed into the list port. Since we want to do a running total, we will feed in a **+** node into reducer and the list of values from the **+** node calculated earlier.

Since we are starting at zero, we will end up with one item too many. So a **List.DropItems** with a value of **-1** will drop the last item on the list. (A positive number input will delete that quantity from the start of the list, a negative number deletes the items from the end of the list).

Calculate the Y Values

This gives us our X values for our points. Onto the Y values next.

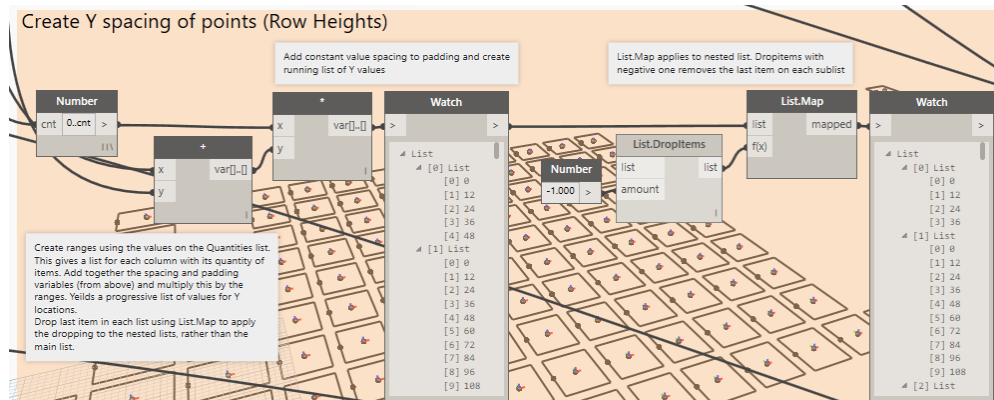


Figure 13

Initially, the Y values seem as though they will be simpler. Since we are using a constant value for Y, just add this to the padding and then repeat for as many items as needed right? Well, since we have a different quantity of items in each column, we have to perform this calculation several times and at different quantities of rows. So let me sneak in a little shorthand here. In Figure 13, notice that the first node is a **Number** node, but that it looks a little different than other **Number** nodes. Typically, Number nodes only have outputs, but this one has an input labelled cnt (see Figure 14).

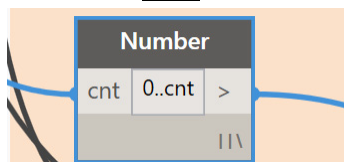


Figure 14

Also take notice of the value typed into this node. It is: **0..cnt**. That is zero dot dot cnt. When you use the dot dot nomenclature, you are indicating that you want a range of values. So **0..10** would yield a list of values from zero to ten. By typing in: **cnt** instead, (short for "count") this becomes a variable and creates an input. (It does not have to be: cnt; you can type any variable name you wish). By making it a variable, you are saying I want a range of numbers between zero and some variable fed into this input. This way we can feed our list into it and get our quantities from elsewhere in the graph. Cool right?

OK, so we have a range from zero to some variable amount. What can we do with that? Looking back at the **Watch** node in Figure 9 that gave us our quantities required for each column, we will feed this list into the cnt input. This will give a list

of lists whose quantities vary based on the values in that previous **Watch** node. So list zero will have values from zero to four and list one will have values from zero to ten and so on. But to convert these sequences of numbers to actual Y values, we then simply multiply them by our constants from Figure 10 above (see the **+** (addition) and ***** (multiplication) nodes in Figure 13).

One last step to complete all of the required Y values. We now have 27 lists of Y values, but each one contains one item too many. So using **List.DropItems** with a value of **-1** again will remove the last item. However, we want the last item removed from each sub-list, not the main list, so **List.Map** to the rescue again!

Create Points

With our X and Y values now calculated, we are ready to create points. A **Point.ByCoordinates** node will work nicely here. Simply feed in the X and y values calculated above into the appropriate ports. You can leave the **z** port empty and it will default to zero (see the left side of Figure 15)

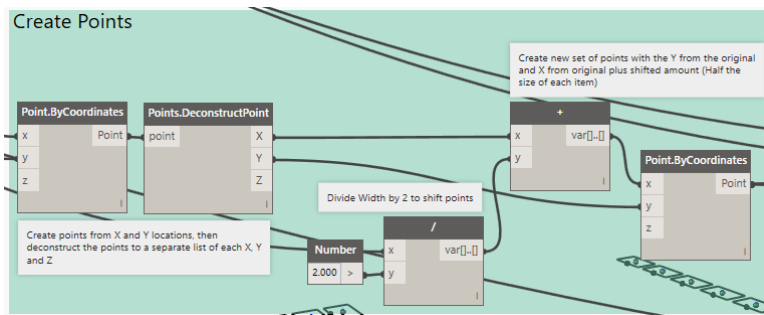


Figure 15

By default, placing family instances and later rooms will end up centering the elements on the points created here. I preferred to shift everything over to end up with the equivalent of “left justified.” The rest of the nodes in Figure 15 accomplish this. Using **Points.DeconstructPoint**, the X and Y values are extracted. I then went back to the original widths calculated for each item and divided it by 2. I then added this result to the X values and passed the Y values unchanged back into another **Point.ByCoordinates** node. The result is two sets of points.

Placing Family Instances

With all of the points in place, you can now create geometry at those points. You can create Dynamo geometry that exists only in your Dynamo graph, or you can use the Dynamo geometry and points to create Revit elements. This is the point where your non-Revit users could leverage the power of Dynamo to process data

from other programs and use it to interface with the Revit team. For this next portion of the example, I created a simple generic model family. It contains a box with adjustable length, width and height. We will leave the height alone and focus on only the length and width. Using the values in our graph so far, we can place and size instances of this family in the model to represent each required room. This could be useful for the space planning team to visualize and create 3D bubble diagrams. I have already included the family in the sample file. So we are just placing it here. The **Family Types** node gives a list of all loaded families in the project. Choose the family desired from the drop-down. Feed this to the **familyType** port of a **FamilyInstance.ByPoint** node. For the **point** port, use the output from the second **Point.ByCoordinates** node.

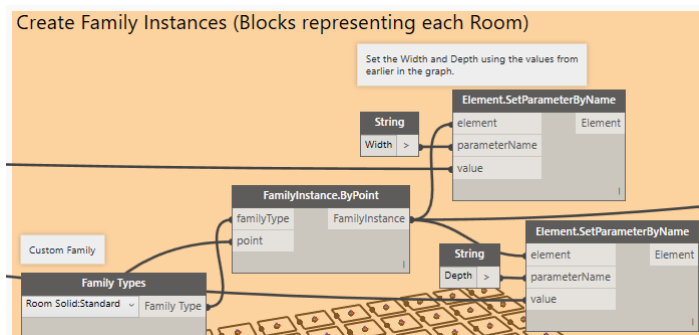


Figure 16

Sizing each Family Instance

When completing the previous step and running the graph, you will have an instance of the family placed at each point, but they will all be sized using the default values in the family. To size each family to the sizes required by our original list of data, use two **Element.SetParameterByName** nodes. This node and the similar **Element.GetParameterByName** node are two of the most useful and frequently used Revit nodes. The “get” node reads parameter values from elements in your model and the “set” node writes values to the elements. So you can use these nodes to interact with the elements in a model and have Dynamo apply the appropriate parameter values. All you have to do is indicate which parameter by feeding its name into the **parameterName** port. Do this with a **String** (or, if you must, a **Code Block**) node. Be sure to type the parameter name exactly as it appears in Revit. This is case-sensitive. So here we just use one **Element.SetParameterByName** node to set the Width and another to set the Depth. The values for Width come from the original calculation made back in Figure 12 (before adding the padding). The values for Y come from the constant (in the inputs area in Figure 10) again without the padding.

Packages

If you want to create rooms instead, or in addition to the families already placed, you will need to download some custom nodes. Dynamo is “open source” this means that its source code is publically available and this also means that there is a vast community of members working to create custom nodes. Custom nodes are grouped into “packages” and are free to download. To access them, use the Packages menu. You can search for a package and install it and it will add a branch to your Dynamo library. Just be aware that if you build a graph that uses a custom node, you will want to communicate this to anyone you share the graph with so they can download the required packages as well. Without the required packages installed, any custom nodes in the graph will not function.

Creating Rooms

There are some very popular packages available that contain custom nodes for doing all sorts of things in Revit. We will showcase nodes from two of these packages here: *Clockwork* and *Steamnodes*.

The first custom node is from the Clockwork package and lets us create room separation lines. With Clockwork installed, search for “room” and you will have access to the **RoomSeparation.fromCurve** node. Steamnodes gives you the **Tool.CreateRoomAtPointAndLevel** node. You can often double-click these package nodes to look inside (the ones that have a titlebar that looks like a stack of paper). But they often use Python code inside. So unless you know Python, it will look like so much colorful gibberish. Best to just not look...

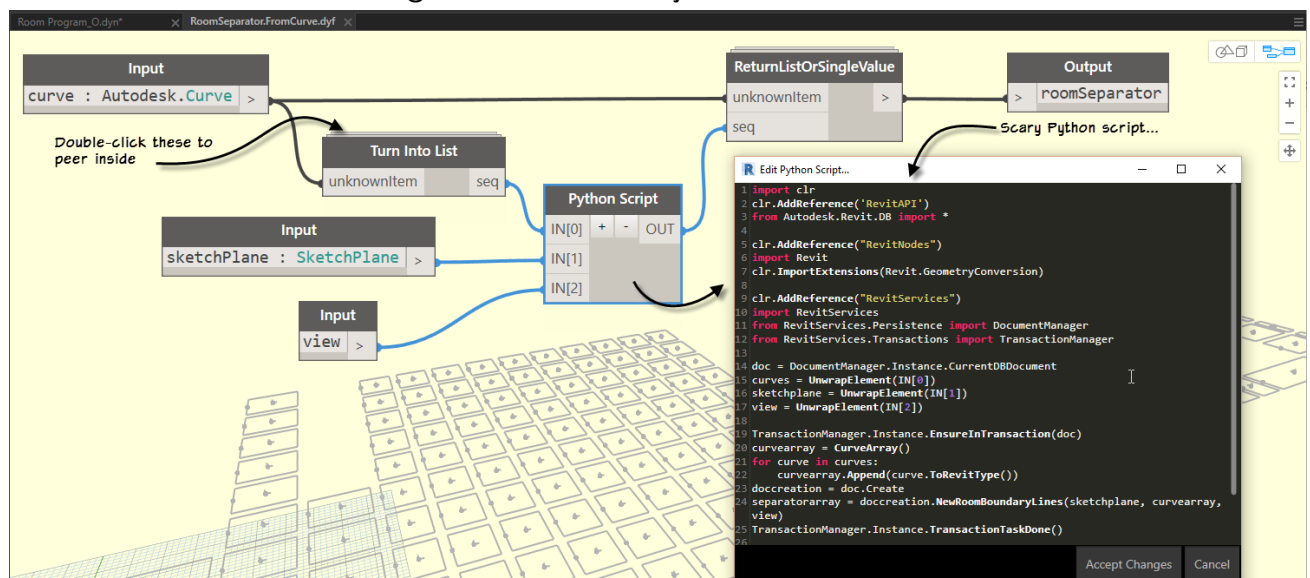


Figure 17

Since rooms need boundaries, the strategy outlined here will be first to create room separation lines, then create rooms within those boundaries.

Create Room Separation Lines in Revit

Let's work backwards through Figure 18. The **RoomSeparation.fromCurve** node has three inputs: curve, sketchPlane and view.

For the view input, the easiest thing to do is use the **Views** node. This allows you to select from a list of views in the project.

The sketchPlane input is a little puzzling; I would think the view would automatically assign a sketchplane, but the easiest thing to do here is use the **Plane.XY** node and feed this into the **SketchPlane.ByPlane** node. Since I am choosing the Level 1 floor plan view, drawing on the XY plane makes the most sense.

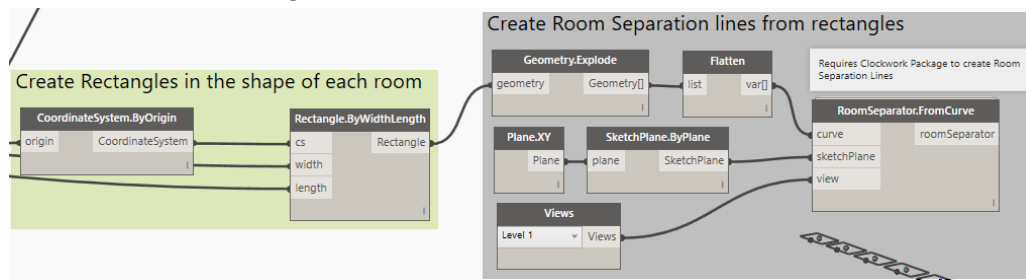


Figure 18

For the curve input, the first thing you need to know is that “curve” in programmer speak is any linear path, be it “straight” or “curvy.” So curve does not necessarily mean “curvy.” A straight line is considered a curve. (Sorry to bring it up again, but if you think back to High School Algebra one more time, you will recall that this is so...) OK, so we can put straight lines in the curve input; anything with endpoints really.

So my strategy here is to draw rectangles first, and then use those rectangles to provide the required curves and create the room separation lines. Dynamo is capable of creating individual lines, polycurves and closed shapes like rectangles. I chose to start with rectangles here because we already have the point locations and widths and heights from above. So with just two nodes, we can feed all of that in and make our rectangles.

The two nodes are pictured on the left side of Figure 18 and are the **Rectangle.ByWidthLength** and the **CoordinateSystem.ByOrigin** nodes. (These are both standard nodes). A coordinate system is exactly like it sounds. It defines the location and direction of a custom X,Y,Z coordinate system. Since its only input is origin, we can simply feed in the points we already have from before (the same

ones we used to place the families). Next we feed this into the cs port of **Rectangle.ByWidthLength** (cs is short for coordinate system) and grab the width and length from earlier in the graph as well. Use the same inputs that fed into the family set parameters above.

This creates rectangles in Dynamo but *not* in Revit. Dynamo geometry can be used to create Revit geometry but this is not automatic. Think of Dynamo geometry as more abstract. But you have to tell it what specific kind of Revit element you want it to create. This is where we use these rectangles to shape the required curves of our **RoomSeparation.fromCurve** node.

BUT, there is one more step...

Dynamo can create a rectangle as an element. But in Revit you would actually be creating four separate lines. Revit does not have polylines or continuous linear elements. All such elements are always separate lines (or curves). So, we need to convert these rectangles to individual lines, then feed them into the curve port. The node for this is called: **Geometry.Explode**. One input and one output so it is a simple pass through.

BUT, if you try to pass this directly into the curve input on **RoomSeparation.fromCurve** it will fail... <sigh>

One more node. **Geometry.Explode** created a complex list. We need to flatten it to make it a simple list. Recall above that we have two types of flatten nodes, well here a simple **Flatten** will do the job since we want to remove all hierarchy from this list. When you flatten the list and feed it into the curve port, you should finally have success. Room Separation Lines will be created in Revit!

Wow, that was a lot of complexity packed into a small group of nodes...

Create Rooms in Revit

Now that we have room separation lines, we are ready to create rooms.

Steamnodes will give us the node we need. So once again, this being a package node, you must download it from the package manager and install it first.

You can search your library after installing it for "room" and the node should appear. We want the **Tool.CreateRoomAtPointAndLevel** node. This has just two inputs: Point(s) and Level1. The points are no problem. We have those already.

Just feed the same set of points in that we used for the families and rectangles. For the level, just grab the **Levels** node from the library and select **Level 1** from the drop-down list (see Figure 19).

That's it! Instant rooms. And since we have the separation lines already, they will all conform automatically to the correct size. Very cool!

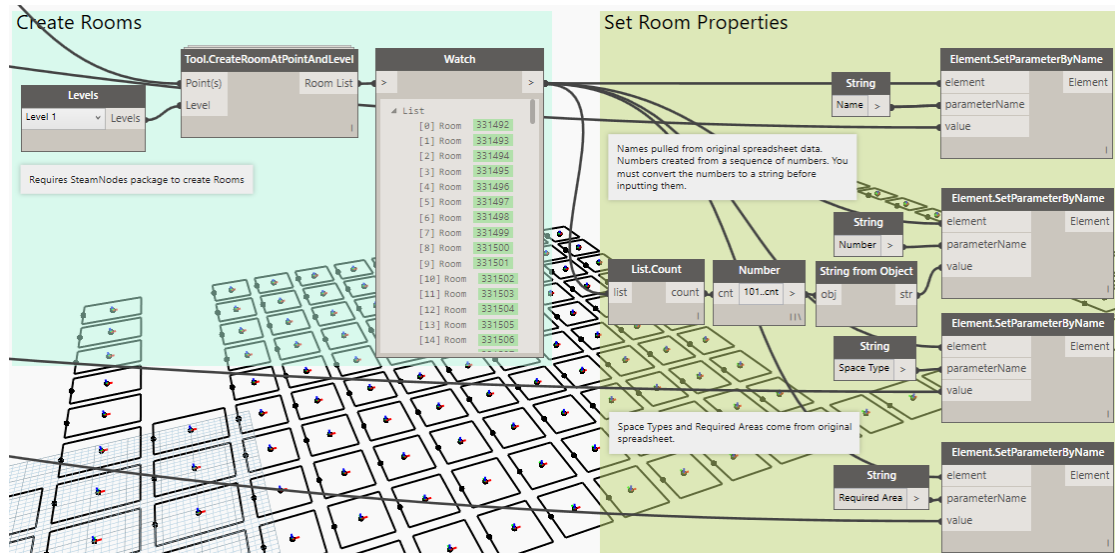


Figure 19

On the right side of the figure we are taking it a step further. Having a bunch of rooms is nice, but you probably want them numbered following a certain pattern and you might want to set the names or even include the required areas or other properties. So let's look at each group of nodes.

Way back in Figure 6, we extracted a list of properties from the processed Excel data. Among those were the room names, space types and the required areas. So here (at the top right and bottom right) we are simply using

Element.SetParameterByName again to feed those properties into the new rooms. Remember that a **String** node is required with the name of each parameter we want, typed exactly as it appears in Revit.

The fourth parameter to set is room Number. This takes a little more work as numbers were not part of our existing Excel data.

Using the trick we saw earlier, we can create a range inside of a **Number** node with a variable by typing: **101..cnt**. 101 is our desired starting room number and "cnt" is just a variable name that we assign for the input. Into this input, we feed the count from our list of rooms (see Figure 20).

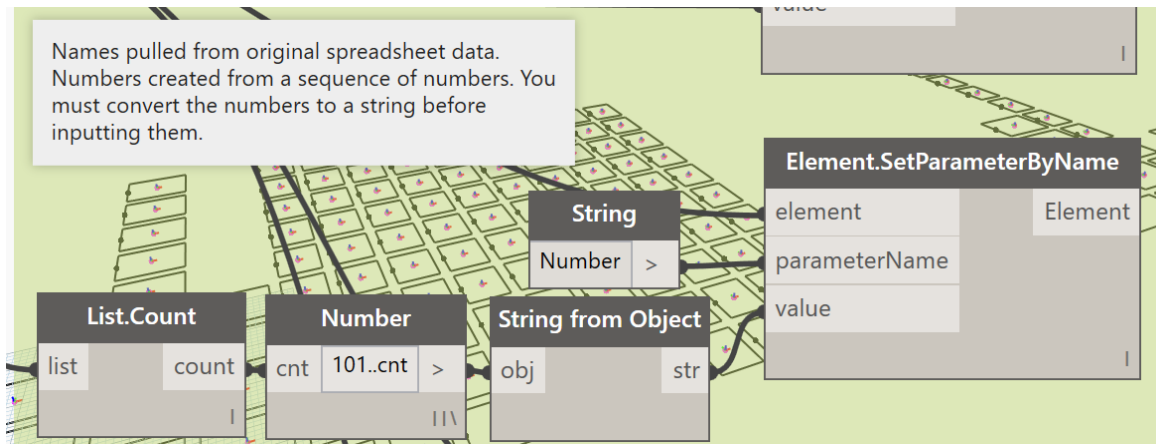


Figure 20

Next you see a **String from Object** node. This is necessary because we need to convert the series of numbers we just created into simple text data so that it is compatible with the data format of Revit's room number field. Data types are very important in any kind of programming; including Dynamo. So often, errors you see will be related to incompatible data types. This node converts for us. The rest is accomplished with another **Element.SetParameterByName** node.

Eye Candy

A picture speaks a thousand words they say. And I am not sure if they said it, but typically a color picture speaks louder than a black and white one. So let's add some color. For coloring the rooms, you cannot do better than a Revit Color Scheme. Simply duplicate the view, on the Properties palette, click the Edit button next to Color Scheme and set one up. Choosing colors by Room Names give a nice result (see Figure 21).

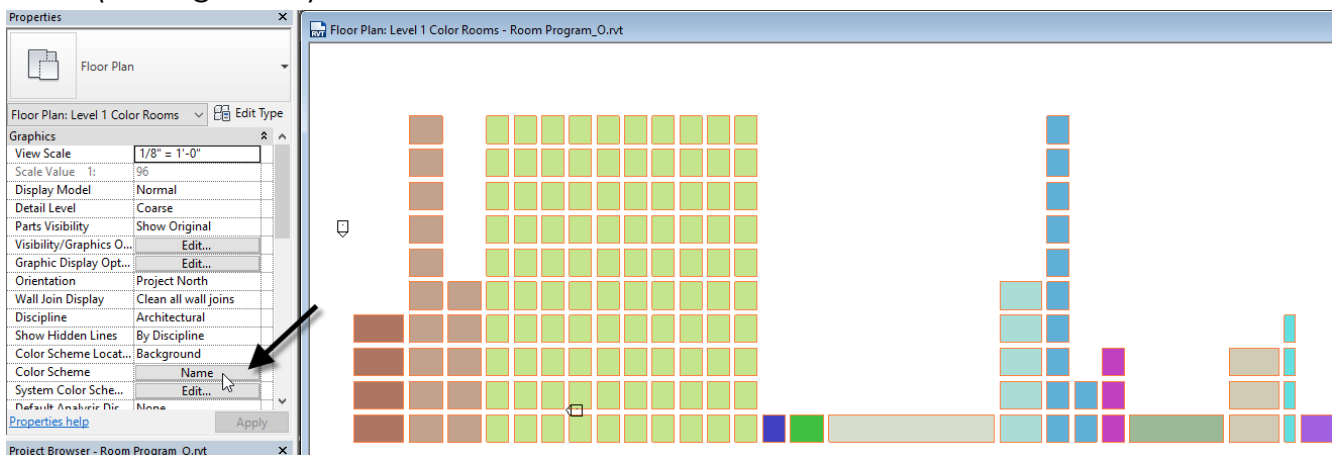


Figure 21

(Be sure to hide the generic models in this view so that the solid elements in the families don't cover the rooms and colors).

If you want to color the 3D families, you can do this with Dynamo. Open a 3D view in Revit first. Then add a couple mode nodes to the graph.

The key nodes are: **Color.ByARGB** and **Element.OverrideColorInView**. If you want different colors, you will also want to feed some sort of range into the r, g and/or b inputs. I did a count again, and fed that into a number range in a couple of these inputs. This will give a smooth gradient from 0 to 255.

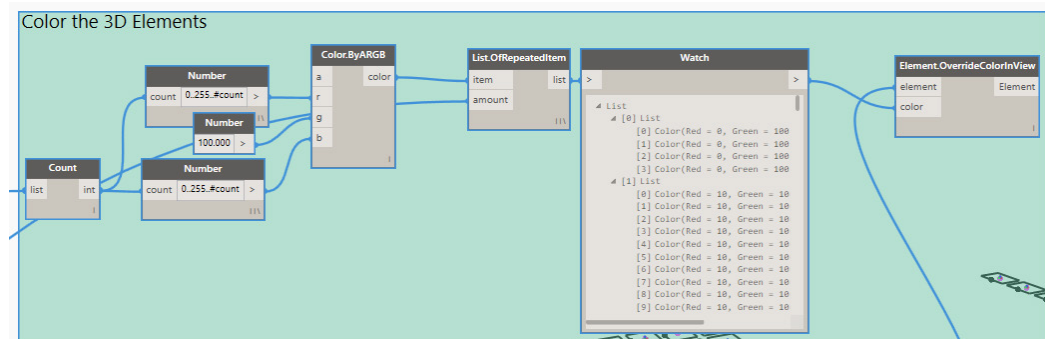


Figure 22

In order to apply the colors by column group, I did the **List.OfRepeatedItem** node again (see Figure 23).

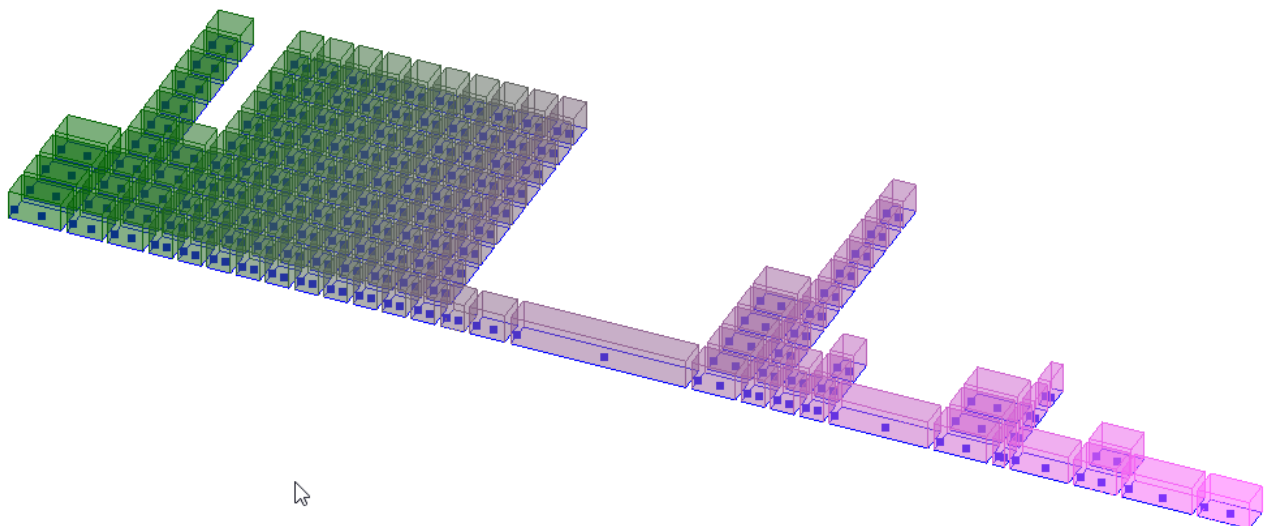


Figure 23

Unfortunately, with 27 total columns, there is not that much contrast from column to column when done this way.

If you want to use specific colors instead, there are other nodes both in the standard library and in package nodes to create specific colors. You can then

feed these into a **List.Create** which lets you build a static list of a set number of items (see Figure 24).

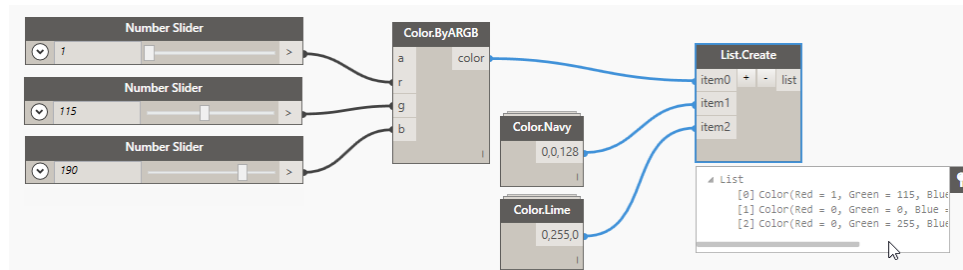


Figure 24

Conclusion

In the workflow showcased in this paper, I have taken the approach to read the Excel file as-is and process the data in Dynamo. An alternative approach to simplify the Dynamo graph is to process the data (including potentially repeating the rows) directly in Excel. Then simply read it into Dynamo and create geometry directly. So which is correct? Well either of course!

If your team is more talented in Excel than Dynamo, they can certainly slice, dice, repeat and parse data in Excel first. Alternatively, they can certainly take the approach we have here and process in Dynamo. Or even do a little of each. It is really up to you and your team. I believe it is important to set your team up for success in the best way possible. So don't force a talented Excel user to do things in a program that is foreign to them unless there are clear downstream benefits to the team and vice-versa.

As I have been learning Dynamo, my single biggest epiphany has been realizing that having tunnel vision is not good for productivity. Use the right tools in the right way for the right job. So a workflow that includes Dynamo as one of its tools can be very beneficial to the team. But if we try to make Dynamo our only hammer, then sadly we will have no choice but to see all problems as if they were nails. So be sure to look at the whole problem and fit Dynamo in where it makes the most sense and can provide the most benefit and by all means, if you can do something easier directly in Revit or in Excel, please do so!

Further Study

You can find more information and tutorials in my books and video training.



Please visit my website at: www.paulaubin.com for more information on my books.

I also have Revit video training available at: www.lynda.com/paulaubin. I have

several courses at lynda.com including: **Revit Essential Training**, **Revit Family Editor** and **Revit Architecture Rendering, Advanced Modelling in Revit Architecture, Formulas and Curves** and many more.



While not one of my courses, if you are new to Dynamo, I highly recommend: **Dynamo Essential Training** with Ian Siegel in the Lynda.com library. This course is just what you need to get up and running quickly.

If you have any questions about this session or Revit in general, you can use the contact form at www.paulaubin.com to send me an email.

Follow me on twitter: **@paulfaubin**

Zach can be found on the web at: <http://buildz.blogspot.com/>

While Zach has not been as active in blogging recently, there are enough previous posts in his archive to keep you busy learning about Dynamo and the Revit massing environment for quite some time! Do check it out if you do not already subscribe.

Zach is on Twitter as: **@ZachKron**

Thank you for attending. Please fill out your evaluation.