AUTODESK UNIVERSITY

AR22128

Paneling With Intent To Fabricate

Colin McCrone
Safdie Architects

Jaron Lubin
Safdie Architects

Learning Objectives

- Compare Revit-native and Dynamo-oriented techniques for paneling
- Account for various types of design constraints and edge conditions
- Control multiple Revit family types conditionally with Dynamo
- Cite examples for how BIM and computation enable complex projects

Description

With examples from two projects (one built and one under construction), see how Dynamo can be used with Revit as part of a design process to rationalize and document a complex façade with flat panels. This instructional demo intends to show how design constraints might be expressed through scripting and ultimately through architecture.

Your AU Experts

Colin McCrone (@colinmccrone) is the Director of Design Technology at Safdie Architects in Boston where he guides the firm's technology initiatives and serves as a project-based computational designer. Previously, McCrone was the Computational Design Evangelist for the Dynamo product team at Autodesk in San Francisco. McCrone has taught computational design and BIM at the California College of the Arts, and he was a researcher in parametric energy modeling for early-stage architectural design at the Program of Computer Graphics at Cornell University.

Jaron Lubin (@jaronlubin) is a Principal at Safdie Architects. Mr. Lubin is central to many of the firm's recent speculative proposals and realized projects, seeing a variety of geographic contexts, multiple scales of work and diverse building and urban programs. His recent assignments include proposals for the National Art Museum of China in Beijing, a new Chinese opera house in Hong Kong, and a competition entry for the National Library of Israel. He was part of the competition-winning design team for the Marina Bay Sands in Singapore and is currently directing many of the firm's Singapore projects including the residential high rise tower Sky Habitat and Project Jewel, a new mixed-use complex at Singapore's Changi Airport.

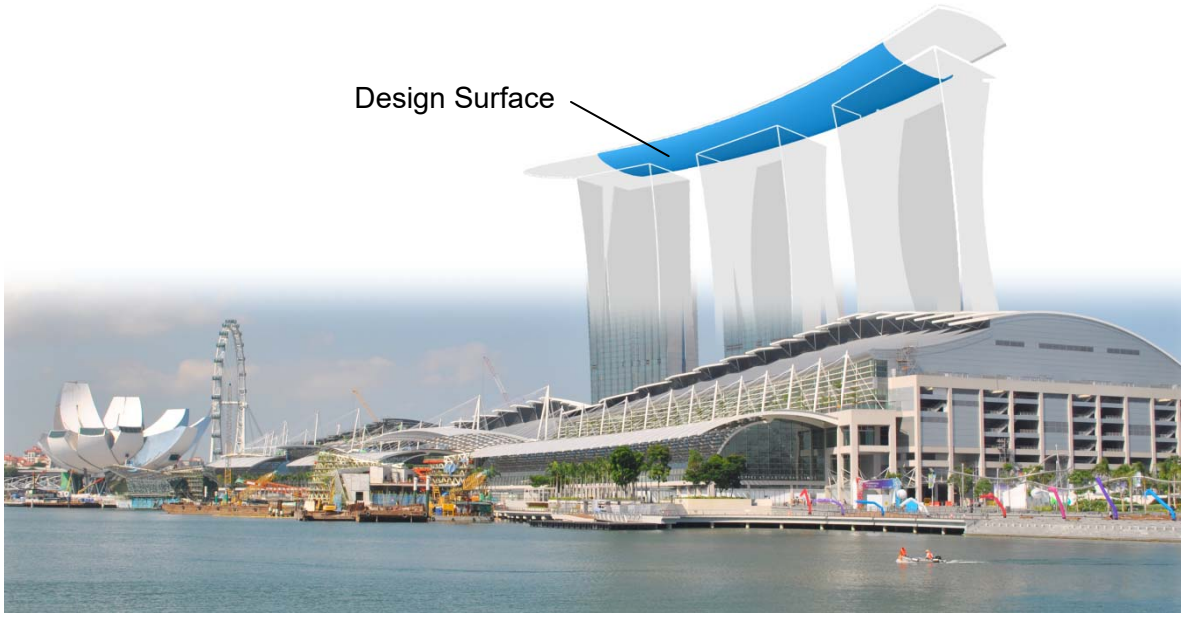
Revit-Native vs. Dynamo-Enhanced Paneling Techniques

The Design Surface

To compare Revit-native and scripted techniques with Dynamo, we will examine the underside of the Sky Park at the Marina Bay Sands hotel in central Singapore. (http://www.msafdie.com/#/PROJECTS/MARINABAYSANDS)

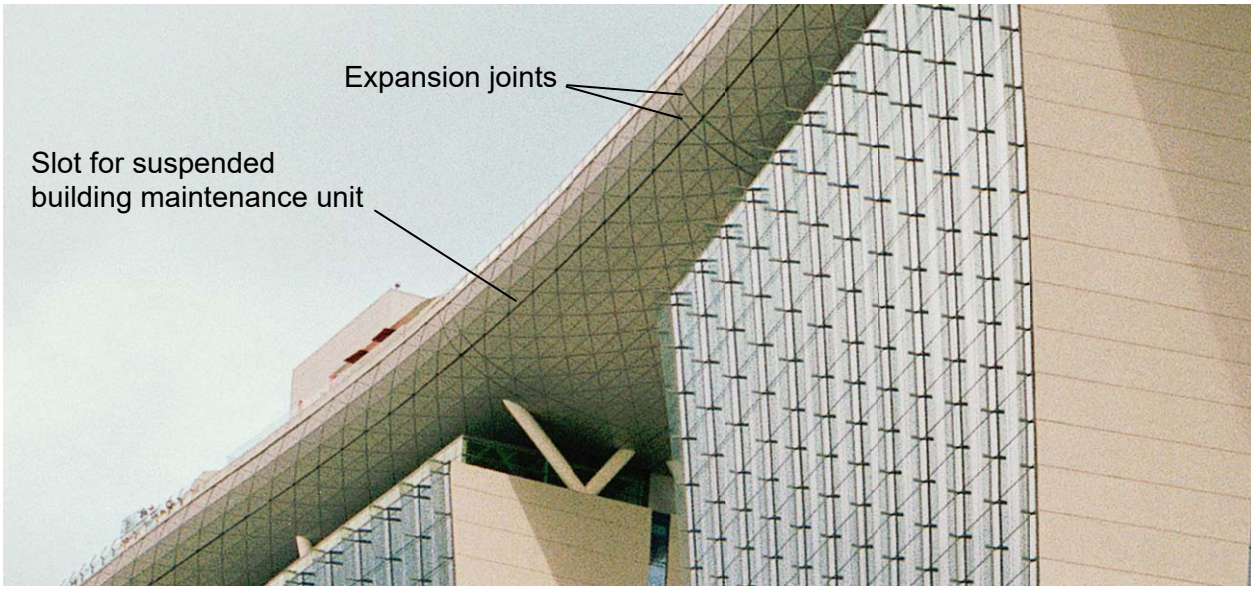
THE MARINA BAY SANDS IN SINGAPORE DESIGNED BY SAFDIE ARCHITECTS AND COMPLETED IN 2010.

We will save discussion of special edge conditions for the next section, so for now we will concentrate only on the central segment of the SkyPark underbelly, which is U-shaped in section and follows an arc in plan.



THE DESIGN SURFACE IS THE CENTRAL PART OF THE U-SHAPED UNDERSIDE OF THE SKYPARK.

The design intention is for the SkyPark shell to read as a diagrid, but the paneling scheme must also allow for cross-wise and longitudinal joints, some of which are locations for expansion joints and for building maintenance unit connections. Triangular panels are used, sized for construction in Singapore. Diagonal seams between panels are emphasized with a 10 cm gap, while most orthogonal gaps are half as wide.



THE PANELING SCHEME WAS CHOSEN TO BE A DIAGRID THAT WOULD ALLOW FOR RECTILINEAR SEAMS.

Paneling in Revit

Paneling techniques for parametrically dividing surfaces in Revit are well documented elsewhere. The class will discuss a divided surface with a pattern-based curtainwall panel family as a means to model the SkyPark panels. Benefits of this approach include use of a Revit feature that is perhaps more widely-known than Dynamo. Drawbacks include:

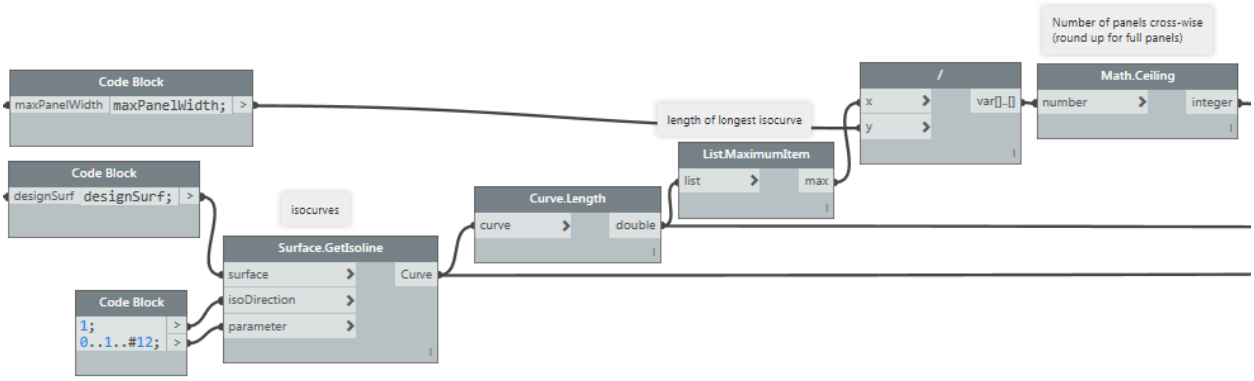
- A divided surface operates by means of the surface's parameterization, so equal-width divisions can be more difficult to achieve for arbitrary surfaces that are not planar or arc-sweeps.
- Modeling edge conditions like the prow of the SkyPark would require a different approach so that adapting the design for changes would be difficult.
- A Revit divided surface approach would be more difficult for some cases, notably for the following example, where the design surface cannot be treated like a simple rectangle, and the parametric division of the surface should not be a guide for the placement of panels.

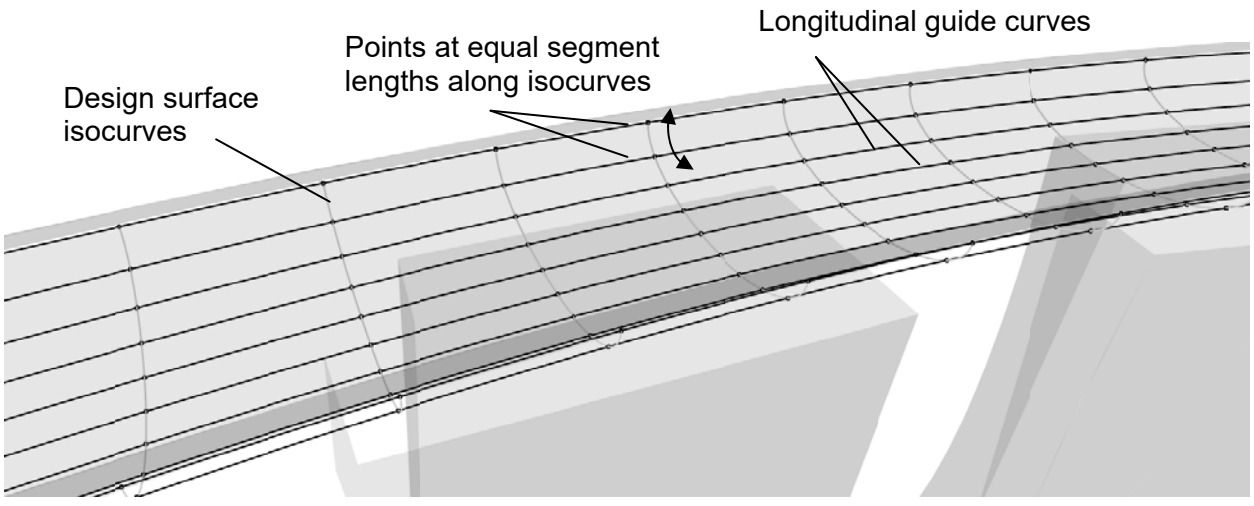
Paneling in Dynamo

For a more tightly-controlled scheme we can turn to Dynamo, in this case running as an add-in to Revit so that the results can be piped into Revit and updated as needed.

There are some handy shortcuts available through Dynamo packages such as Lunchbox that will simply create a diagrid of panels hosted on a given surface. These are a great option to shortcut fussy logic when your surface is well-behaved in the NURBSy-sense, but of course ours isn't. David Rutten has a great blog post on the parameter space of NURBS curves if you are unfamiliar (https://ieatbugsforbreakfast.wordpress.com/2013/09/27/curve-parameter-space/). Basically, since our surface is neither linear nor constructed from simple circular arcs, we've got to do this the "long way" to make sure we get what we need.

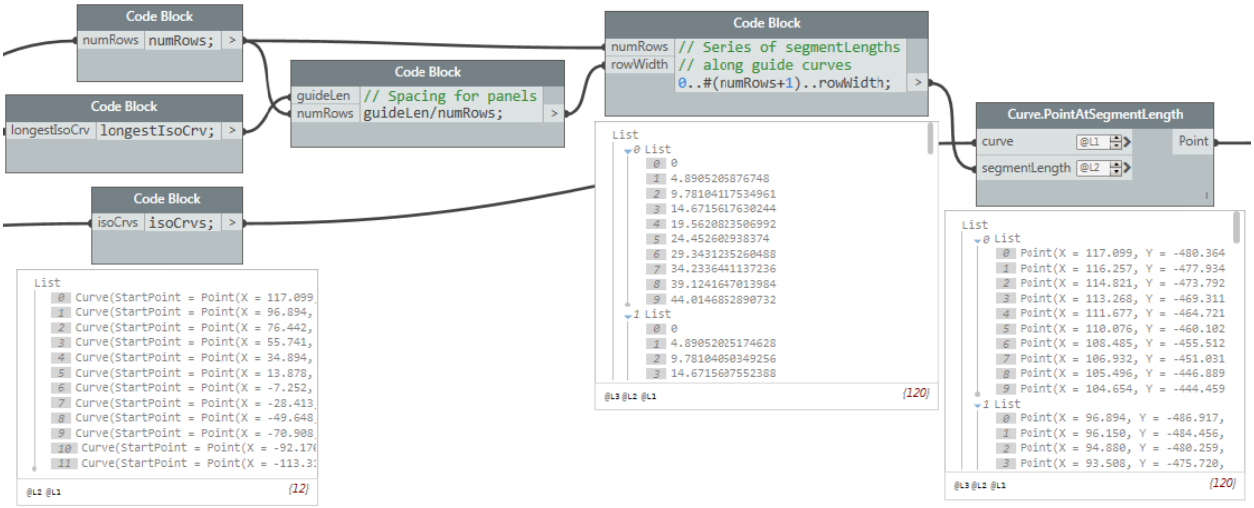
Dynamo has an easy way to find evenly-spaced points along a curve even if it's not so easy with a surface. We can use a set of curves derived from the surface to find equally-spaced points that will in turn define a set of guide rails on which to hang panels. Find cross-wise curves (gray lines below) from the parameter space of the curve Surface.GetIsoline, then use these curves to find how many full panels fit cross-wise. The choice of 12 isocurves is arbitrary.



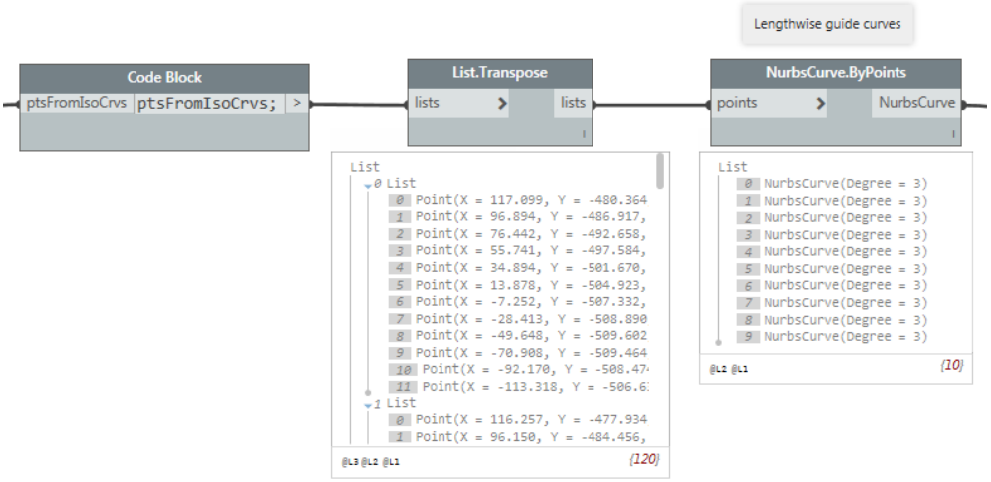


THE DESIGN SURFACE'S PARAMETER SPACE IS NOT EVENLY SPACED, SO FIRST FIND THE SURFACE'S ISOCURVES (GRAY LINES), THEN CONSTRUCT THE GUIDE CURVES (BLACK LINES) FROM EQUALLY-SPACED POINTS ALONG THE ISOCURVES.

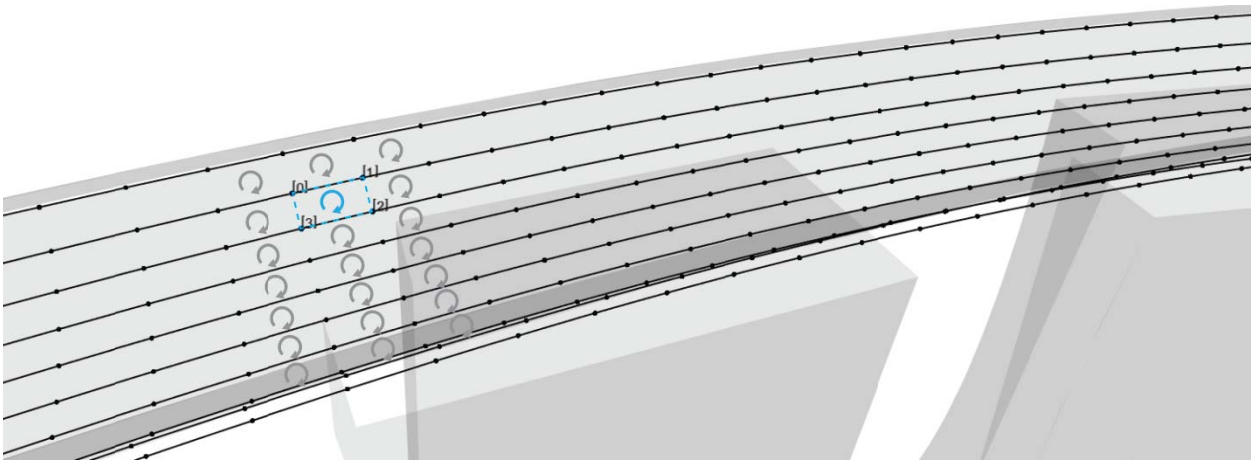
With the number of rows of panels known, create a series of segment lengths for each isocurve. The easiest way is to use series syntax (http://dynamobim.org/0-4_tips/) in a code block if that's your thing (don't be scared!). To match the list of segment lengths to the appropriate isocurve, use the List@Level feature to force the node to act at the appropriate level in the data structure hierarchy. (http://dynamobim.org/introducing-listlevel-working-with-lists-made-easier/).



You'll already see a grid of points along those isocurves. You'll need to reorganize these points with List.Transpose so that they can be stitched together to define the longitudinal guide curves. You're done with those points; the isocurves were just a means to find the longitudinal guide curves. The list of guide curves should be "flat," meaning no sub-hierarchy like the sets of points that were used to create them.



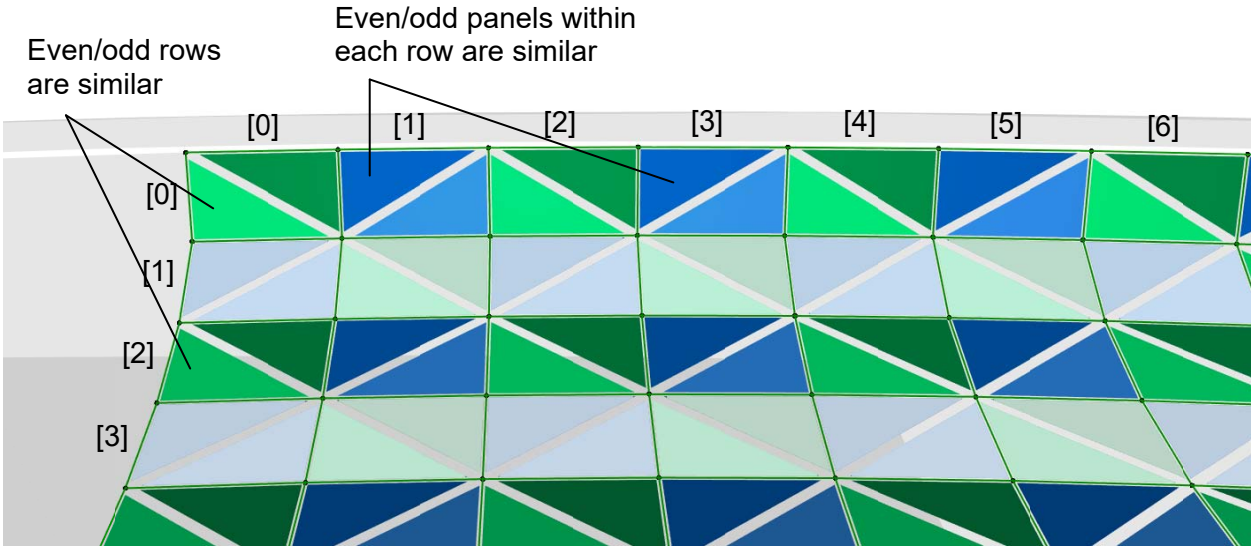
Recalling the process to fit panels by their maximum width along the isocurves, use a similar technique with the guide curves to fit panels according to their maximum length. When you find the series of segment lengths, find the points at those lengths along the guide curves. As a result, you will have a 2-dimensional grid of points strung along those guides that will become corner points for panels.



FROM AN ORGANIZED 2-DIMENSIONAL SET OF POINTS THAT DESCRIBE THE CORNERS OF RECTANGULAR PANELS, REORGANIZE THE POINTS IN SETS OF 4, ONE PER PANEL.

The next step sounds complex, but "there's a node for that." To translate the 2-dimensional grid of points to sets of 4 points describing each single cell of the grid each, use one of several similar custom nodes available on the package manager. For this paneling routine, it's better that the node return sets of points that are themselves organized by row, so PointList.QuadsFromGrid from the Ampersand package works well. Whatever method you chose should return sets of 4 points that are ordered in a consistent direction, either all clockwise or all counter-clockwise.

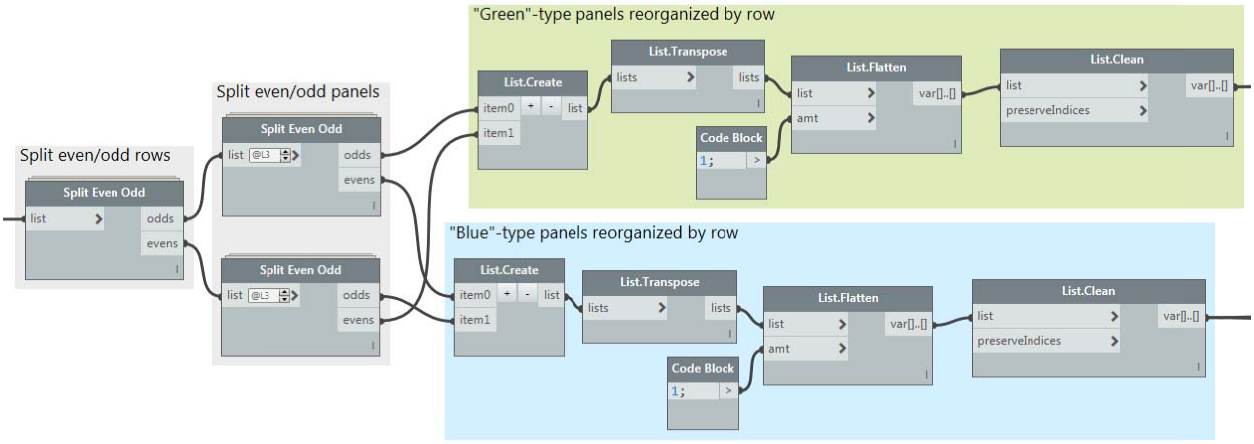
Forecasting a bit, we know that we will need to find triangular panels within these rectangular panels that have alternating diagonal divisions as in the diagram below. The false color diagram below shows that each rectangular panel should be divided into a pair of triangles whose diagonals oriented one of two ways, represented by blue and green sets. It's a checkerboard problem, and we can use the structure of the point data to manage it neatly.



ANTICIPATING THE FINAL DESIGN, THE SETS OF 4-POINTED PANELS ARE DIVIDED INTO EVEN/ODD ROWS THEN EVEN/ODD PANELS PER ROW.

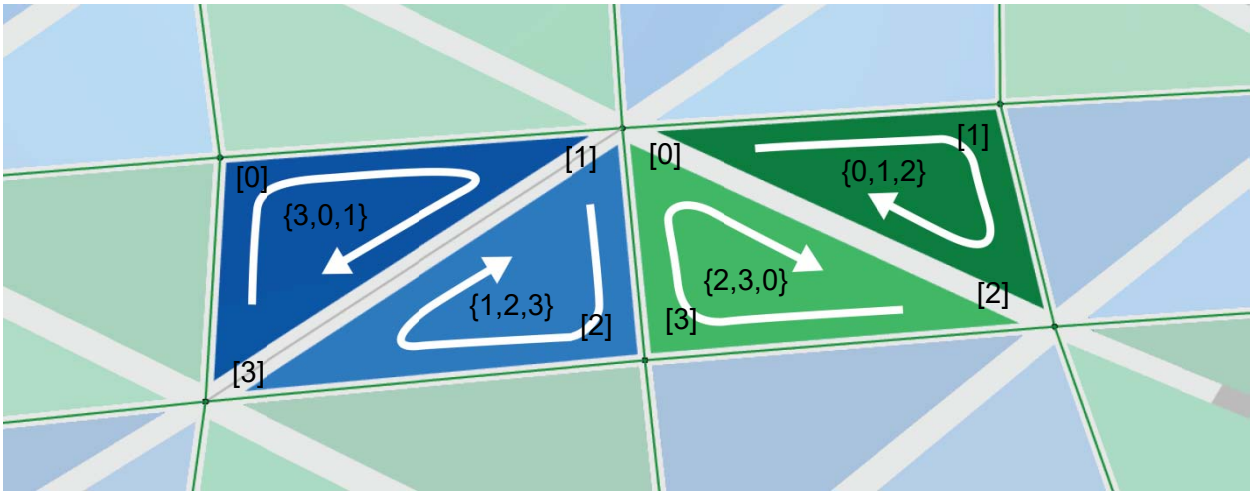
Since splitting a list into even and odd items will be a repeated task, it's a good candidate for a custom node. One for this purpose in the Ampersand package called "List.SplitOddEven". Use it once to split even and odd rows from one another, then once more within the row (use List@Level to choose which level in the list at which to operate) for each row. After the second column of nodes in the graph below, the four output ports represent the four types of panels:

- Odd row / odd panel = green type
- Odd row / even panel = blue type
- Even row / odd panel = green type
- Even row / even panel = blue type



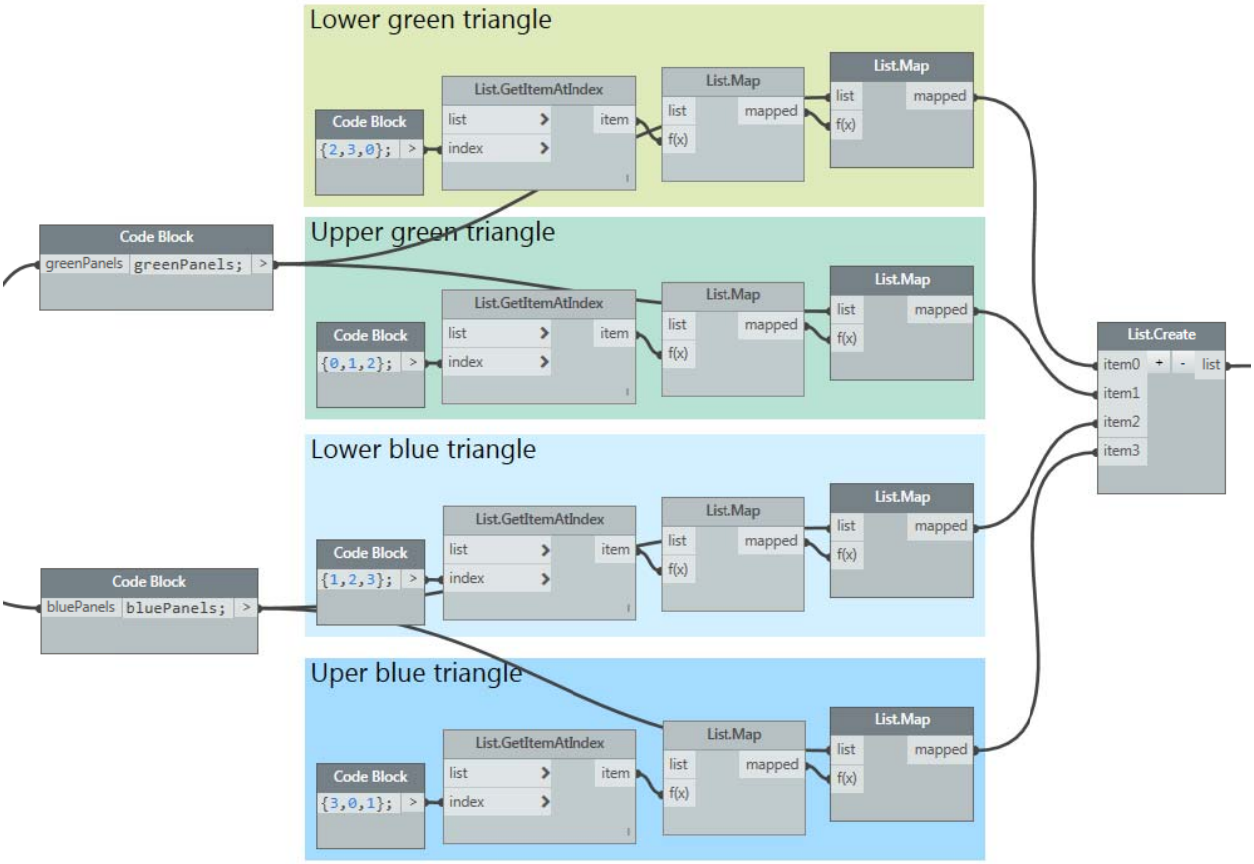
What follows is not entirely necessary, but it should be useful. For each set of rectangular panel points, we can reorganize them by row using a trick with List.Transpose.

Once the rectangles are divided into blue and green types, find the triangles inside them by choosing sets of vertices. Take care to make sure each triangle is wound the same way (clockwise below). To make the next step simpler keep the diagonal last.



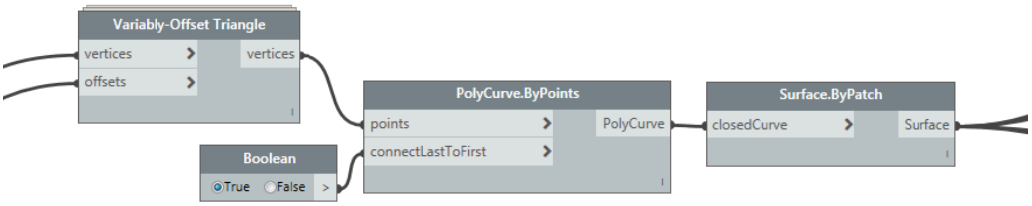
POINTS FOR TRIANGULAR PANELS ARE FOUND BY SELECTING SETS OF 3 POINTS FROM THE 4-POINT PANELS. CARE IS TAKEN TO MAKE SURE THAT EACH SET IS "WOUND" IN A CONSISTENT DIRECTION.

List.Map is used to change the level at which the node List.GetItemAtIndex operates. The point data are carefully organized into hierarchical lists of rows and panels, so to choose 3-point sets from within the 4-point rectangular point sets, dive down two levels into the data structure. Read more about list mapping here (http://dynamoprimer.com/en/06_Designing-with-Lists/6-3_lists-of-lists.html) if you are unfamiliar. The diagram on the previous page shows why particular sets of indices are used in each case.



Once all the triangular panel points are defined, throw them in a single list now that we no longer have a need to distinguish them by type. The list of panel points is not perfectly ordered to start from the upper left corner of the design surface to bottom right corner, but the process to do so is similar to the List.Transpose trick shown previously.

We're not done quite yet; panels formed from these sets of points would be seamless, but the design calls for gaps between the panels—gaps which vary by seam. This is also a good candidate for a let's-only-figure-it-out-once kind of process, a.k.a. "custom node."



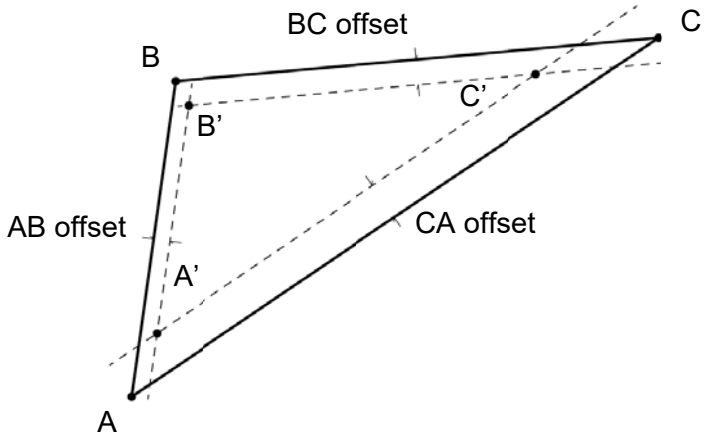
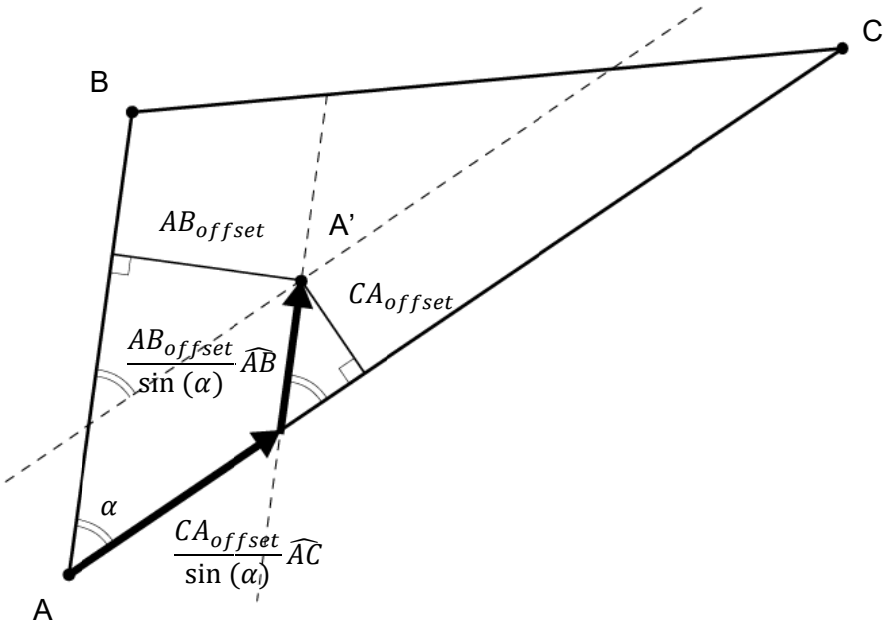


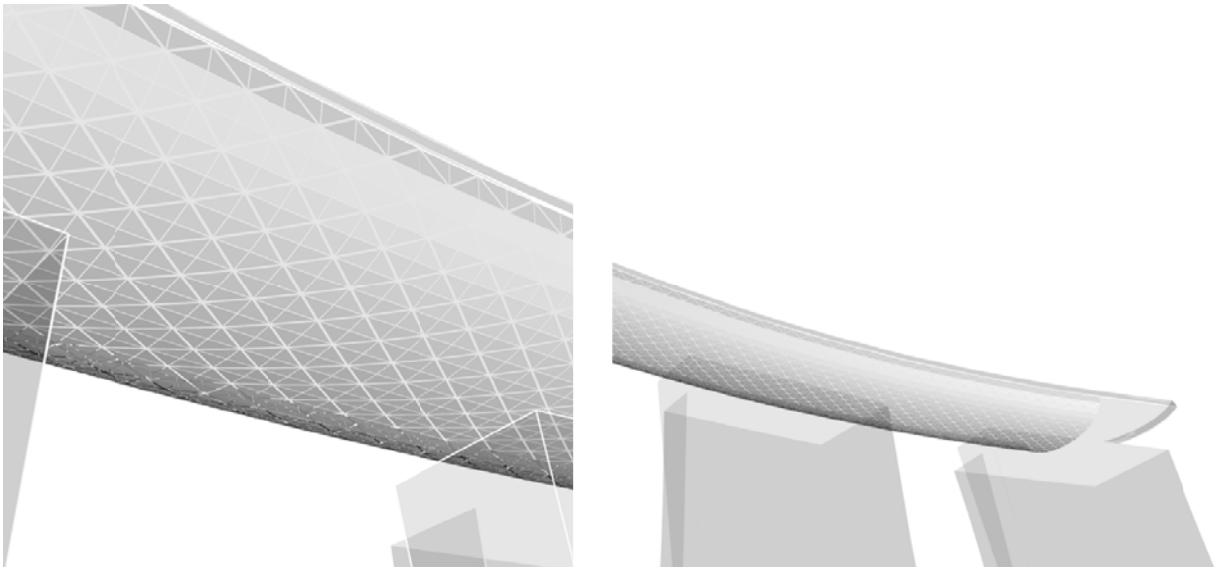
DIAGRAM FOR THE VARIABLY-OFFSET TRIANGLE REQUIREMENT.
WE FIRST FOUND TRIANGLE ABC, BUT WE NEED TO FIND TRIANGLE A'B'C'.

For this process, a custom node called "PolyCurve.VariablyOffsetTriangle" is in the Ampersand package. To use it, specify the points {A,B,C} and the offsets per side {AB, BC, CA}, and it will return the points {A', B', C'}. The AB and BC offsets will be half the normal gap width while the CA offset will be half the diagonal gap because we made the side CA consistently the diagonal earlier. Use the node with blissful ignorance or check out the diagram to see how it works.



THE POINT A' IS RELATED TO A BY THE ANGLE α AND THE OFFSETS FOR THE SIDE AB AND AC.
THE VARIABLY-OFFSET TRIANGLE NODE WORKS BY ADDING THESE VECTORS TOGETHER.

As always, flex your model to find where it breaks by adjusting the inputs through reasonable ranges. This will help you uncover logic errors.



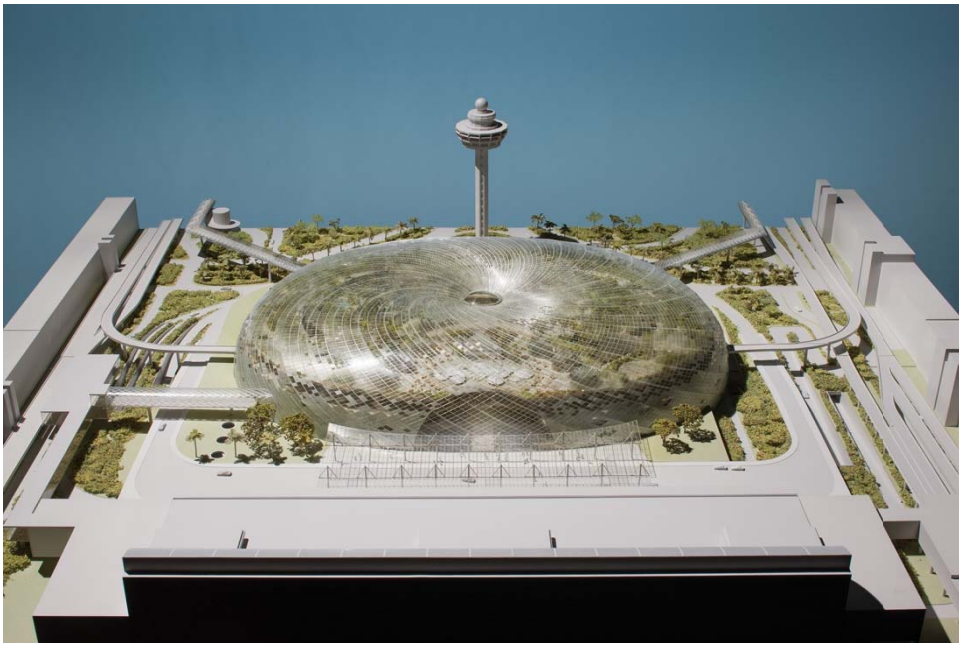
PANELING SCHEME FOR THE MARINA BAY SANDS SKYPARK UNDERBELLY.

PANEL INSTALLATION FOR THE MARINA BAY SANDS SKYPARK.

Accounting for Edge Conditions

A Challenging Design Surface

The following sections will make reference to Jewel Changi Airport, now under construction, which is a mixed-use building and a future centerpiece for Singapore's award-winning Changi International Airport. Project Jewel features a large, asymmetrical design surface which is both a high-performance weather barrier and structural system. The envelope is tuned for constructability and for climate control; the shell of glass and aluminum panels and structural members provides shading to the interior while admitting a minimum level of daylighting for the large gardens at the building's center. The majority of the glass and steel shell will be experienced by visitors from both the outside and inside.



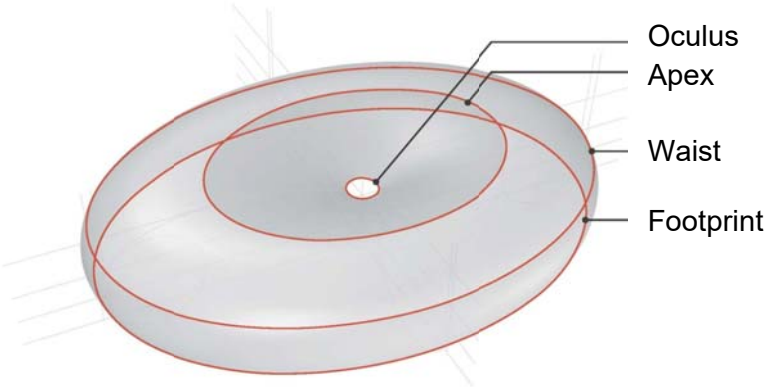
PHYSICAL MODEL OF PROJECT JEWEL, NOW UNDER CONSTRUCTION.
(MODEL IMAGE COURTESY OF SAFDIE ARCHITECTS.)

The exterior of Jewel is shaped as an elliptical glass toroid (see also: donut, doughnut, bagel) with a hole in the middle, the "oculus," which is the location of a 40-meter-tall indoor waterfall, the "Rain Vortex." The water feature tumbles to the bottom of a bowl-shaped Forest Valley, an eight-acre terraced garden, which is a new offering to the landside amenities of Changi Airport.

The design challenge for the Jewel Changi Airport envelope design is characterized by many design constraints such as existing site conditions, airport regulations, structural considerations, and proper response to Singapore's tropical climate. Among the design challenges are:

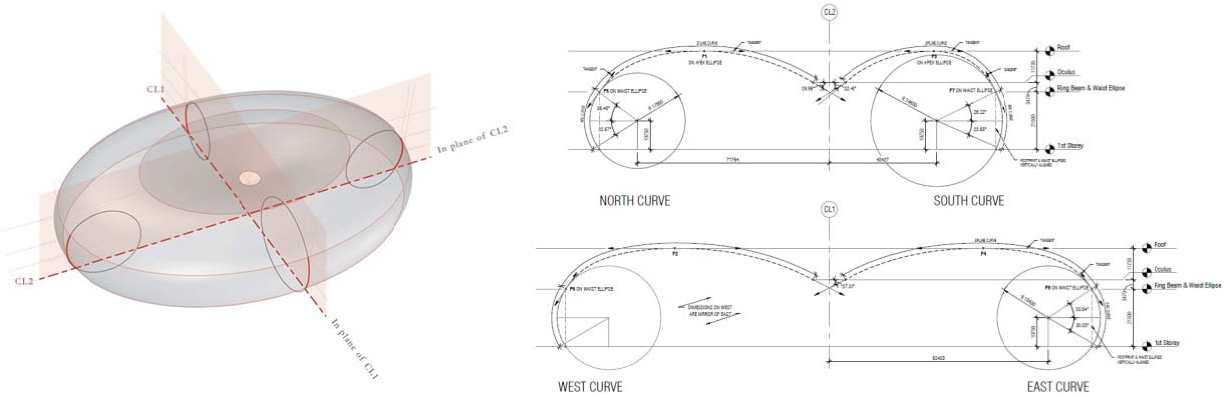
- The Changi Airport Skytrain, a people mover system (PMS), passes through the site, through the very center of the building, and it should not be moved. This demands that the oculus be off-center so that the waterfall will not fall onto the PMS.

- For structural reasons, the exterior envelope must be symmetrical, planar, and level at the intersection with the ground (the “footprint” datum) and at the roof’s critical springing point and structural ring beam (the “waist” datum).
- To set the maximum surface height in accordance with airport radar system requirements, the highest point along any radial section should be at the same elevation (the “apex” datum).
- The oculus must be a planar circle parallel to the ground plane to facilitate the design of the complex water weir edge design for the Rain Vortex.



DATUMS AND CONSTRAINTS FOR THE DESIGN SURFACE.

These criteria for the design surface were encoded into splines in two orthogonal reference planes passing through the center of the oculus. A network surface was created in the CAD tool Rhinoceros from these splines, and the surface was used throughout the project to govern the locations of all other building elements including façade panels and structural members.



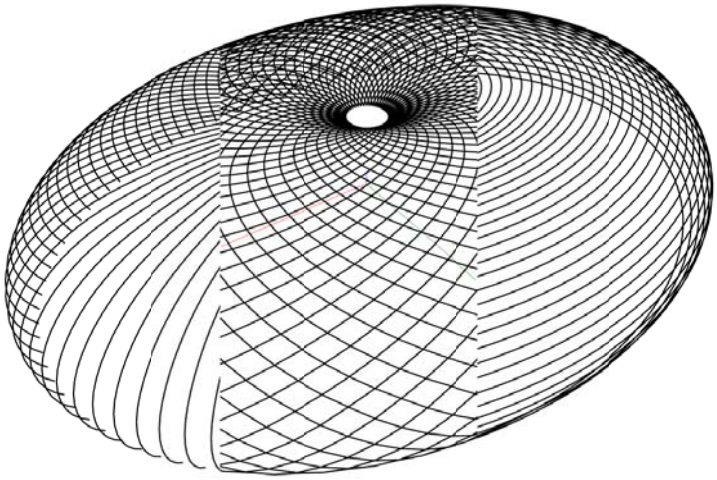
DESIGN SURFACE SETOUT DIAGRAMS FOR REPRODUCIBILITY

To coordinate with consultants and contractors, Safdie Architects documented the surface with precise geometric descriptions to enable anyone to reproduce the surface. To limit the scope of this class, we will take the design surface as a given.

Scheme 1: Hoops and Biases

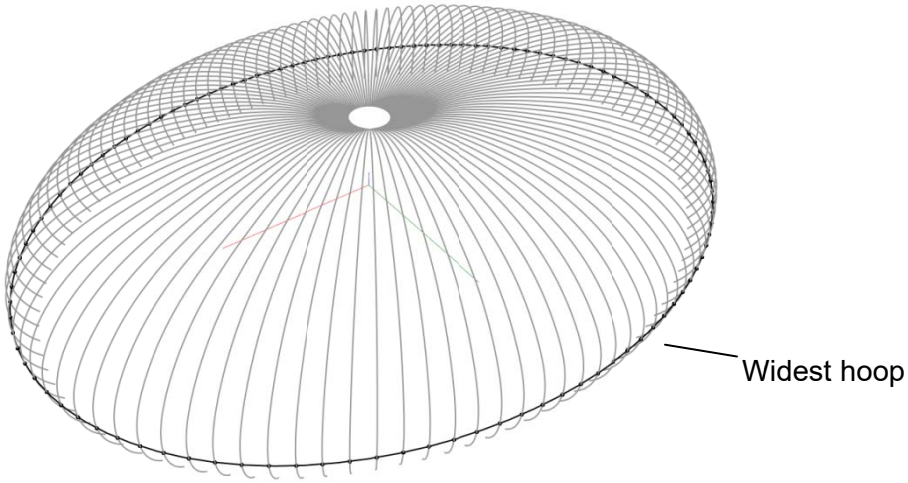
The reader will recognize similarities between some of the more basic processes described for the Marina Bay Sands surface and this problem, so the explanation will focus on the differences. Major differences include a new set of design constraints and the topology of the surface: it can't simply be treated like a rectangle. We start with a simplified version of the paneling scheme.

Note the definition of some terminology below that we will use to refer to lines along the surface which may be divisions. Similar terminology shows up with the ply of rubber strands in tires.



DEFINITION OF THE TERMS, FROM LEFT TO RIGHT: RADIALS, BIASES, AND HOOPS.

As before, we will specify a maximum width and a maximum length (height) for panels, and we find the most extreme situation at the widest hoop. As before, divide the hoop length by the max panel width and round up, then find the series of points at equal segment lengths around the hoop, which together with the center point of the oculus, sets the position of the radial members.

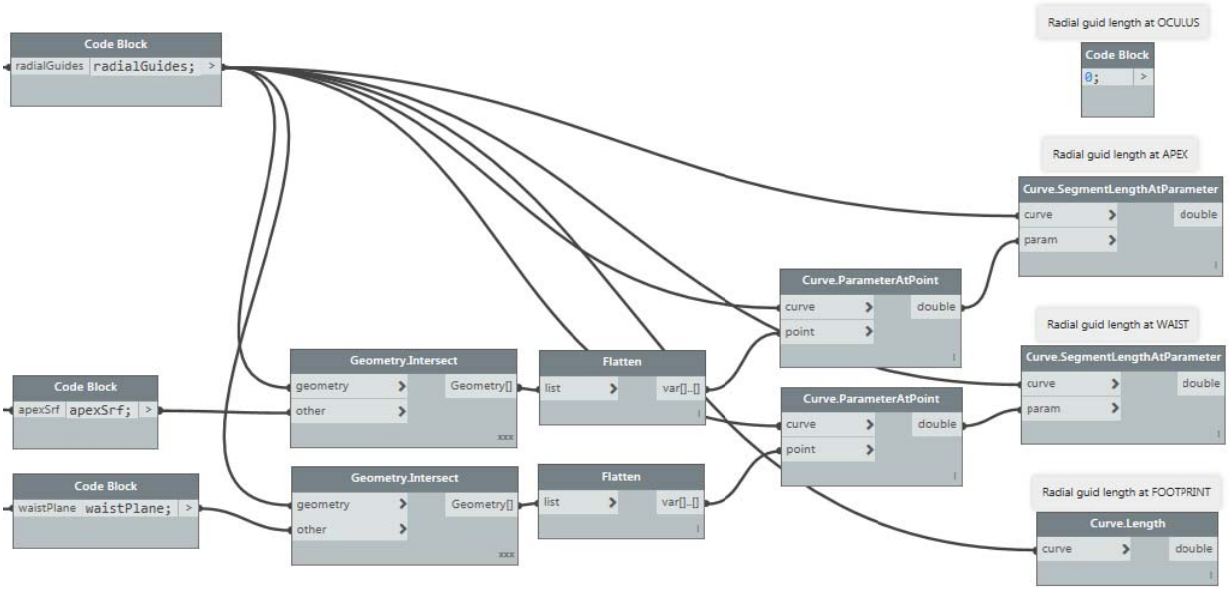


RADIALS ARE SPACED ACCORDING TO THE MAXIMUM PANEL WIDTH AS LAID OUT AROUND THE LARGEST HOOP.

The radials will be spaced evenly at the widest point, but not at the oculus because the form is not circularly symmetric and because the oculus is not in the center.

Unlike for Marina Bay Sands, to fit panels by their maximum length (or “height”), we must obey the design constraint that particular hoop-oriented seams between panels must be perfectly planar at specified levels: the oculus, apex, waist, and footprint. Because the design surface is not circularly symmetric, this won’t happen by accident. To align these seams, we effectively treat the design surface between each datum as a separate region for paneling. Equivalently, we find points along the radials in separate segments.

We find the distance along each diagonal to the point at which the radial crosses each of the four datums. The radials are organized so that they begin at the oculus, so the distance to the oculus datum is always 0. The distance to the footprint is the full length of the curve, and the distances to the intermediate datums are found through geometric intersection between each radial and a plane or surface representing the waist and the apex.



We will use series syntax inside a code block to neatly divide each of the curve segments into evenly-spaced points. Series syntax will save a lot of math and a lot of extra nodes and wires. In particular, the formulation “start..#number..interval” will be handy. Others of these are documented in the Dynamo Help files that install with the product, accessed via Help/Samples/Core/CoreRangeSyntax.

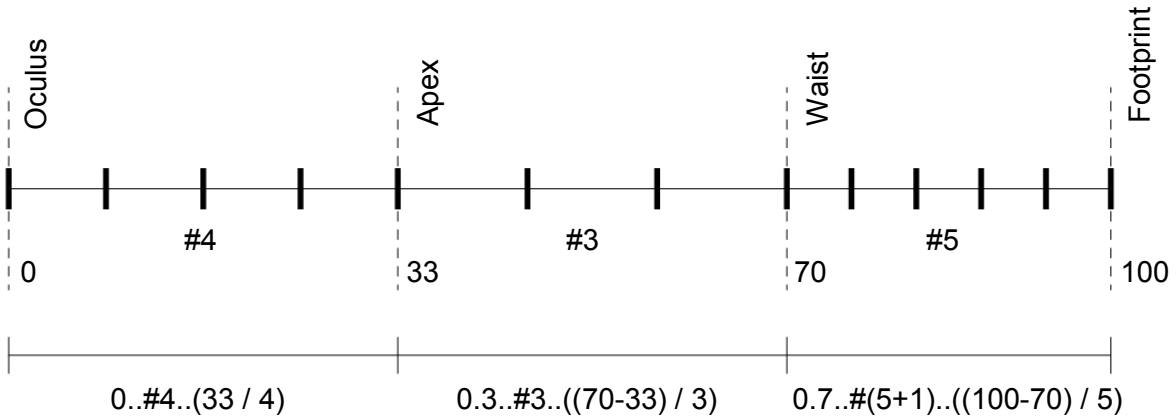
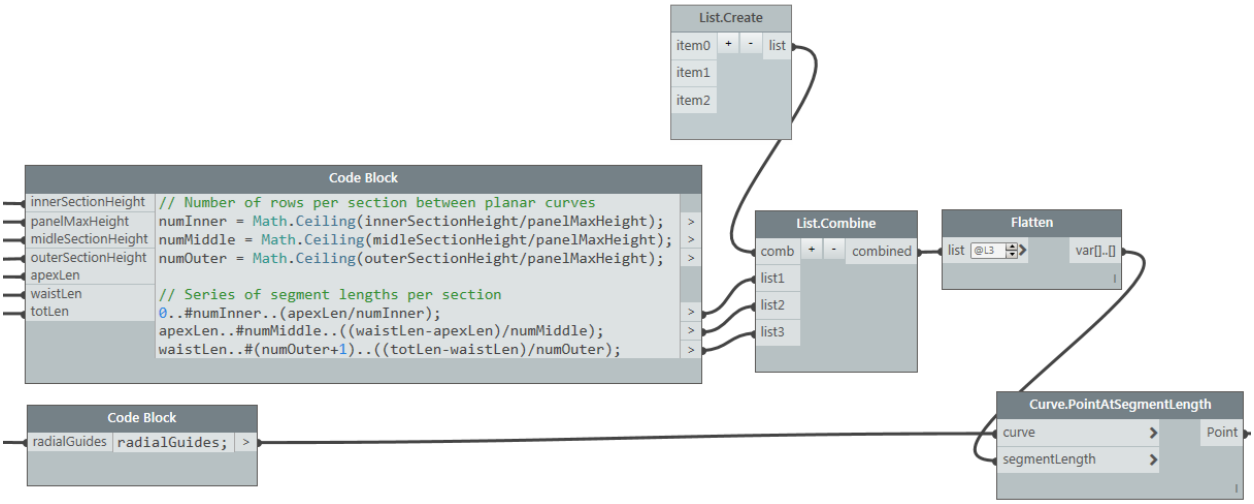


DIAGRAM FOR CONSTRUCTING A SERIES OF SEGMENT LENGTHS ALONG EACH RADIAL WITH THE SERIES SYNTAX (START..#NUMBER..INTERVAL). DIFFERENCES IN THE SPACING OF CURVE DIVISIONS IS EXAGGERATED FOR CLARITY; IN PRACTICE, DIVISIONS SHOULD BE AS SIMILAR AS POSSIBLE

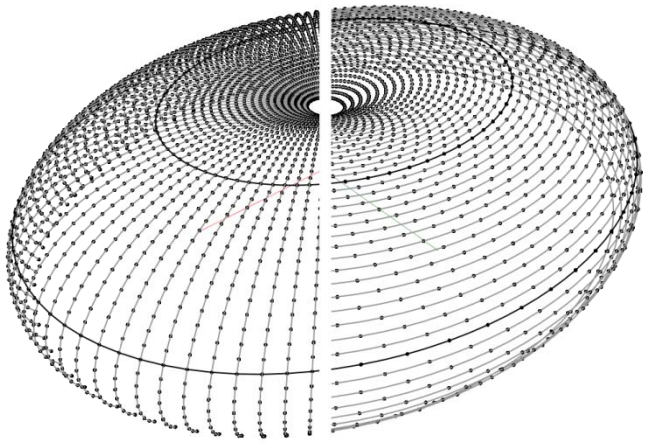
In Dynamo, this part might look intimidating, but the diagram above should help you decode the logic. In this case, one big code block is easier to read than a collection of little nodes and the spaghetti wires they would generate. After the series are generated, they are reorganized into a unique series of segment lengths per radial, and those segment lengths are used to find points.



To produce the values shown in the diagram, the in the input quantities in the code block would have these values:

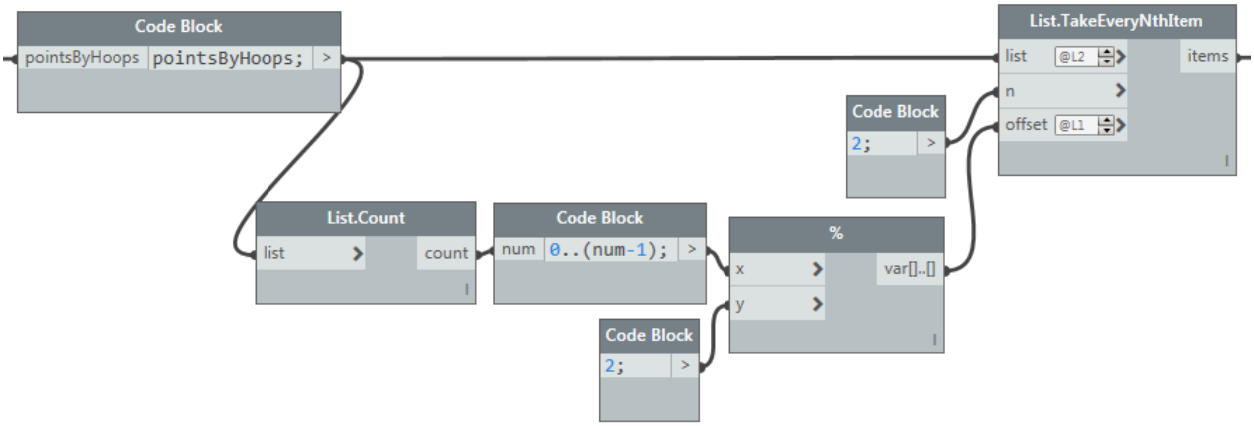
numInner	4
numMiddle	3
numOuter	5
apexLen	33
waistLen	70
totLen	100

Though not necessary for this paneling scheme, use List.Transpose to reorganize the points by hoops rather than by radials. This will make more of our logic reusable when we introduce the final paneling scheme.

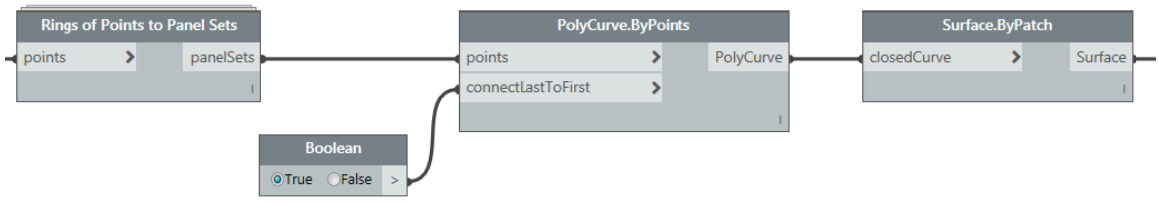


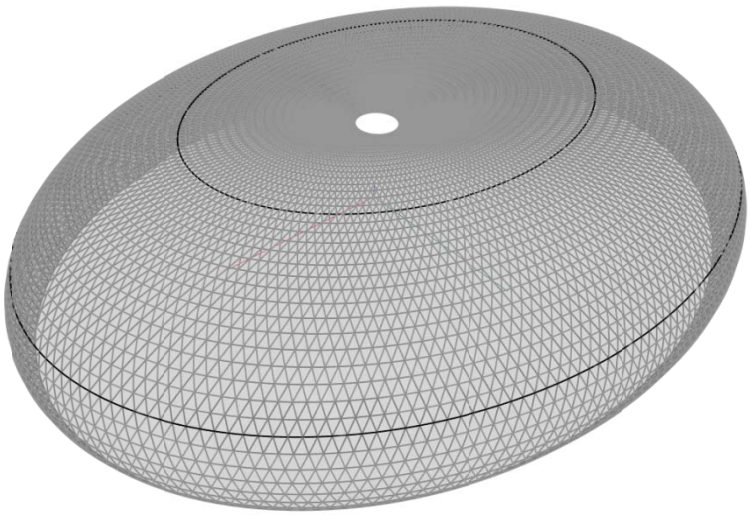
REORGANIZE POINTS BY HOOPS, THEN SELECT OFFSET SETS OF EVERY-OTHER POINT

Here is another trick (in comparison to the method used to separate the rectangular panels for Marina Bay Sands) to choose every other point in every other hoop. Use List.TakeEveryNthItem, with n = 2 for "every other," and a list of offsets of alternating 1s and 0s. If you are unfamiliar, the % node is a "modulo" operator, which gives the remainder after division, in this case by 2.



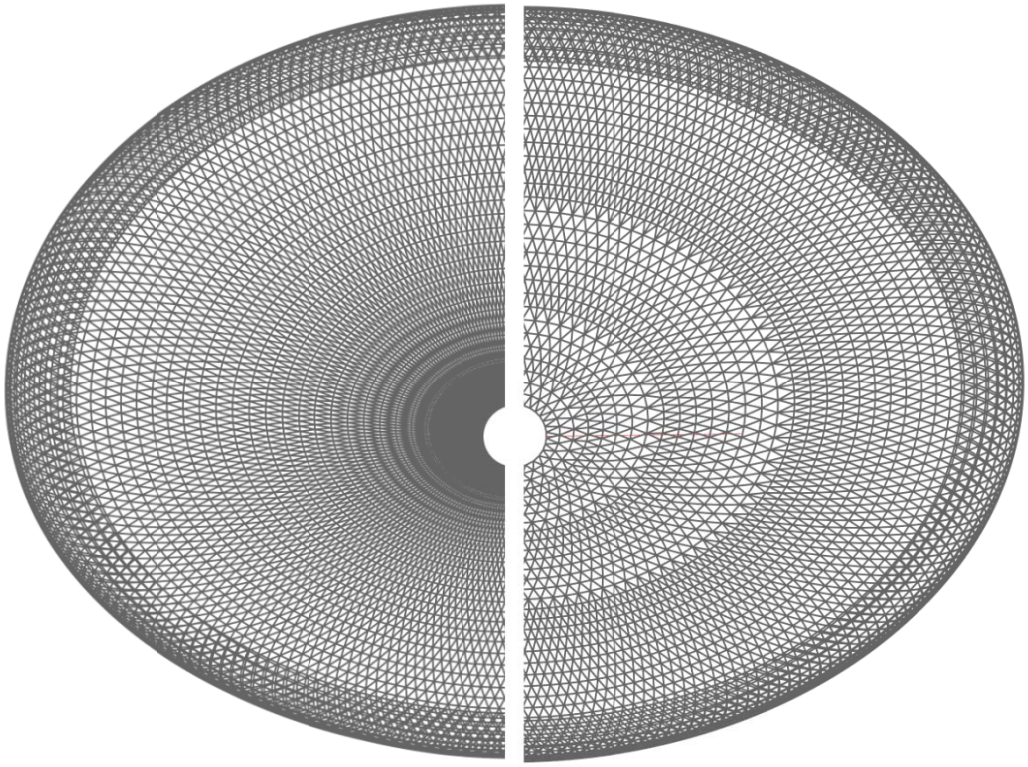
Finally, all that is left is to reorganize the rings of points into 3-point sets describing triangular panels. The process is similar to the Marina Bay Sands method except that the lists should wrap around from end to start again because each row is a ring.





THE SIMPLE VERSION OF THE JEWEL PANELING SCHEME, SHOWING IN BLACK THE APEX AND WAIST DATUMS WHERE THE PANEL SEAMS ARE PLANAR AND PARALLEL TO THE GROUND.

But that's not the scheme with which Jewel will ultimately be built. The structure, which runs between every panel, becomes so dense near the oculus that it's almost opaque. The final paneling scheme is an adaptation of this theme, but where the number of panels per row is reduced near the oculus. The final selected design scheme privileges a more consistent panel size over the consistent reading of the biases.



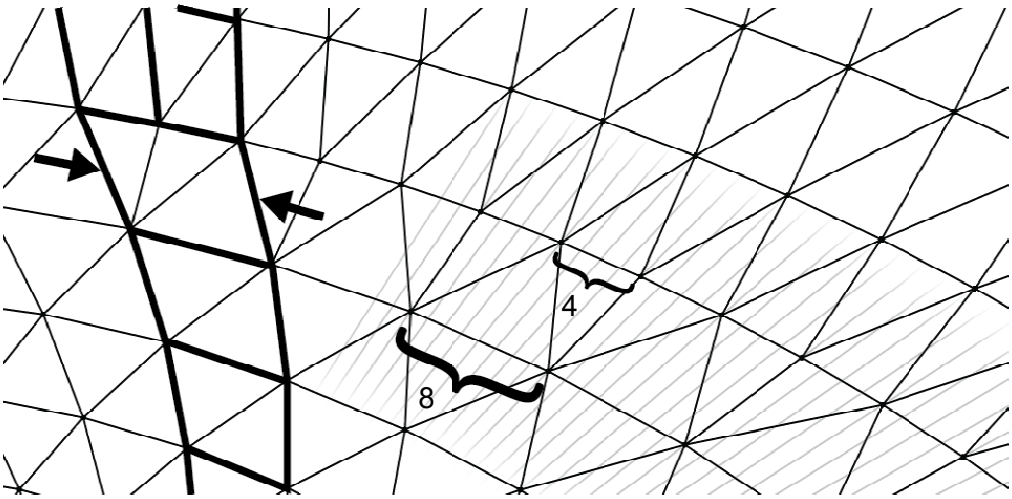
COMPARISON OF THE SIMPLIFIED "HOOPS AND BIASES" PANELING SCHEME WITH THE FINAL PANELING SCHEME, WHICH REDUCES THE NUMBER OF PANELS NEAR THE OCULUS.

Scheme 2: Adjustments for Panel Consistency

A problem with the unmodified hoops and biases scheme is that the number of panels in any one ring never varies, so there are as many panels–and as many densely-packed structural members—that meet the edge of the oculus as meet the building's much wider footprint. Many schemes were tested in the design process, and ultimately a modified version of the hoops and biases scheme was chosen with the goal to achieve overall similar panel sizes and panel ratios because the roof structure would be most often experienced from below.

All but two panels are unique in the final scheme, but it was determined that for this building, it was less costly to have great variation in the panel sizes than to have a great deal of variation in panel joint or structural node details. Here's how it works.

Three transition, or "paring" rows are introduced into the design where the number of panels is halved as you move toward the oculus. In the diagram, the transition row is marked at the left.



AT SPECIFIED ROWS, THE NUMBER OF PANELS IS HALVED, MOVING TOWARD THE OCULUS. THE PARING (TRANSITION) ROW IS MARKED AT LEFT. THIS IS ACCOMPLISHED BY DOUBLING THE NUMBER OF RADIALS BETWEEN CORNER POINTS.

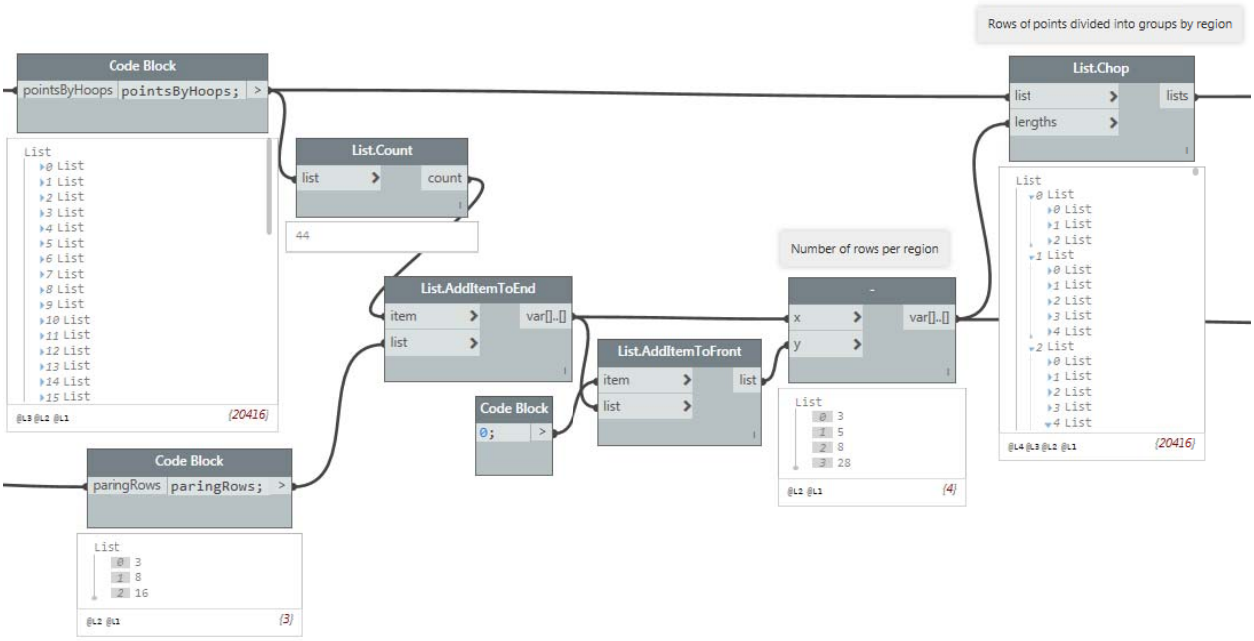
This transition is achieved by changing the number of radials between "nodes" (panel corner points) depending on the region. Here, a "region" is a set of rows between paring rows where the number of panels per row is pared down. The final design has three transition rows and therefore four "regions." To achieve this, the following tweaks are made the logic in scheme 1:

	Scheme 1	Scheme 2
# panels per row	Fewest # of panels to fit around widest ring	Fewest # of panels to fit around widest ring, rounded up to nearest 2^(# regions)
Selecting node pts	Choose every other point	Choose every 2^(#region) points
Creating point sets	Use normal logic for all rows	Use normal logic for most rows plus new logic for transition rows.

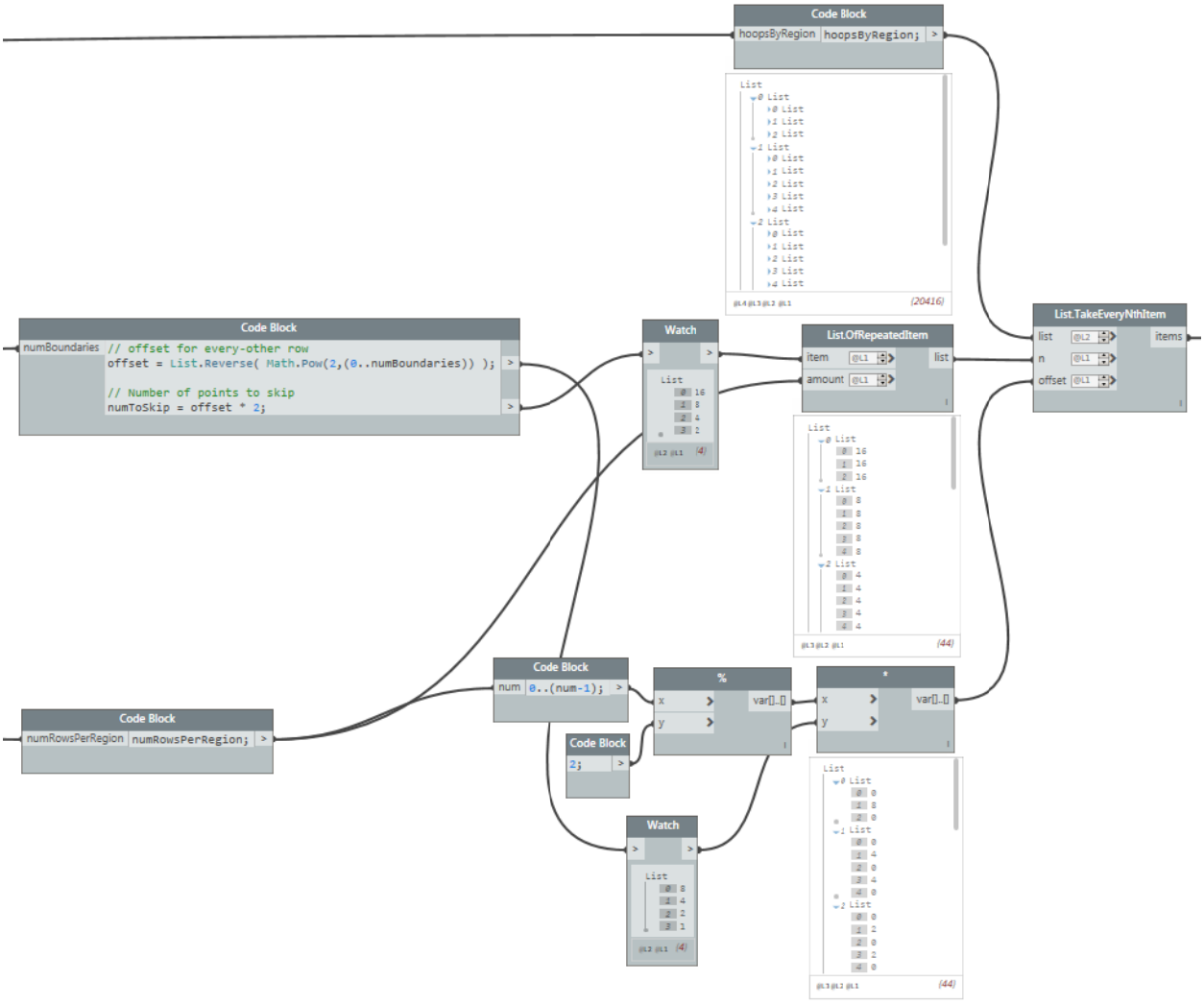
First, a new input must be added to the graph to set which rows are the paring rows. To make the specification of this quantity easier, in both examples we have been counting rows starting at the oculus, but we will number the regions starting at 1 from the footprint to make some of the math easier. Note that as we speak of "regions" now, we refer to the paneling only; this is unrelated to the sections of the surface geometry we spoke of earlier that were defined by planar datums.

The part of the logic that is the trickiest is where we select the points we want from the grid of hoop/radial intersection points because it's no longer a simple every-other trick. The goal will be to keep the script flexible to changes including changing the number and location of paring rows. To avoid hard-coding the number of regions in any way we cannot, for example, separate the rows of points into four different identical branches of the graph. Instead, we will add a layer of hierarchy to the list of points to correspond to the subgrouping of regions.

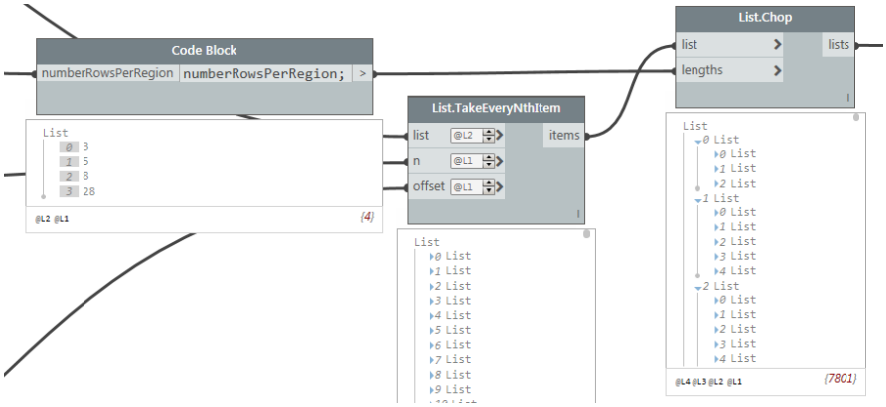
Divide the grid points (that are organized by hoops counting outward from the oculus as before) into sets by region. In the example, the paring rows are numbers 3, 8, and 16, resulting in a total of four panel regions. In the list of points by hoops at the beginning, sets of points per hoop reside at level @L2. In the list of points by hoops and regions at the end, @L2 corresponds to the hoop and the new level @L3 corresponds to the regional subgrouping.



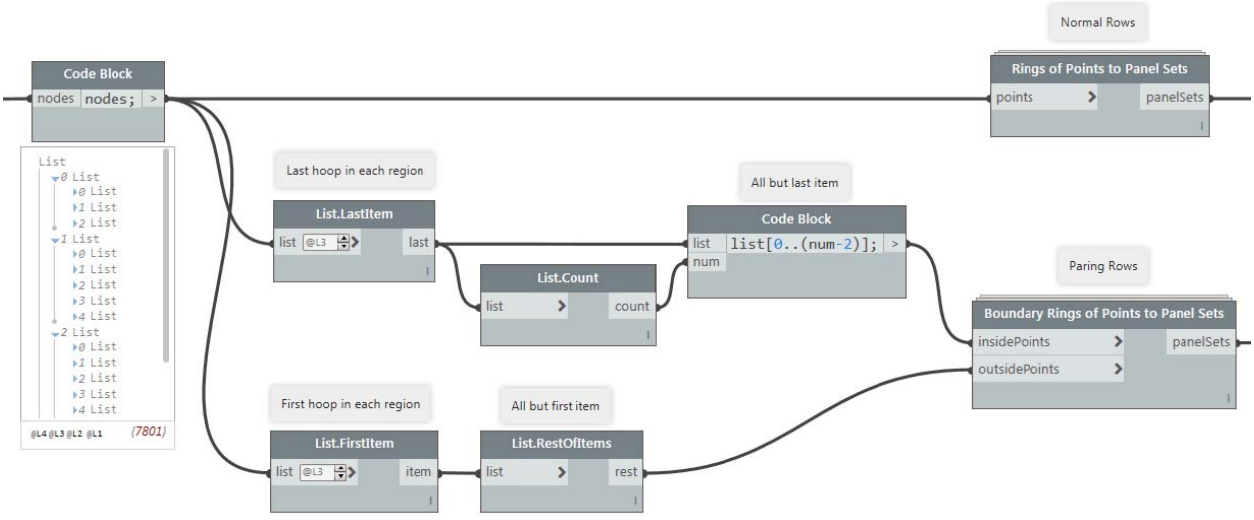
Next, we adapt the List.TakeEveryNth trick from scheme 1 (page 16) to take every 2^(number_of_region), meaning every 2nd, every 4th, every 8th, and so on, rather than strictly every other. As before, we use List@Level options on the input ports of List.TakeEveryNth to make sure that it acts where we want it to act in the incoming data structures.



List@Level sometimes behaves strangely (as of the Dynamo 1.2.0 release) with multi-input nodes whereby it may match the data items correctly initially but somehow lose some of the data hierarchy. That happens in this case, where List.TakeEveryNth forgets the data structure level corresponding to the regions. The results aren't wrong, but it's not convenient to our downstream processes to leave it that way. Restore the subgroups for regions with List.Chop.



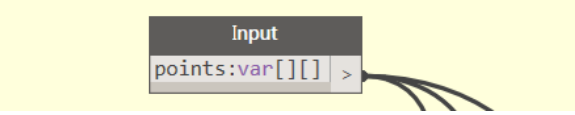
Once all of the nodes (corner points of panels) are selected, the last task, just as before, is to reorganize them into sets of three for each panel. In the graph below, the node points, now organized by region and by hoop, are turned into three-point sets for panels. All normal rows are treated exactly as in scheme 1, and new logic is introduced to take care of all paring rows. By dividing the panel point set logic this way, the graph remains adaptable to a changing number of regions.



Following this chunk of the graph, the point sets are recombined and ordered by row using the same List.Transpose and partial flatten trick shown earlier.

A word on custom nodes

There is a distinct advantage to using custom nodes to create the panel sets as shown in schemes 1 and 2. Namely, we are able to reduce the complexity of the problem to solving just the simplest case—the panel sets for just one "region" of panels. In scheme 1, we used the node "Rings of Points to Panel Sets" just once because the whole thing was just one region, and we used the same node in scheme 2 with multiple regions and it still worked. This is because the data type of the input is specified to be a 2-dimensional list using the syntax var[][] in the input node inside the custom node definition. This means that the node will only work with a 2-dimensional list, and it will dive like a heat-seeking missile into your data until it finds one. In scheme 2, we have multiple regions (4, to be exact) so the input to the custom node is a 3-dimensional list, which is just 4 two-dimensional lists. The node "replicates" 4 times. Or basically, it just works. For more, check out this blog post: http://dynamobim.org/what-does-var-mean/. The other reason to create a custom node is that this problem will likely come up again, so why solve this more than once?

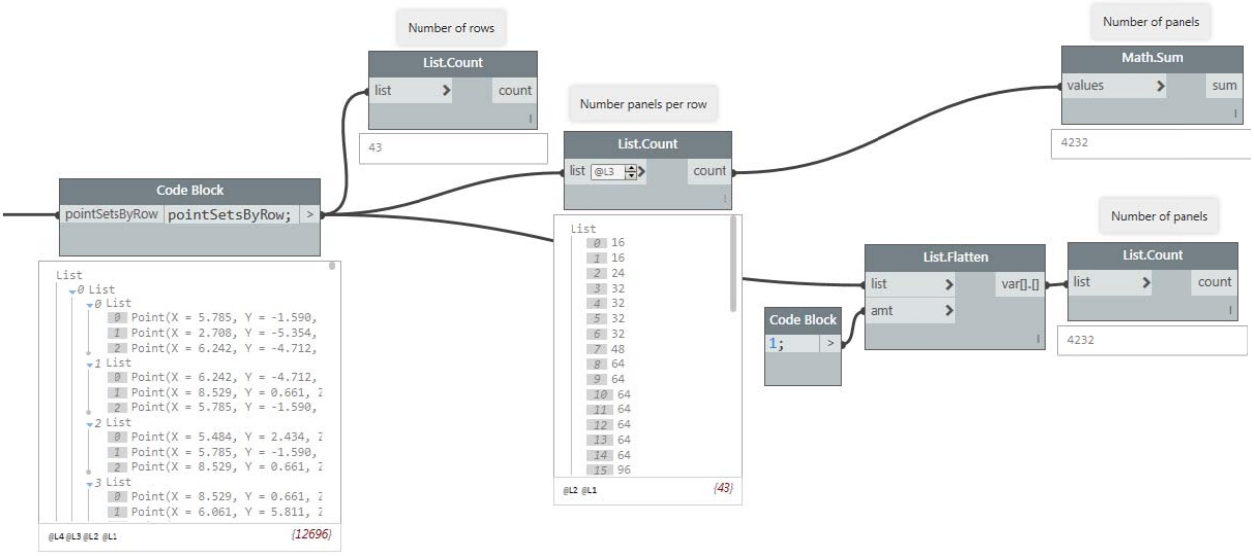


Controlling Family Types by Condition

Information

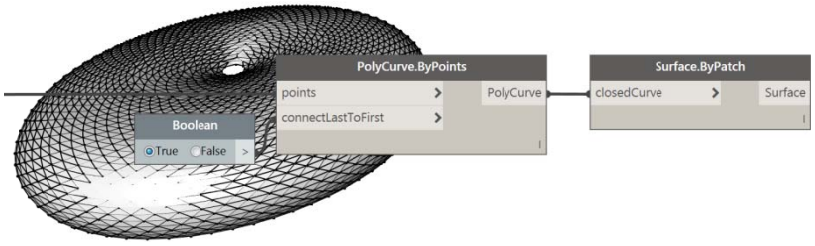
At the end of each paneling scheme above, we left off with organized sets of points, where each set describes a panel. If we ever need it, there is a lot of information embedded into the structure of the list of vertices. Taking the end result for Project Jewel, for example:

Information	Property of data structure
Panel's vertices	Items in list at level @L1
Panel's column number (roughly)	Index of list at level @L2
Panel's row number	Index of list at level @L3
Number of panel sides	List.Count at level @L2
Number of panels per row	List.Count at level @L3
Number of rows	List.Count at level @L4
Total number of panels	List.Count after List.Flatten by an amount 1



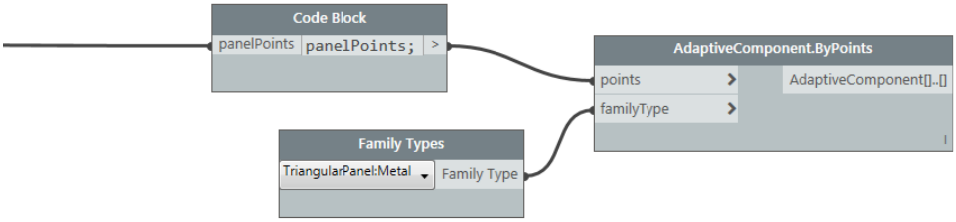
Check Your Work

The ultimate goal is to place family instances in Revit for each panel, which only requires points, but as a general rule, before sending any large amount of data to Revit, make sure it's right first. Check that the point sets actually describe panels.



Points to Panel Family Instances

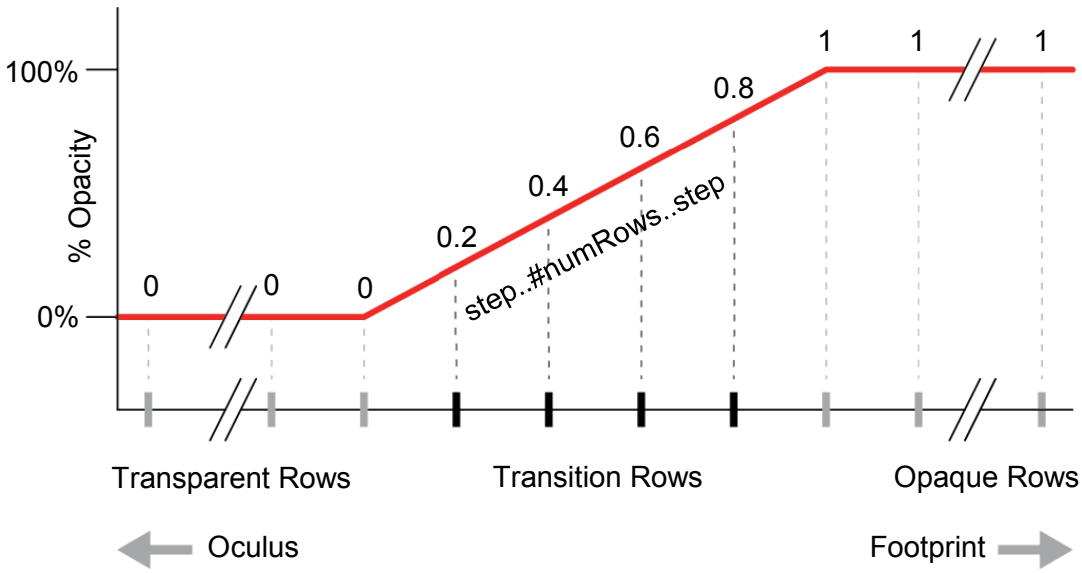
Now that we know where the panels are, the task remains to place Revit family instances with that same information. With the Marina Bay Sands example, this couldn't be easier; nearly all of the panels have the same properties save for size, so we can use Dynamo to place a lot of panels with the same type. Any deviations from panel uniformity, such as changing one panel to include a vent, is probably best done manually once your Dynamo-ing is done.



The situation is more complex for Jewel because the surface is more complex: the design calls for mostly metal panels toward the base, mostly glass where the surface is overhead, and some smooth gradation of opaque and transparent panels between the two regions. (In reality even more panel types are required for vents, for solar shading, for doors, etc., but you'll get the point by looking at just two types.)

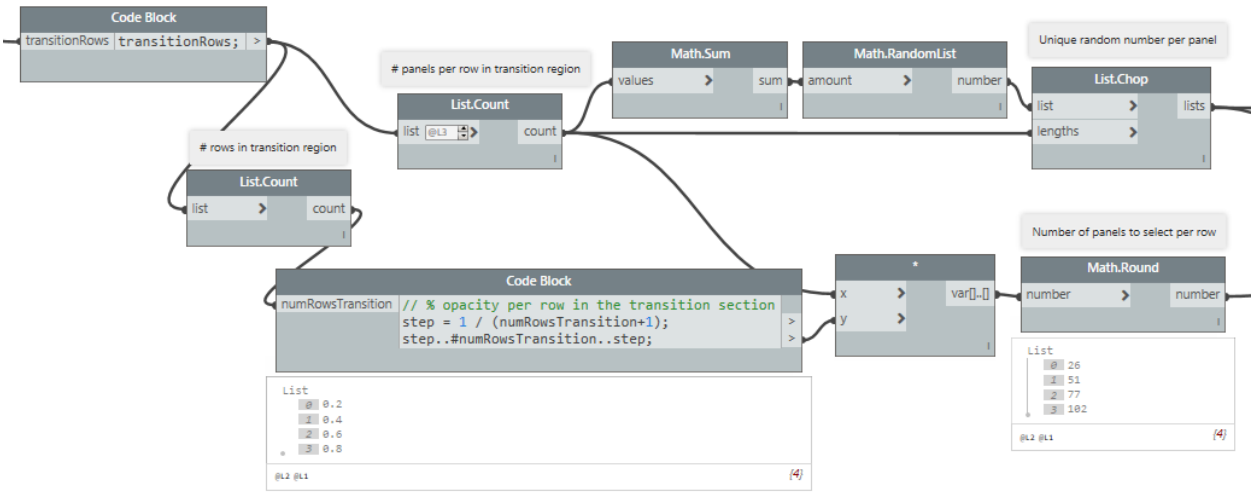
FOR PROJECT JEWEL, MORE THAN ONE TYPE OF PANEL MUST BE PLACED IN THE REVIT PROJECT. PANELS SHOULD BE OPAQUE TOWARD THE BOTTOM WHERE THE SHELL ACTS LIKE A WALL, CLEAR GLASS AT THE TOP WHEN OVERHEAD, AND PANELS MUST SMOOTHLY TRANSITION BETWEEN THE TWO REGIONS.

Panel point sets are already organized by rows, so we will start by distinguishing three regions of panel rows: one where all panels are transparent, a transition zone, and a region where all panels are opaque.

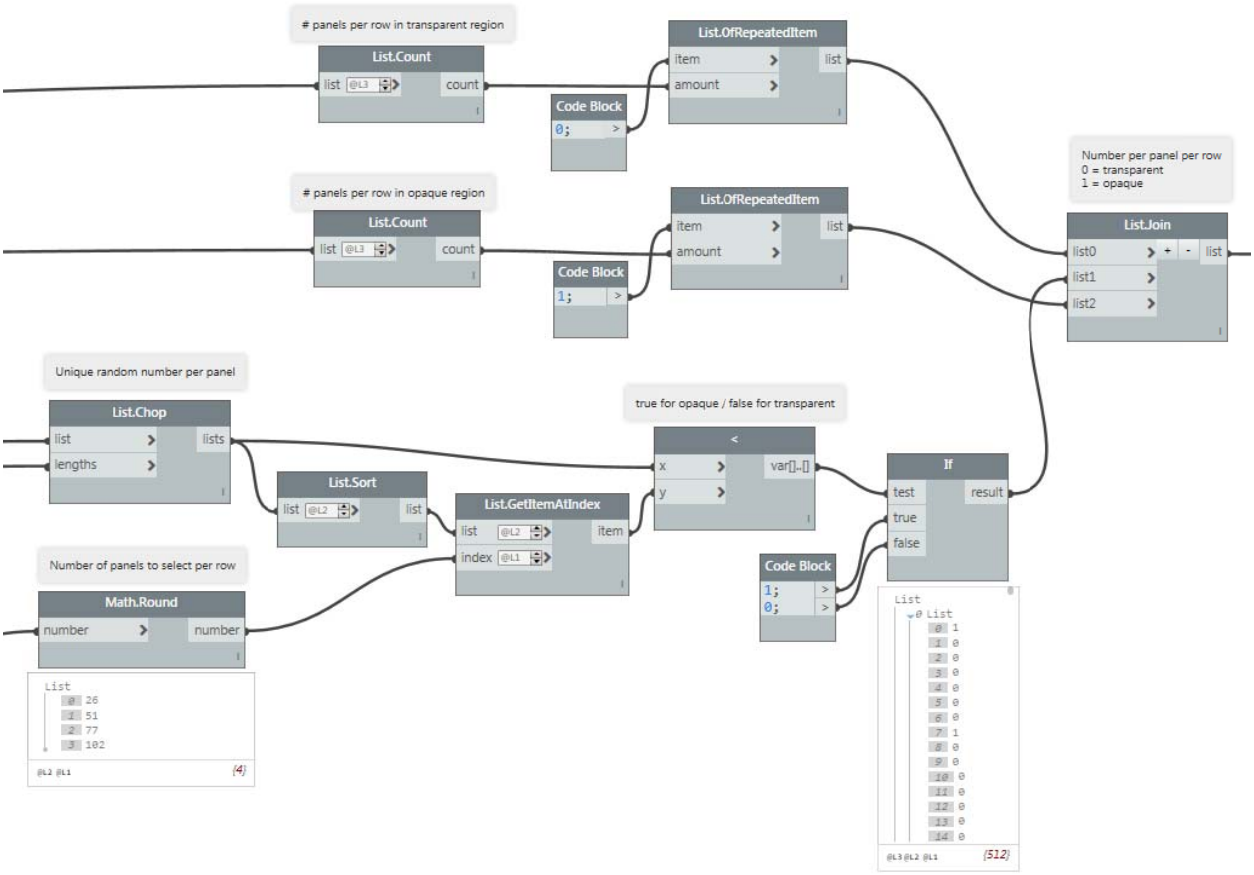


LOGIC FOR CONTROLLING THE % OPACITY (% SOLID PANELS VS. GLASS PANELS) PER ROW OF PANELS. THE % OPACITY IS INCREASED PER ROW IN THE TRANSITION ZONE IN A MANNER THAT CAN BE DESCRIBED BY THE SERIES SHOWN.

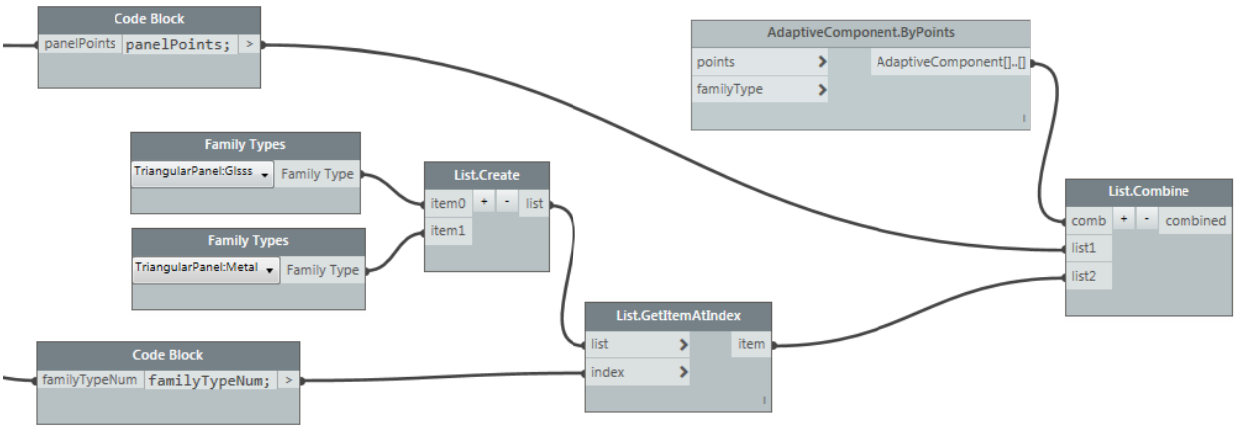
To create a gradient effect for the transition between rows of fully transparent panels to rows of fully opaque panels, select an increasing percentage of panels in the rows of the transition zone to be opaque. The series is described graphically above and with Dynamo below. Use this information to find the number of panels per row that should be opaque.



The random part comes in when we assign a unique random number for every panel. In the example, 26 panels from the first transition row should be opaque, so just pick the panels with the lowest 26 random numbers to be opaque.



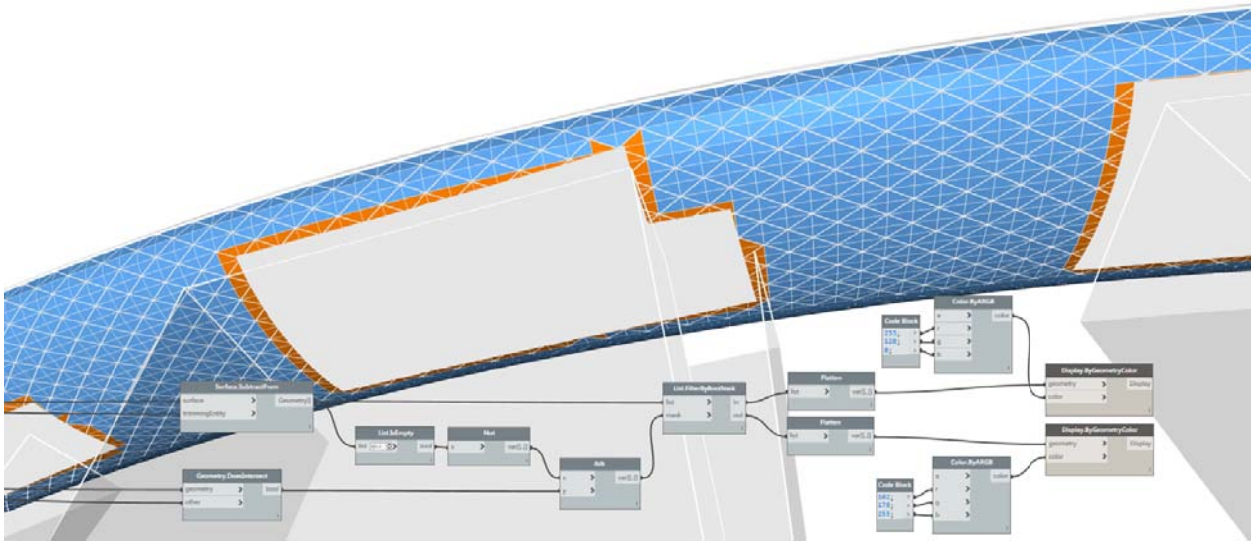
The general logic is to create a list of numbers, one per panel with the same data structure, that describes which family type to pick from a list. With just two family types, a simple true/false Boolean will suffice, but for more than two panel types, numbers will keep your logic clearer and more extensible. To add a third family type to the logic, simply add a third family type to the list and assign each panel a family type number of 0, 1, or 2.



BIM Enables Complexity

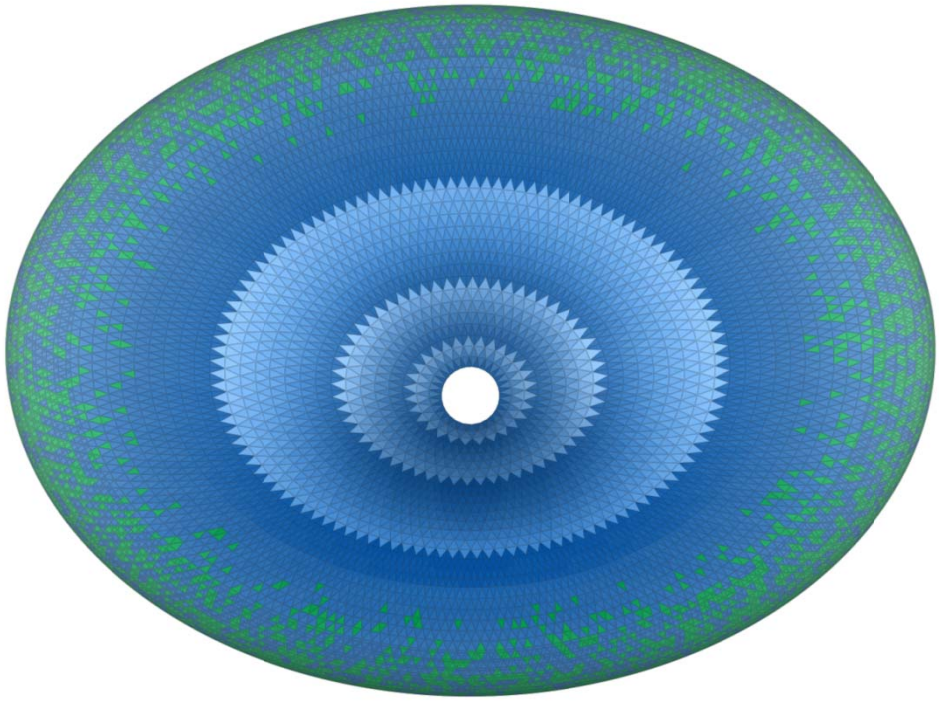
More Data!

Computational methods are often used in early design stages to explore, to find forms or to solve problems. In production stages, where documentation and fabrication are the goal, computational methods may also be used to understand a design with data. The following are some geometric analyses accomplished with comparatively little work once the script is written. For both Project Jewel and the Marina Bay Sands, the Architects developed many additional computational routines to extract and share information with the engineering consultants, client team, and contractors.

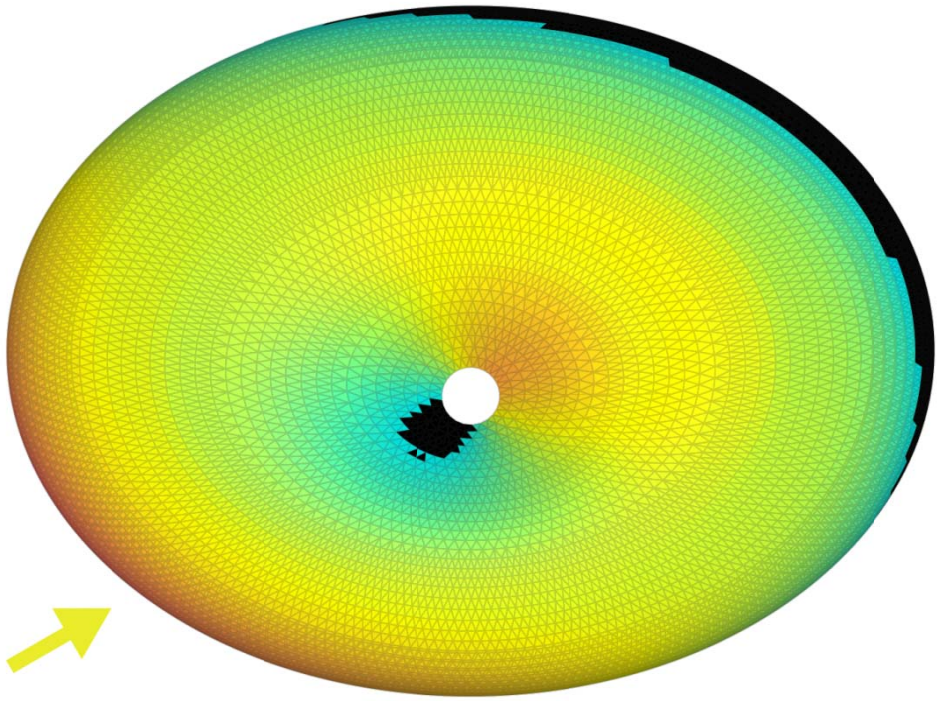


IDENTIFY ALL SPECIAL-CUT PANELS.

Any of these calculated quantities can be piped to the panel family instances in Revit to become proper BIM parameter values with a simple Element.SetParameterByName node.



SORT PANELS BY TYPE AND BY SIMILAR SIZES.



CALCULATE THE SOLAR EXPOSURE PER PANEL AT CRITICAL TIMES.

Managing the Project

Safdie Architects has designed and built many large and complex projects across the world, all of which are realized by large and complex teams of consultants, contractors, and clients. The role of the design architect is to protect the design intent, which requires that we clearly communicate both the design logic and the technical steps we took to achieve the design. It is important for us to find means and methods to optimize, analyze, and then transmit the information to all parties in as clear and common a language as possible. In addition to traditional design documentation such as drawings and models, we also share the computational work as "open source" with the teams. We provide a guide book detailing "recipes" that illustrate the workflow so as to bring all team members along, whether they are literate to the complex digital tools or not. Good stewardship of our designs begins with managing and communicating the data clearly.

Acknowledgements

This work is made possible by the talents and hard work of many individuals who contributed to the Marina Bay Sands and Jewel projects. Special acknowledgement for contributions to technical solution finding belongs to Charu Kokate, Director of the Singapore office of Safdie Architects; Damon Sidel, computational design architect; and designers and engineers at Buro Happold Engineering, notably Gustav Fagerström, technical designer.

SafdieArchitects

http://www.msafdie.com/