Dynamo

Computational Design Strategies in Revit: Building Complex Forms Using Dynamo

Suleiman Alhadidi & Geoff Kimm

Summary:
This session will introduce how to visually script geometries using Dynamo in Revit. Participants will explore how Dynamo can be used as a tool for studying complex forms and creating parametric morphologies.

Prerequisites:
There is no prerequisite for this workshop but we expect that this workshop is dedicated to expert level in Revit. We recommend that you install Dynamo and you introduce yourself to the massing environment in Revit/Vasari and the adaptive components.

Key Learning Outcomes:
1. Learn how to use Dynamo in Revit.
2. Learn how to divide design problems into logical solution using visual scripting.
3. Learn how to build parametric complex forms using scripting.

HASSELL

O(n)

www.onstudio.co

About the speakers:

SULEIMAN ALHADIDI is an architect, researcher and artist. He is currently working as a Computational Design Consultant at HASSELL, and teaching at The Royal Melbourne Institute of Technology in the area of Digital Architecture and Computational Design. He is also leading a Master level design studio at the University of Melbourne. He founded "Mutation Studio" in 2008; a studio dedicated to architectural design experiments. In 2013 he co-founded O(n) Studio as a design-based studio which explores the use of new technology in architecture design, design workflows and computational design strategies.

Suleiman worked previously as an architect with COOP HIMMELB(L)AU Vienna, Austria, before which he worked with Laceco international. He has practiced architecture in Australia, Europe and the Middle East. His work has been awarded, exhibited and published in Australia, Italy, Portugal, France, Jordan, China, Hong Kong, Korea and the UK. His current research interests include architecture design strategies, digital architecture, BIM technologies, and deployable complex kit-of-parts, interactive built environment, and fabrication technologies.

GEOFF KIMM is a software developer, researcher, and graduate architect. He has specified, created and worked with complex, high-availability information systems and analysis tools, and has degrees in environments, architecture, and science. He is currently leading a Master level design studio at the University of Melbourne

He has undertaken university-level research in the fields of computer science and digital architecture and has written digital design tools in diverse areas including those of complex systems and emergent behaviour, finite element analysis, computer vision, programme optimisation, and 3d scanning. His current research interests include urban modelling, responsive systems, and inflatable architecture simulation.

In 2013 he co-founded O(n) Studio to explore computational design and its integration in architectural practice to solve highly contextual challenges, and implementation of new technologies in architecture workflows.

01 Introduction

This workshop will focus on use of computational strategies within Building Information Modeling and how this can influence the architecture design workflows in Revit.

01.01 The production of computational thought

Computation has a profound impact on both the perception and realisation of architecture form, space and structure. It shifts the way one perceives form, the way in which the built environment is purposed and the way a design idea is produced. In this workshop we will focus on some computational design theory and solutions. Computational design has created a positive impact on industry and innovation, and with the introduction of computational workflows in Autodesk Revit, we expect that many architecture firms (especially the innovative ones) will increasingly implement such approaches in their workflows.

Flinders St Station_HASSELL & Herzog De Meuron.

01.02 Design Computation Era

In recent years, computation has become an established driver for design innovation, as seen in the optimised use of computer-aided design (CAD) to create high-profile cultural

buildings such as Frank Gehry's Guggenheim Museum Bilbao and Foster + Partners The Sage Gateshead.

CAD Eras (Book: Inside Smartgeometry: Expanding the Architectural Possibilities of Computational design : p.40)

The above diagram summarises the history of practical Computer Aided Design: the 2D drafting era, the building information modelling (BIM) era, and the computational design era. Design computation introduces the distinction between a generative description of a building (as a graph or a script) and the resulting generated model. The designer is no longer directly modelling the building: instead she or he develops a graph or a script whose execution generates the model.

01.03 The production of complex forms using design computation

Architecture has become less noisy in its display of the digital, and digital applications have assisted the development of better building aesthetics and performance by applying computational processes to issues such as material, tectonics, environment and even psychological impact. From four or five years ago excitement has centred largely on the emergence of new technologies for design strategies at the heart of building and practice that use well-refined computational techniques in BIM platforms, and that are applied even beyond design and construction to the monitoring and assessment of completed refined structures to produce better built environments.

Complex form finding research projects at O(n) Studio.

02 Dynamo

02.01 What is an algorithm?

In general, we approach computational design by dividing the problem into smaller problems or procedures to solve it. In computer science, mathematics and even in our field the step-by-step sequence of processes to solve the problem can be referred to as a definition, script or an algorithm.

The logic of an algorithm.

Once we have defined a particular type of problem, a script sequence can be designed which allows the computation of a solution of that problem.

02.02 Visual scripting and Dynamo

In recent years, visual scripting was introduced for non-programmers to do what programmers do; therefore, designers are able to design their own tools in order to use in their design process. This has introduced a new dimension for designers to expand their digital toolbox and create more innovative and unique outcomes.

Visual scripting has been used in many areas such as games and illustrations for young people using Scratch (http://scratch.mit.edu/) , video and audio editing using MAX MSP, video games using uScript, and the most relevant example is design using Grasshopper3D in Rhino.

Visual scripting using MAX MSP: splitting image/video into 8 equal dimension parts.

Dynamo and its expansion of the functionality of Revit and Vasari is a good example of how such tools can be used in BIM to create innovative outcomes.

03 Session- Part A

Before attending the workshop we recommend that you install Dynamo 0.6.3 from this website http://dynamobim.org/ and familiarize yourself with its interface.

03.00 Installing Dynamo & Interface

Installing Dynamo should be straightforward. It can be installed in either the Revit or Vasari environments, but we are dedicating this workshop to the use of Dynamo in Revit and specifically in the Massing environment.

Revit has two environments for the creation of projects: the massing environment which is used for early design exploration and the project environment which is used for design development and the documentation of a project. Dynamo can be used in the project environment for data management and the generation of details and quantities and so on which is an interesting area in itself. however we will focus in this workshop on the conceptual aspect and particularly on how to expand Revit's capabilities to create complex forms and systems.

Revit welcome screen: select new conceptual massing from Families option.

After installing the Dynamo add-in you can open Revit 2014 and use the default massing template (metric massing).

Dynamo add-in under Add-Ins | Visual Programming.

Dynamo is found under Add-Ins | Visual Programming.

Dynamo interface.

Once you start Dynamo a new window will appear which can act independently from Revit.

In this workshop we will give an introduction to Dynamo however we assume that most of the participants know about Dynamo basics. We recommend that you check the Dynamo website for a detailed introduction. The first three steps of this workshop though are dedicated to providing a basic introduction.

03.01 Dynamo introduction - create a point in Dynamo

Dynamo introduction: creating a point.

A point in Dynamo can be created with the `XYZ` node which takes an input for each of the X, Y, and Z values. These values are here specified in a `Number` node which has been given the value 0.0. As the point is to be on the origin the output of this `Number` node is used for

each of the X, Y, and Z inputs of the `XYZ` node. The output of the `XYZ` node is here passed to a `Watch 3D` node which allows geometry to be previewed.

The Dynamo script must be run to see its result. This can be done by clicking the Run button in the bottom left of the Dynamo window.

Dynamo introduction: using sliders and a reference point.

A `Number` node can be replaced by a `Number Slider` node or an `Integer Slider` node. To see the effect of this change in the Revit environment (or any XYZ point in general), a reference point must be created from XYZ data using a `Reference Point` node. The Dynamo script must be rerun to see the new reference point.

03.02 Dynamo introduction - referencing a family in Dynamo

Dynamo introduction: referencing a family.

A family in Dynamo can be referenced by the `Select Family Type` node which allows a family loaded into the Revit environment to be chosen. A `Watch` node is used in this example to display the name and ID of the selected family.

03.03 Dynamo introduction - creating, dividing, and placing components on a curve

Dynamo introduction: creating a curve from points.

In this step three XYZ points are created as the basis of a curve. These three points are combined into one list in a `List` node. A curve is created from this list of points in a `Hermite Spline` node. The curve is previewed in a `Watch 3D` node.

Dynamo introduction: dividing the curve.

Division points of the curve created in the previous step are now found by using an `XYZ Array On Curve` node. This node takes a parameter `count` to which has been connected the output of a `Number` node with a value of 5.0 and therefore five division points are created. The division points are previewed in a `Watch 3D` node. Note that the `Watch 3D` node also displays the curve even though it is not directly connected — this behaviour can be turned on or off by right clicking on the `XYZ Array On Curve` node and toggling the *Preview Upstream* option.

Dynamo introduction: placing components on the curve division points.

The family that was previously referenced is now placed on the curve division points by a `Create Family Instance` node. Note that the family instances cannot be viewed in the `Watch 3D` node.

Dynamo introduction: created family instances in Revit.

As a result of the script the referenced family is now instantiated in the Revit environment at the division points of the curve. The curve is also present in the Revit environment as in the script a model curve was created from the output of the `Hermite Spline` node by a `Model Curve` node.

03.04 [Demo] Phyllotaxis growth

Phyllotaxis growth: script output.

This step is to explain some operations and nodes in Dynamo and will illustrate these by modifying some parameters to adjust the output of the given script. Afterwards in step 05 creation of a complex mathematical form using Dynamo will be shown.

Phyllotaxis growth: Dynamo script overview.

Open the file *Phyllotaxis growth.dyn* in the massing environment and you will see the script shown above.

Phyllotaxis growth: number nodes in Dynamo.

The script generates points from mathematical formulas. We start by introducing the idea of creating a number sequence in Dynamo. To create a series of numbers you can use the `Number` node and specify the start number and the end number using this syntax: `'start number .. end number'.`, If you need to specify the number of subdivisions between the start number and the end number you need to follow it with `'..#number of subdivisions'`. For example, `'1 .. 2 .. #5'` produces the output 1.0, 1.25, 1.5, 1.75, 2.0.

Phyllotaxis growth: change in the value of s - from left to right: 0.25, 0.15 and 0.3

After creating the XYZ coordinates in an `XYZ` node, you can explore the creation of three type of geometries; reference points, circles which have variation in their radii, and curves. The ratio of the circle growth can be adjusted by changing the value of `s` passed to `phi2pi` to, for example, 0.25, 0.15 or 0.3, shown above, and changing the value of `Z` can create sine wave-like curves, shown below.

Phyllotaxis growth: change in z to create sine waves- like curves

03.05 Mathematical complex geometries - Nautilus Shell - Part A

Mathematical complex geometries: series of points for a nautilus shell generated from mathematical formulas for X, Y and Z values.

The link between architecture and mathematics appeared in early design theories and has been developed over the centuries. Recently, with the introduction of computational design theories and mathematics as a fundamental generator of geometries, it appears as a genetic code in architecture design.

Complex mathematical geometry: Nautilus shell script.

The aim of this step is to show how such an application of complex mathematical formulas can be achieved in Dynamo. The script for this step is shown above.

Complex mathematical geometry: Nautilus shell: creating series of lists

The first part of the script is to create a series of lists to define the domain of the mathematical formulas. We start here by creating variable inputs for the numbers — in Dynamo we can use either `Number` nodes or `Formula` nodes to achieve the this. During

the workshop we will discuss how to create cross referenced lists by applying the `Repeat` node and then introducing a data structure by rearranging the lists by using `Transpose Lists` and then `Flatten Completely`.

Complex mathematical geometry: nautilus shell - creating XYZ coordinates by using mathematical formulas

The next step is to apply mathematical formulas using `Apply Function` nodes to generate the X, Y, and Z values and to create XYZ points from those values using the XYZ node. The following formulas are used:

```
X = (pow(sin(u),2) * sin(v)) * pow(1.2,v) *100
Y = (sin(u)* cos(u)) *(pow(1.2,v)) *100
Z = (pow(sin(u),2) * cos(v)) * (pow(1.2,v))*100
```

Then we can connect the XYZ node to a `Reference Point` node to show the points in the Revit massing environment.

03.06 Mathematical complex geometries - Nautilus Shell - Part B

Mathematical complex geometries: outcomes: curves and panels on the nautilus shell.

In this step we will create geometry using the points we have defined for our nautilus shell. We will start with curves and will then panelize the shell.

Mathematical complex geometries: Dynamo script Creating lines and panelizing the geometry

The extension to the script to do this is shown in the image above.

Mathematical complex geometries: culling the start and end of the list.

In the image above lists are culled to remove identical points to create curves from those points in the following instruction. In order to achieve this we trim the list by using a `First of List` node and a `Rest of List` node; we then reverse the list by using a `Reverse` node and repeat the same process.

Mathematical complex geometries: Creating Hermite Splines in U and V directions for the geometry

Next we use a `Hermite Spline` node to create the curves between the points of each sublist. Afterwards we use these curves to create additional curves across them by dividing them with an `XYZ Array On Curve` node to create a list with sublists of XYZ points. This new list is then flipped by using `Transpose Lists` and the result is used to create new curves running across the original ones.

From this point on we can use adaptive components and populate them on the geometry. If you are using a template other than the one provided in the dataset, you need to load the *O(n) Studio-Basic roof panel* family into the massing file in order to use it in Dynamo.

Adaptive component: basic extrusion

The adaptive component shown above was created using the adaptive component family template. To recreate it you need to place four points on the *Level 1* plane (the horizontal plane) and select them and then you need to turn them to adaptive. Next you can connect those points by using reference lines — make sure that you are activating the 3D snapping option in order for lines to follow the points above the working plane. Afterwards you need to select the chain of lines and use create form component to extrude the shape in 3D. In the workshop we can discuss how to adjust some of the parameters (like extrusion limit and material) of this component in Dynamo.

Mathematical complex geometries: Populating the adaptive component on the geometry

In Dynamo you can use some custom components — predefined script elements — provided by other users by searching for a package under the *Packages* menu. We will be using `Grid to Quads` to arrange the points into quads in order to place the basic adaptive component which we created earlier. Afterwards we pass the quads to an `Adaptive Component by XYZs` node and also pass in a family type selected in a `Select Family Type` node. For this example, and if you are using the template provided, you should change the family type to *O(n) Studio-Basic roof panel*.

03.07 Mathematical complex geometries - Nautilus Shell - Part C

Mathematical complex geometries: pavilion created from the nautilus shell.

This part of the session modifies the nautilus shell script used previously to create and panelize a small architectural pavilion. The outcome is shown above. There are three main steps to do this.

Mathematical complex geometries: halving the domain and switching the Y and Z inputs.

First, change the domain of the shell to create a half shell by adjusting the inputs as shown in above left image. Then switch the Y and Z inputs as shown in the above right image.

Mathematical complex geometries: modifying the script to create the roof geometry 02.

Second, we need to modify the culling of the points for the curve as we need only to cull the list from one side this time.

Third, we will create a new adaptive component. The adaptive component needs to be designed properly so it will flex and deform in three dimensions; therefore the geometry should adjust itself to any four points in three-dimensional space it may be applied to.

Mathematical complex geometries: completed roof adaptive component.

Mathematical complex geometries: five steps to create the adaptive component.

There are five main steps to create the roof adaptive component used in this example.

1. Start by opening the adaptive component family template and create 4 points in the space. Connect two points with a 3D reference line by activating the 3D snapping option. Afterwards connect the other points to the midpoint of the first line.
2. Create normalized points on the curves — we will use these to keep the same geometrical shapes by maintaining the relationship of the line to the other geometries.
3. Connect each dot that was created in the previous step in order to create 3D geometry from them.
4. Extrude the 4 parts of the frame individually.
5. Create a sweep form for the inside part. For this example we will be using a circle of 80mm for the main bracing and 50mm for the secondary bracing.

After finishing the file, you can do a final save before loading it to the conceptual massing family which is used for our example. You can then change in Dynamo the `Family Type`

`Selection` node to this family. It is expected that this operation will take up to a couple of minutes to process as we are placing a lot of geometry in the file. You can further this exercise by creating architecture detailing and structural supports for the panels by using the same method.

03.08 Questions and answers

04 Session - Part B:

This part of the session begins by generating UV points within Dynamo to perform operations upon a surface. It then presents further examples of mathematical complex geometries in Dynamo and a streamlined workflow for switching between formulas within a script. The session part concludes with two examples of manipulating adaptive components in Dynamo.

04.01 [DEMO] Surface UV points - part A

Surface UV points - part A: simple UV grid on a surface.

This introductory example produces a simple n by n grid on a surface. There are four significant steps. First, the `Select Face` node is used to select the surface upon which to operate. Second, the `Get Surface Domain` node outputs the UV domain (the minimum and maximum values for both U and V) for the selected surface. Third, the `UV Grid` node divides the given UV domain into n points for each of the U and V dimensions to produce a grid of n^2 UV points. Fourth, the UV points in the grid are evaluated on the surface in the `Evaluate Surface` node to find the corresponding XYZ points. As the UV grid was generated from the same UV domain for that surface the XYZ points fill the surface from edge to edge.

The use of UV points is extended in the examples that follow.

04.02 [DEMO] Surface UV points - part B

Surface UV points - part B: placement of a single UV point on a surface with Python.

This example continues the use of UV points and introduces their manipulation in Python by creating a `Python Script` node that allows a single UV point to be positioned on a surface.

Surface UV points - part B: Python script for positioning a UV point on a surface.

The example is similar to the previous example — the most significant difference is a `Python Script` node is used in place of the `UV Grid` node. This node has been renamed in the script to `UV Point`. Within the `Python Script` node a single UV point is created within the given surface UV domain, the U and V values of which are generated by the following code:

```
u = umin + (umax - umin) * uratio
v = vmin + (vmax - vmin) * vratio
```

The new UV point is then created by:

```
uv = UV(u, v)
```

and is assigned to the output of the `Python Script` node by:

```
OUT = uv
```

04.03 [DEMO] Surface UV points - part C

Surface UV points - part C: random UV points on a surface using Python.

This example extends the use of UV points within a surface UV domain in a `Python Script` node from placing a single point to placing multiple points. It also demonstrates using Python to control the position of a family instance on a surface.

Surface UV points - part C: Python script for positioning random UV points on a surface.

The `Python Script` node creates n UV points randomly positioned within a given surface UV domain. The script also allows for a surface UV domain border within to *not* create UV points — this is useful, for example, when otherwise a UV point near the edge of the surface might cause a family instance placed on the corresponding XYZ point to overhang the edge of the surface. The most significant lines of the script are:

```
for i in range(0, count):
    OUT.append(UV(rand(umin, umax), rand(vmin, vmax)))
```

To break this down, the line:

```
for i in range(0, n):
```

means to repeat the indented line(s) following it *n* times. The function:

```
rand(a, b)
```

generates a random number in the range *a* to *b*. The code:

```
UV(u, v)
```

creates a new UV point with values *u* and *v*, and the code:

```
OUT.append(x)
```

appends *x* to the output variable. All this may be written in *pseudocode*:

```
for count times do the following:
    add to the output a new random UV point
```

Surface UV points - part C: tree family instances placed on a surface.

The surface XYZ points corresponding to the generated UV points are found, as in the previous examples, by passing them into an `Evaluate Surface` node. The XYZ points are then passed to a `Create Family Instance` node to create instances of the family specified in the `Select Family Type` node.

04.04 [DEMO] Surface UV spiral

This example focuses on extending the use of UV point specification in `Python Script` nodes to create geometry on a surface.

Surface UV spiral: Dynamo script overview.

This script maps a spiral curve onto a surface and then extrudes that curve. It has three parts.

A. Generation and visualisation of the spiral UV set.
B. Evaluating the UV points on the surface as XYZ points and creating a curve from these.
C. Extrusion of the curve using face normal vectors.

Surface UV spiral - part A: UV Spiral Python Script node mapping a spiral on a UV domain.

In part A, a spiral is generated in UV space in a `Python Script` node. Note that the UV spiral at this stage is *not* mapped to the surface UV domain. A helper `Python Script` node — `Visualise UV` — is used at this stage to translate the UV values to XYZ values so that they may be viewed in a `Watch 3D` node.

Surface UV spiral - part B: Remap UV Python Script node mapping a UV domain to a target domain.

In part B, the UV spiral is mapped to the UV domain of the target surface in a `Python Script` node, here called `Remap UV`. If this step were omitted the UV spiral would most

likely not match the UV domain of the target UV surface; it would be either too large, or too small, or otherwise misaligned. The XYZ points on the surface are then found for the UV points by using an `Evaluate Surface` node. A `Hermite Spline` node is applied on these XYZ points so that the result may be visualised as a curve in a `Watch 3D` node.

Surface UV spiral - part C: Face Normal Python Script node for computing normals of a face for given UV points.

In part C, the XYZ points on the surface calculated in the previous part are extruded. The first step is to in a `Python Script` node, renamed `Face Normal`, find for each XYZ point on the surface the surface normal vector at that location. (Note that the normal is here found within a `Python Script` node to illustrate the use of Python within Dynamo — it would be possible to instead use the standard `Surface Normal` node.) These normals are then scaled (using `Multiply`) and added (using `Add`) to their corresponding XYZ points.At this stage there are two lists of XYZ points: the points for the spiral on the surface and the points that have been offset from these. The `Line by Endpoints` node is used to create a line between each pair of points. The `Loft Surface` node is used to loft a single surface from this list of lines, but first the lines are converted to model curves using a `Model Curve` node.

Surface UV spiral: curve extruded using face normal vectors.

The result is a surface that follows the geometry of the original surface. This example is simple for ease of understanding, but the techniques it shows are applicable to more complex situations. Two extensions that are of potential interest and are trivial to implement

are to modify or replace the `Remap UV` node such the input UV pattern is repeated across the target surface UV domain, and to, instead of lofting extruded mapped points, loft mapped points between two adjacent surfaces.

04.05 [DEMO] Surface UV recursive division

Revit offers ways of dividing a surface according to a pattern but Dynamo allows more complex patterns to be applied. This examples shows how Python within Dynamo can be used to recursively divide a surface. An adaptive component is then applied to the generated points.

Surface UV recursive division: example sequence of increasing levels of division detail.

The surface division in this example is a *recursive* process; that is, the same steps are repeatedly applied in a branching execution flow. The recursive algorithm used here works by partitioning a given surface UV domain into two parts. Each of these two parts are then passed into the *same* algorithm, and are thus themselves each divided into two parts. These parts, of which there are now four (being the result of two lots of two divisions), are treated in the same way and the process continues until a limit is reached.

Surface UV recursive division: Dynamo script overview.

The general flow of the script in this example is similar to that of the *Surface UV points - part C* example — an operation is performed on a surface UV domain, the XYZ points on the surface are found for the UV points, and something is then applied to these XYZ points. A significant difference, however, is that an `Adaptive Component by XYZs` node is used in place of the `Create Family Instance` node.

Surface UV recursive division: UV Recursive Tiles Python Script node for subdividing a UV domain.

The recursive division itself is done in a `Python Script` node, in this example called `UV Recursive Tiles`. The function definition header within that node:

```
def partition(minu, minv, maxu, maxv, dir):
```

defines a function called `partition` and says that it will take five values. The first four of these values are to define the minimum U, minimum V, maximum U, and maximum V values of the surface UV domain. The last value, `dir`, defines in which dimension to divide — either U or V. The limit of recursion is controlled by the line:

```
if (maxu - minu) <= limit or (maxv - minv) <= limit:
```

which says that if the UV domain passed into the function is less than `limit` in either the U or V dimensions then the recursive process should not be called further by this *instance* of `partition()` and the points for the current UV domain should be returned. The recursive process is started by the line (which is outside of the `partition()` function definition):

```
OUT = partition(0.0, 0.0, 1.0, 1.0, dir)
```

and thus the UV space that is divided spans a rectangle defined by `UV(0.0, 0.0)` to `UV(1.0, 1.0)`. It is therefore only coincidence if the UV domain of the subdivision actually matches that of the surface to which it is to be applied — the `Remap UV (Python Script)` node that has been introduced previously is used to ensure that chance does not need to be relied upon.

Surface UV recursive division: family instances placed on recursively subdivided surface.

XYZ points on the surface are found for the mapped UV points using the `Evaluate Surface` node. A simple pyramid adaptive component — *O(n) Studio: Pyramid* — is selected in a `Select Family Type` node and this is applied to the XYZ points by an `Adaptive Component by XYZs` node.

The division of a surface in Dynamo using the general approach shown in this example is not limited to using recursion, nor to using only one type of adaptive component, and nor to using an adaptive component at all. Possible extensions that could be implemented with only minor changes are adjusting and fine tuning the `UV Recursive Tiles` node logic, using different adaptive components on different areas of the surface depending on a particular condition, or, without using adaptive components, using the XYZ points to 'rebuild' the surface or extract geometry according to a certain pattern.

04.06 [DEMO] Surface UV recursive division colouring

This demonstration colours the saved family instances from the *surface UV recursive division* example given above.

Surface UV recursive division colouring: Dynamo script overview.

The saved family instances must be loaded in Revit's project environment for the node that performs the colouring, `Override Element Color in View`, to work.

Surface UV recursive division colouring: getting family instances and element parameter values.

The family type to reference — *O(n) Studio: Pyramid* — is selected in a `Select Family Type` node and the instances of the family type are found by using the `Get Family Instance by Type` node. The value of the parameter that will be used to determine the colour of each instance, in this case Area, is found by using a `Get Element Parameter Value` node.

Surface UV recursive division colouring: normalising the Area family element parameter values in Python.

The values of the Area parameter (i.e. those passed from the `Get Element Parameter Value` node) are normalised; that is, they are adjusted to range between 0.0 and 1.0. This is performed in a `Python Script` node renamed in the script to `Normalise`. These normalised values are then passed to a `Color Range` node and determine a color to use in the specified colour range for each family instance.

Surface UV recursive division colouring: coloured family instances.

The result is that each instance of the *O(n) Studio: Pyramid* family is coloured in the view according to the relative size of its area. Parameters other than area can be used to control the colouring, or could be used as the input to other operations in Dynamo.

04.07 [DEMO] Dynamic selection of mathematical surfaces

This example addresses some problems with entering formulas into Dynamo and selecting a particular one to use. It also gives further examples of creating mathematical surfaces in Dynamo. `Python Script` nodes are used extensively throughout this example.

Dynamic selection of mathematical surfaces: problem - the order of the inputs of the standard Formula node depends on the order of variables in the expression.

As shown in the image above, the order of inputs to a `Formula` node in Dynamo depends on the order in which they are used in the expression given in the node (cf. Grasshopper). This causes issues when maintaining a script as changing a formula can lead to variables being incorrectly used and switching between formulas that have variables in different orders

is not easy. If such changes are made the script must be manually updated or, more seriously, the need for updating the script is overlooked and errors enter the workflow.

Dynamic selection of mathematical surfaces: problem - creating and selecting from a list of formulas is not streamlined.

The image above illustrates a problem with selecting between multiple formulas in Dynamo: each formula must be individually entered into its own `Formula` node. This, in conjunction with the `Formula` node variable order problem given above, means that maintaining a variety of formula options when exploring a computational problem is not streamlined.

Dynamic selection of mathematical surfaces: simplified script evaluating an expression in a Python Script node.

Python offers a way of evaluating code by calling its `eval()` function. The sample script above uses this by calling the `eval()` function in a `Python Script` node called `Eval` on an expression passed in from a `String` node. This is explained in more detail below.

Dynamic selection of mathematical surfaces: Python component for evaluating the expression.

In the sample script a `Python Script` node uses the `eval()` function to evaluate a given expression for given U and V values. The appropriate mathematical functions and constants are loaded from the Python `math` library — `sin(), cos(), pi, pow()`, and so on — and the U and V variables for each iteration of the evaluation loop are passed into the `eval()` function using the `locals()` function. The expression syntax is similar to that of the NCalc mathematical expression library used in Dynamo.

Dynamic selection of mathematical surfaces: Dynamo script overview.

The Dynamo script for this example allows 31 mathematical surfaces to be selected between. The domains for the U and V values and the formulas for the X, Y and Z values are each defined in their own `String` node as a list of 31 lines. The selection of the surface to generate — and thus the selection of the corresponding domains and formulas — is dictated at a single point within the script. In this example the domains and formulas are given in `String` nodes but it would be possible to instead link these to an Excel worksheet.

Dynamic selection of mathematical surfaces: Python Script node for selecting an item from a string list.

A `Python Script` node allows a single line specified by an index to be selected from a list of many. This node is used to select the domains for the U and V values and the formulas for the X, Y and Z values from their corresponding string list.

Dynamic selection of mathematical surfaces: Python Script node for creating a domain range.

A `Python Script` node is used to simplify the specification and generation of domain ranges for the U and V values. The node uses the Python `eval()` function and simple string manipulation so that domain ranges may be specified in the same way as in Grasshopper by using the syntax “`a to b`”.

Dynamic selection of mathematical surfaces: examples of generated forms.

This example, while limited in its implementation to demonstrating the generation of a mathematical form from a selection of 31 possible forms, introduces methods and a workflow approach that are extensible to other computational solutions.

04.08 [DEMO] Building a parametric bridge with complex adaptive components

Parametric bridge: with adaptive components that change in size to create lightweight elements.

This example will be detailed in the workshop and is only covered in summary here. It illustrates computational design concepts that can also be used to design towers as well as roof systems.

Parametric bridge script: part 01/02.

To start this script you need to first open the associated Revit file and then open the Dynamo script. The image above and the image below show the overall script in two parts.

Parametric bridge script: part 02/02.

The script illustrates how to create a surface using geometry inputs created in Revit and how to use an attractor point in Revit to manipulate the panels parameters. It uses the location of the attractor point to specify the best location to place the lightweight panels.

Parametric bridge: point and circle Revit geometry input to Dynamo (left), and polygon created from the circle in Dynamo and midpoints of the polygon were transferred in 3D (right).

Within the Revit file are a point and a circle created vertically using the point plane and these are input into the Dynamo script. We will reference these elements in Dynamo. The overall logic of the script is to create a curve that is generated from geometrical operations and transformations on the referenced point and circle, then apply a sequential rotation to this curve, and then divide the resulting curves into panels.

Parametric bridge: Creating the bridge curve to be rotated in the next step.

After creating points from the input geometry particular list operation nodes — `Build Sublist`, `List`, and `Transpose Lists` — are used to reorganize the points to later create splines.

Parametric bridge: applying transform operations to array the curves with sequential rotation.

Next, multiple copies of the first section (represented as a point list) are made and then moved by adding to each an XYZ point with an `Add XYZs` node. These XYZ points are generated by connecting the output of a `Number Sequence` node to the `X` input on an `XYZ` node — the XYZ points will effectively act as movement vectors.

Parametric bridge: arrayed rotation of splines.

At this stage we will also create splines by using a `Hermite Spline` node and we can then apply a sequential rotation to those curves with a `Transform Curve` node that accepts as input a `Rotate Transform` node.

Parametric Bridge: Creating the surface.

From those curves we can create a surface by lofting with a `Loft Surface` node. In order to do this we need to convert the list of curves to model curves with a `Model Curve` node.

Parametric bridge: creating nest-like curves to represent structure/division for panels.

The next step of the workflow divides the curves and connects the division points to create a nest like frame.

Parametric bridge: transformed curves.

The above image shows the effect of the transformed curve network.

Parametric bridge: populating the pattern-based components.

In last stage of the script we will populate the pattern-based components (nested families) to the quad points in a way similar to that used in section 04.06 of this document. We will define the size of the components by creating an attractor point in Revit. To achieve this we need to calculate the distance between the attractor point and the approximate centre of

each panel. In the workshop we will discuss ways to achieve this, however in the script there is one solution specified.

This script is interesting not only for form generation and structural logic, but also has potential for environmental solutions as well.

04.09 [DEMO] Complex animated window fin systems

Left : animated fin systems with a gradient change in color properties done using Dynamo.
Right: case study (similar project): Liverpool Villahermosa Department Store by Iñaki Echeverria souce: http://inhabitat.com/liverpool-villahermosa-department-store-gets-a-twisting-concrete-double-skin/liverpool-villahermosa-jaime-navarro3/

This example illustrates manipulating parameters for each of a series of adaptive components. The resulting geometry has a precedent in the Liverpool Villahermosa Department Store by Iñaki Echeverria.

Complex animated window fin systems: animated fin systems using irregular curves input with a gradient colour change.

An adaptive component fin is arrayed along two curves and three variations are introduced: the dimension of the middle of the component, the twist of the component, and the material of the component.

Complex animated window fin systems: Dynamo script overview.

The script will be demonstrated in detail in the workshop. The script flow has six significant steps which are described below.

Complex animated window fin systems: selecting curves and finding division points.

First, two curves are selected from the Revit environment and division points are found by `XYZ Array On Curve` nodes.

Complex animated window fin systems: selecting and arraying the family.

Second, the adaptive component is selected in a `Select Family Type` node and arrayed along a curve by an `Adaptive Component by XYZs` node.

Complex animated window fin systems: adjusting the dimension of the middle of the adaptive component.

Third, the dimension of the middle section of the adaptive component is adjusted by a `Set Element Parameter` node based on a parameter `Dm` passed in from a `String` node.

Complex animated window fin systems: loading and sampling the image and adjusting the twist of the adaptive component.

Fourth, an image file is loaded in a `Read Image File` node. The image is sampled by a `Color Brightness` node and the resulting value is used to control the twist of the adaptive component instances by a `Set Element Parameter` node based on a parameter `Rm` passed in from a `String` node.

Complex animated window fin systems: loading materials and combining them into a list.

Fifth, six materials are each loaded in a `Get Material by Name` node and these are combined into a list in a `List` node.

Complex animated window fin systems: applying materials (with the oscillating material sequence connected).

Sixth, the materials are applied to the adaptive component with a `Set Element Parameter` node based on a parameter `MaterialColor` passed in from a `String` node. Two ways are given for organising the materials; one creates an oscillating sequence from the material list using `Reverse` and `Repeat` nodes and the other creates a random sequence from the material list using the `Repeat` and `Shuffle List` nodes.

04.10 Questions and answers

Further enquires:
_O(n) Studio email: design@onstudio.co

05 References:

_O(n) Studio Dynamo resources http://onstudio.co/
_Dynamo website: http://dynamobim.org/
_Dynamo in Autodesk Vasari: http://autodeskvasari.com/dynamo