

Visual Dataflow Modelling : A Comparison of Three Systems

JANSSEN Patrick and CHEN Kian Wee
National University of Singapore, Singapore
patrick@janssen.name, chenkianwee@gmail.com

Abstract. Visual dataflow modelling (VDM) uses visual programming as a modelling technique for creating complex procedural design models. This paper compares three VDM systems, focusing on the cognitive stress associated with particular VDM constructs. The creation of iterative procedures is identified as a key area in VDM where cognitive stress tends to be high. Two iteration constructs are described, which are referred to as *list-based iteration* and *node-based iteration*. The research suggests that node-based iteration constructs have important advantages over list-based iteration constructs.

1. Introduction

Visual programming languages enable users to create computer programs by manipulating graphical elements rather than by entering text. When visual programming is applied to design, it results in a modelling approach that we refer to as *visual dataflow modelling* (VDM). Recently, VDM is becoming increasingly popular within the design community, as it can accelerate the iterative design process, thereby allowing larger numbers of design possibilities to be explored (for example, see [1, 2, 3]). A number of CAD systems now provide VDM interfaces, allowing designers to define form generating procedures without having to resort to scripting or programming [4, 5, 6]. These systems typically use the "*nodes and links*" based VDM approach.

A *node* can be thought of as a computational function that performs some action. The function in each node may require some input data, and may produce some output data. Nodes can therefore have *inputs* and *outputs*. The output from one node can then be wired to the input for another node, thereby creating a *link*. Links therefore represent the flow of data through the network. In some systems,

nodes have specific points on the edge of the node that represent the inputs and outputs. These points are referred to as *input ports* and *output ports*. Nodes also have certain inputs for which the user can manually enter values using the graphical user interface. These types of inputs are referred to as node *parameters*.

This paper will compare and contrast three VDM environments :

- McNeel Grasshopper is a plug-in for the 3d modeller McNeel Rhino3d (Rhino). For the following experiments, version 0.8 was used.
- Bentley GenerativeComponents (GC) was originally developed as a plugin for Bentley Microstation. However, recently GC was developed as a stand alone product, with Microstation embedded inside GC. For the following experiments, version V8i was used.
- Sidefx Houdini is a stand alone modelling and 3D animation package. Houdini includes advanced particle systems and dynamics. For the following experiments, version 11 was used.

1.1. Focus of comparison

All three systems use a "nodes and links" based VDM approach that is broadly similar. However, at a more detailed level, each system differs significantly in terms of how various VDM constructs are used. This research focuses on how the VDM constructs used in each of the three systems affect the difficulty of the modelling task. We refer to this as the *cognitive stress* associated with each construct. The aim of the research is to qualitatively compare and contrast the cognitive stress associated with VDM constructs from the perspective of a designer who is creating complex parametric models.

One important restriction that we have imposed is that we will assume that the designer only has basic scripting skills. Therefore – as far as possible – we will try and avoid using scripting in any of the three systems. However, completely avoiding scripting is not actually possible, especially in the case of GC, which relies heavily on scripting.

The three system also differ in many other respects, such as user interface, user documentation, speed of execution, and so forth. As far as possible, we will try and avoid these issues.

1.2. Description of systems

Each of the three systems uses different terminology to describe similar components. In order to ensure readability, we have decided to use consistent terminology for all three systems. For clarity, the original terminology is still highlighted in various places.

VISUAL DATAFLOW MODELLING : A COMPARISON OF THREE SYSTEMS

For each system, will refer to two distinct views : the *geometry view* where users can see the three-dimensional geometry, and the *network view* where the user can see the dataflow network. When describing networks, we will use the terms *nodes*, *links*, *inputs*, and *outputs*, as described above.

2. Three VDM systems

In order to introduce the three VDM systems, we will describe how each system was used to build a simple parametric surface. The modelling task in this case will consist of three steps :

- First, two BSpline rail curves will be drawn on the ground plane.
- Second, a BSpline section curve will be generated between the start points of the two rail curves. The peak point of this section curve will be generated by finding the mid-point between the two rail start points, and then translating this mid point up in the z-direction.
- Third, a BSpline surface will be generated by sweeping the section curve along the two rail curves.

The three systems will be discussed in reverse chronological order, starting with the newest system – Grasshopper, and ending with the oldest system – Houdini.

2.1. Grasshopper

The Grasshopper network for the parametric surface is shown in Figure 1. In Grasshopper, the user works directly in the network view. The Grasshopper environment presents the user with libraries of predefined nodes that can be dragged into the network view. Users can interactively create the wiring by linking the output of one node to the input of another node.

2.1.1. *Creating the lofted surface*

The Grasshopper environment communicates with the underlying Rhino modelling program. A user can draw geometric entities in Rhino, and then import these entities into the Grasshopper environment. In the example above, the two rail curves used to create a lofted surface were first drawn in Rhino and then imported into the Grasshopper environment in order to generate the surface.

In order to create the cross section curve, the start points of the two rail curves are first extracted using the End node. A Line node is then used to draw a line that joins two start points. The Eval node is then used to find the mid point, and this point is then translated up in the z direction using the Move node. The two start

points of the rail curves and the new translated point are then merged into a list using the Merge node. The cross section curve is then created from this list using the Curve node. Finally, the lofted surface is created from the two rails curves and the cross section curve using the Sweep2 node.

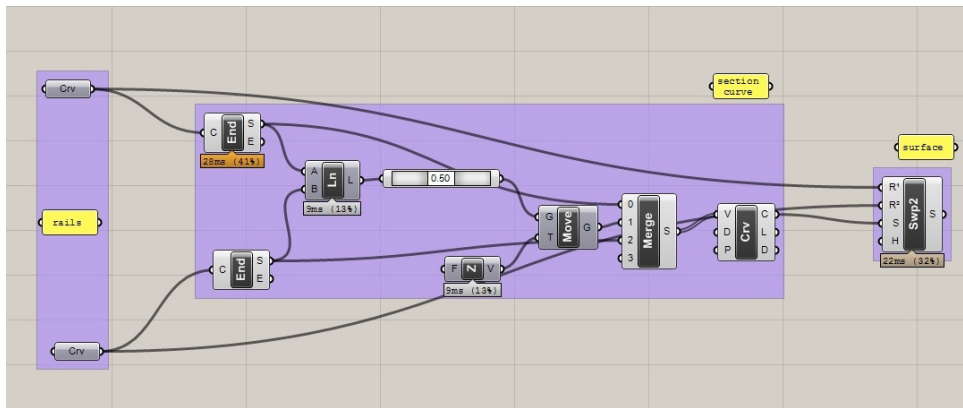


Fig. 1. A Grasshopper network for generating a parametric surface.

2.1.2. Grasshopper networks

Grasshopper provides nodes for creating geometric entities such as curves and surfaces. In addition, there are also many nodes for performing non-geometric low-level tasks, such as basic arithmetic and list manipulation.

Data for node inputs can be supplied in two ways. The user can either create a link or enter a value. Creating a link means that the user wires the output from some other node into the node's input. Entering a value means that the user simply selects the input and types in a value for that input. (In Grasshopper, no distinction is made between node inputs and node parameters) The latter method is only possible for inputs that require primitive data types.

The types of data that can flow through the links in a Grasshopper networks includes geometric entities and primitive data types (such as integer, floats, strings, booleans, and numeric ranges). The data is structured as *lists* and *trees*. Trees are similar to lists of lists (although the underlying representation is actually a dictionary data structure). Lists and trees will be discussed in more detail when iteration is discussed.

2.2. Generative Components

The GC network for the parametric surface is shown in Figure 2. In GC, the user can only interact with the network in very limited ways. The network view is used mainly as a visual aid, but the user cannot drag new nodes onto the network, or create new links from one node to another node. In addition, the nodes in GC do

VISUAL DATAFLOW MODELLING : A COMPARISON OF THREE SYSTEMS

not have any input ports. The way to create and connect nodes will be discussed in more detail below.

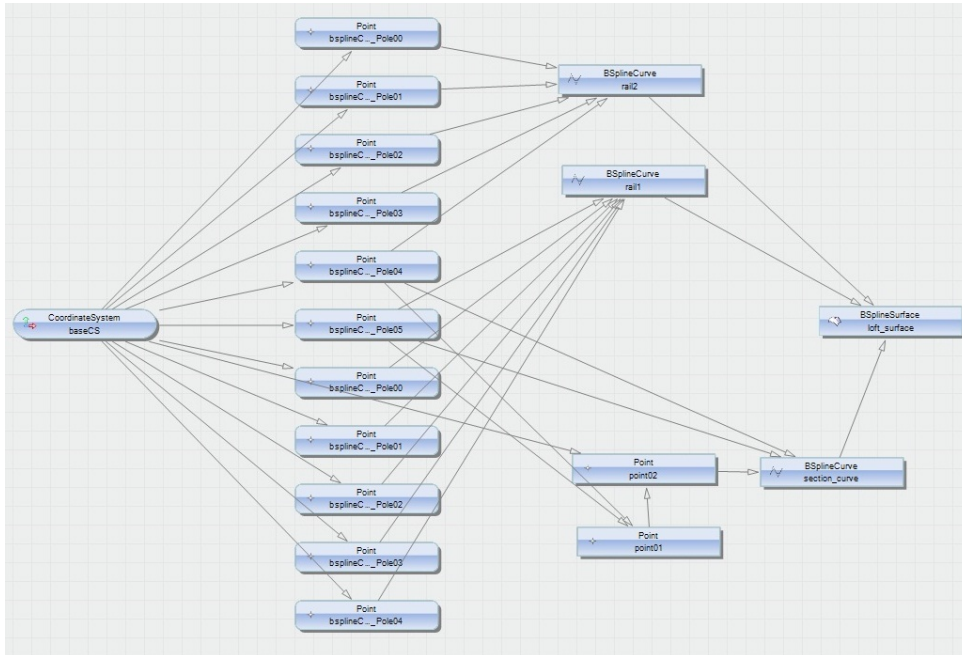


Fig. 2. A GC network for generating a parametric surface.

2.2.1. Creating the lofted surface

The GC environment communicates with the embedded Microstation modelling program in a way that is similar to Grasshopper/Rhino. A user can draw geometric entities in the Microstation environment, and then import these entities into the GC environment. This is referred to as *promoting* a geometric entity. However, when a geometric entity such as a curve is imported, the points that were used to construct the curve are all imported as well, each as an individual node. This results in many point nodes, as can be seen in Figure 2 above.

In the example above, the node on the left is the base coordinate system, which is a default node present in all GC networks. The two rail curves used to create a lofted surface are first drawn in Microstation and then imported into the GC environment. This results in 11 Point nodes and two BSplineCurve nodes.

In order to create the cross section curve, a third point needs to be created. A Point.CentroidOfSet node is used to find the mid point between the start points of the two rail curves. A Point.ByDirectionAndDistance-FromOrigin node is then used to translate this mid point up in the z direction. A BSplineCurve.ByPoles node is used to create the cross section curve. Finally, the lofted surface is created

from the two rails curves and the cross section curve using the `BsplineSurface.FromRailsAndSweptSections` node.

Note that scripted expressions were used in various places in this network. This is the reason why certain nodes are not required in GC. For example, consider the GC `BSplineCurve.ByPoles` node and the Grasshopper Curve node. These nodes do essentially the same thing. However, in Grasshopper, the input points have to be merged using a Merge node. In GC, this step is not required, since the merging of the points is done using a scripted expression.

2.2.2. GC networks

Most of the nodes in GC represent geometric entities. There are some non-geometric nodes, (such as the `GraphVariable` node which represents a user defined numeric value), but these are quite limited in number. The low level nodes found in Grasshopper do not exist in GC since such tasks are performed using scripted expressions.

Nodes in GC are categorised into types and sub-types. (In GC, the node type is referred to as a *Feature Type*, and the sub type is referred to as the *Update Method*). The node type typically defines the type of geometric entity that will be created. The sub-type then defines the method for creating this geometric entity. For example, for the Line node type, one of the sub-types is `ByPoints`. The dot notation is used to refer to the node type and sub-type, for example `Line.ByPoints`.

Nodes are created by first selecting a node type from a list, and then selecting a node sub-type from a sub-list. Once the sub-type is selected, the user is then presented with a number of inputs for which values need to be specified. (In GC, these inputs are referred to as *input properties*).

The user needs to enter values for all the inputs. If the inputs require other nodes, then the user can enter the names of nodes that already exist in the network. For example the `Line.ByPoints` node has two inputs : Start Point and End Point, both of which accept Point nodes. If the inputs require primitive types (such as integers, doubles, strings, or booleans) then the user can enter the actual value. (Thus, as with Grasshopper, no distinction is made between node inputs and node parameters.) The user can also enter a scripted expression that will return an appropriate value. These expressions can also retrieve values from other nodes.

Once all the input values have been entered, the user can then generate the node, and (assuming no errors were made), the node together with the appropriate links will then appear in the network view.

The fact that nodes represent geometric entities means that GC sometimes requires a reverse logic, as compared with Grasshopper and Houdini. When placing a node, the GC user must first decide on the end result that is desired (i.e. the node type), and then decide on the method for creating that end result (i.e. the

node sub-type). For example, in the parametric surface example, consider the node used to translate the mid-point up in the z direction. In GC, there is no node called Move or Translate. Since the end result of moving a point will be another point, the user must create a node by first selecting the Point type, and then selecting ByDirectionAndDistanceFromOrigin sub-type.

As with Grasshopper, the types of data that can flow through a GC network includes geometric entities and primitive types (such as integer, floats, strings, booleans). The data is structured as *lists* or *lists of lists* (which in GC are also referred to as *sets* or *arrays*). GC relies heavily on scripted expressions for manipulating this data.

2.3. Houdini

The Houdini network for the parametric surface is shown in Figure 3. In Houdini, the user can work either in the geometry view or in the network view. The Houdini environment presents the user with libraries of predefined nodes that can be dragged into the network view. As with Grasshopper, users can then interactively create links by connecting the output of one node to the input of another node.

Houdini has a number of different VDM type environments. For modelling, two VDM environments exist, organised hierarchically – called the Object network, and the Surface Operator (SOP) network. The Object network can contain one or more SOP networks. In this paper we will focus on SOP networks, since this is where most of the modelling is done.

2.3.1. *Creating the lofted surface*

All modelling in Houdini is performed using a VDM approach. Even if a user chooses to draw a curve interactively in the geometry view, nodes are still automatically generated in the network view. There is therefore no need to import geometry.

In the example below, the two rail curves used to create a lofted surface are drawn interactively in the geometry view. As a result, the two curve nodes are automatically generated in the network view. Note that Houdini does not create points as individual nodes as is the case with GC. Instead, the points are stored as parameters values within each Curve node.

In order to create the cross section curve, the data from the two rail curves are first merged using the Merge node. The Delete node is then used to delete all the geometry except the two start points of the rail curves. In order to find the mid point between these two points, a line is created using the Add node, and then the midpoint is found using the Carve node. This point is then translated in the z direction using the Transform node. The two start points of the rail curves and the new translated point are then merged into a list using the Merge node. A polygon

is created using the Add node, and this polygon is then converted into a BSpline cross-section curve using the Convert node. To create the lofted surface, the Rails node is first used to copy the cross section curve along the two rails. Finally, the surface is created by lofting the cross-section curves using the Skin node.

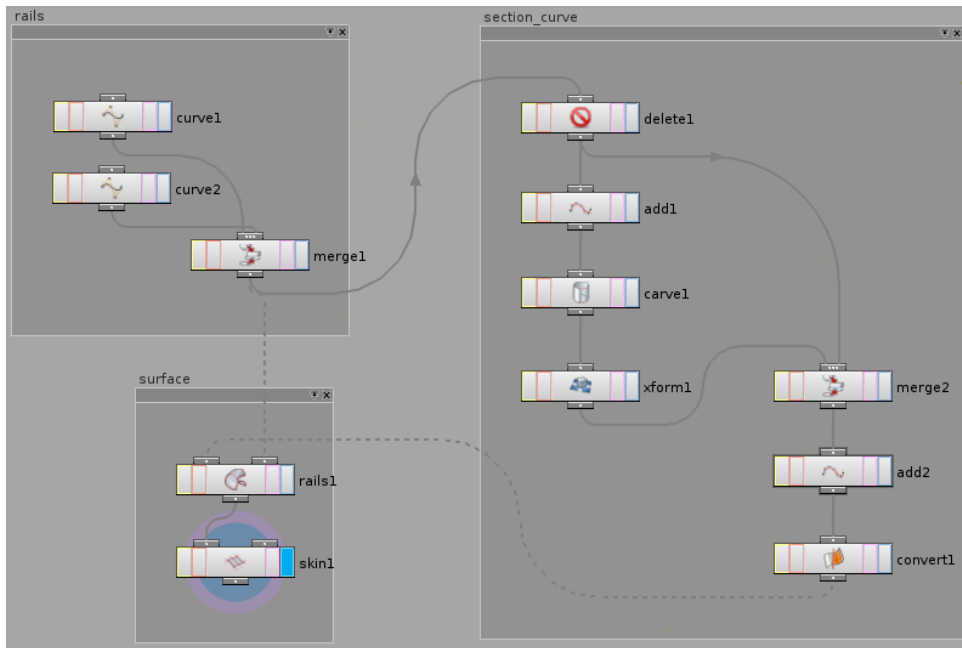


Fig. 3. A Houdini network for generating a parametric surface.

2.3.2. Houdini SOP networks

In Houdini, a distinction is made between node inputs and node parameters. Each node has one or more input ports where geometry can be fed in. Each node also has a set parameters that affect what the node does. For example, for the Curve nodes above, one of the parameters is the order of the NURBS curve to be created.

Users can enter parameter values through the graphical user interface. Alternatively, users can also enter scripted expressions that return an appropriate value. The expression can also retrieve a value from another node in the network.

In Houdini, the inter-node links created by scripted expressions are not visible in the network. A Houdini network therefore consists of two separate networks, which we will refer to as the *geometry network* and the *parameter network*. The geometry network is the visible network consisting of links connecting node outputs to node inputs. The parameter network is the invisible network, and is constructed from scripted expressions. In the example above, no scripted expressions were used, so there is no invisible parameter network.

VISUAL DATAFLOW MODELLING : A COMPARISON OF THREE SYSTEMS

The type of data that can flow through the network is different for the geometry network and the parameter network. For the geometry network, only geometric data can flow through the network links. Each piece of geometric data is associated with attributes that the user can interrogate and have access to. For example, a point will have attribute values for its x, y, and z coordinates. For the parameter network, only primitive data types (such as integer, floats, strings, and booleans) can flow through the (invisible) network links.

Note that the geometry network and the parameter network can be interlinked. For example, a parameter expression can be written that retrieves a float value from an attribute of geometric point in node somewhere else in the network.

3. A case study

In order to further explore and test the VDM systems, an exercise was conducted where the three systems were used to build a more complex parametric model. The VDM constructs used in the three systems were then compared in terms of cognitive stress.

3.1. The modelling task

The modelling task that was set for the exercise is shown in Figure 4. This task consists of four main steps :

- Step a1 : Starting with two NURBS rail curves, a third NURBS section curve is constructed.
- Step a2 : A NURBS surface is generated by sweeping the section curves along the rail curves.
- Step a3 : The surface is subdivided into a set of four sided polygons. These polygons have normal pointing outwards, and may be non-planar in some cases.
- Step a4 : Each polygon is replaced by a roof module constructed from planar polygons.

Steps 1 and 2 are actually the same as the surface modelling task described in section 2. Step 4 is a more complex step, and consists of four sub-steps. These sub-steps are applied to each polygon produced by step a3.

- Step b1 : The base polygon is reduced in size by scaling it around its centre point, so that a gap is created between each polygon.
- Step b2 : A top polygon is created by copying the base polygon, and then translating it in the normal direction by a translation distance t .

- Step b3 : The top polygon is reduced in size by scaling it around its centre point, by a scale factor of s . The top scaled polygon is then made planar by moving the corner points.
- Step b4 : A set of 13 new planar polygons are created from the corner points of the base and top polygons. In order to create these polygons, the mid-points of the four edges of the base polygon also need to be calculated.

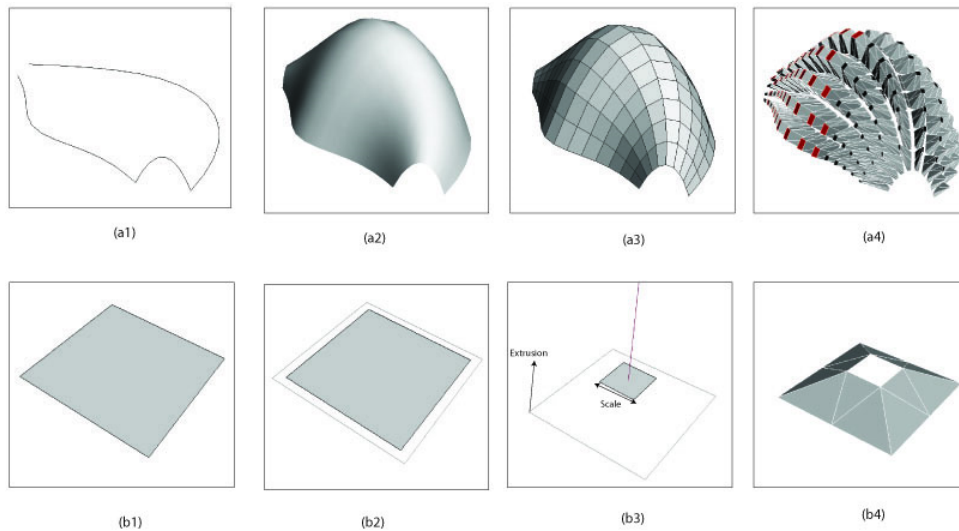


Fig. 4 : The steps that define the modelling task. Steps a1 to a4 are the main modelling steps. Step b1 to b4 define the process of creating a roof module in step a4.

In order to generate the roof modules for each polygon, certain parameters are required. Steps b2 has a translation parameter t , and step b3 has a scale factor parameter s . These two parameters are calculated in a way that means that their values will be different for each base polygon. This means that each polygon will be translated and scaled by a different amount, thereby further differentiating each roof module. These two parameters are calculated as follows :

- Translation parameter t is calculated based on the angle between the base polygon's normal vector, and a vector in the x direction (which we assume to be due north). The smaller this angle, the larger the translation distance will be. The range of 0 to 180 degrees is mapped to the range 0 to 1m.
- Scale factor parameter s is calculated based on the minimum distance between the centre point of the base polygon and a curved line (which we assume to represent a road). For distances greater than 20m, a scale factor of 0.8 is used, for distances between 10m and 20m, a scale factor of 0.5 is used, and for distances less than 10m, a scale factor of 0 is used.

3.2. Implementation

The three systems were all used to implement the modelling task described above, and in all three cases the task was completed successfully. Note that the modelling steps in some cases deviated slightly from those described above. However, the resultant logic of the parametric model was identical for all three systems. The approximate number of nodes used in each network was between 80 and 90 for Grasshopper, between 90 and 100 for GC, and between 70 and 80 for Houdini.

After having completed the modelling tasks, the processes of building the three networks were then qualitatively compared in order to try and identify areas of high cognitive stress.

In each modelling step, differences in cognitive stress were apparent, but the step that foregrounded these differences to the largest extent was step a4, and in particular sub-step b3. These steps required an iterative type of process, and each system differed significantly in how this process could be implemented.

3.2.1. Iteration in Grasshopper

In Grasshopper, there are no nodes that explicitly perform iteration. Instead, iteration is supported by nodes iterating over list and tree data structures. In Grasshopper this is referred to as *data matching*.

To understand data matching, consider the Eval node used in step a4 for finding the centre point of a surface. This node expects to receive two inputs, the surface to evaluate and the uv coordinates. If the node receives one surface and a (0.5,0.5) coordinate, then the node will generate one centre point. However, if the node receives a list of values on one or more of the inputs, then the node assumes that the user is trying to iterate in some way. So if a list of 50 surfaces are fed into the input, then a list of 50 centre points will be generated.

The data matching process can become very complex. For example, for the sub-step b3, this approach is used to scale the top polygons (which in Grasshopper are actually surfaces). First, the list of top polygons is used to produce a list of centre points, and this list is used to produce a list of distances from the road curve. The list of distances is then used to produce three lists of boolean flags indicating whether a particular scale factor should be used. Three lists of repeating scale factors are then created. The boolean lists values are used to cull the scale factors list. For example, if the boolean list is [[True], [False], [True], [False], ...], and the scale factor list is [[0.8], [0.8], [0.8], [0.8], ...], then the resulting culled list will be [[0.8], [], [0.8], [], ...]. (Note that the boolean lists, the scale factor lists, and the culled lists can all be represented as trees, thereby reducing the number of nodes that are required.) The three culled lists are then merged to create one final scale factor list. For example, if the two other scale factor lists were [[], [0.5], [], [], ...] and [[], [], [], [0], ...], then the resultant list

would be [[0.8], [0.5], [0.8], [0], ...]. The scale factor list is then used to scale the original list of polygons.

Throughout this process, the user must ensure that the order and structure of various lists in the network are consistent with one another. When working in Grasshopper, this is by far the area with the highest cognitive stress.

3.2.2. *Iteration in Generative Components*

In GC, iteration works in a similar way to Grasshopper, by iterating over list data structures. In GC this is referred to as *replication*.

When a node is created in GC, the user will be presented with the inputs for that node. Inputs labelled as *replicable* allow for the use of lists or lists of lists as inputs. When a list is used, the node will iterate through this list and replicate its output. This means that in GC, a similar approach could be used to the Grasshopper approach.

GC also allows users to create custom nodes (called *user defined features*), and this turns out to be very useful in limiting the cognitive stress of the task. Any existing network of nodes can be saved as a custom node. For the custom node, the user can set the inputs and outputs, and also set which inputs are replicable. Once created, a new node can be used in the same way as any of the built-in nodes.

For the implementation of step a4, two custom nodes are used, one nested inside the other. The first custom node is used to generate the whole roof module. The second custom node is used to generate the triangulated faces on each side of the roof module.

The inputs for the first custom node are one base polygon and one road curve. The outputs are the 13 new polygons. The advantage of this approach is that it allows the complexity of the roof module creation steps to be encapsulated inside a single node. Cognitively, this is very appealing, since the user can focus on a smaller problem first and solve that problem in isolation.

The process of generating the roof module occurs inside the first custom node. For step b3, the implementation is as follows. First, the top polygons is used to create a centre point. This point is then projected onto the road curve, resulting in a projected point. The centre point and the projected point are then used to create a projection line. The top polygon and the projection line are then used to create a normal line. In this case, a scripted node has to be used (where the node sub-type is *ByFunction*). The script extracts the length of the projection line, and then maps this length to one of the three scale factors using a set of *if* expressions. The top polygon and the normal line are then used to create the scaled polygon. In this case, a scripted expression is used to get the scale factor from the length of the normal line.

In GC, the use of custom nodes significantly reduced the cognitive stress related to list-based iteration. However, this actually results in a different type of

cognitive stress. Custom nodes exasperate the reverse logic nature of GC that was mentioned previously. For example, in this case, the user cannot just start working on the main network. Instead, the user needs to analyse the modelling task, break it down into a set of custom nodes (which in this case includes one custom node being embedded with another custom node), and then build each custom node in reverse order, starting with the deepest embedded node. We refer to this as a *reverse-order modelling* method, in contrast to the *forward-order modelling* method used by Grasshopper and Houdini. In general, forcing the user to adopt a reverse-order modelling method hinders open ended design exploration.

3.2.3. *Iteration in Houdini*

In Houdini, iteration can be achieved in two main ways. First, many nodes have the ability to iterate through the entities in the input geometry list. This is similar to the type of iteration found in Grasshopper and GC. However, in this case, the geometry list is always a one dimensional list (i.e. lists of lists are not possible, although entities in the lists can be grouped in other ways). For example, the PolyExtrude node will iterate through every polygon fed into its input and perform the action specific by the node parameters.

Second, Houdini also provides explicit iteration nodes, such as the ForEach node. This node is actually a container node, so other nodes can be placed inside this node. The nodes inside the ForEach node act like the body of the *for-each* loop in a textual programming. Typically, the ForEach node is used to iterate over geometric entities fed into its input. The ForEach node therefore combines both iteration and encapsulation within a single node.

For the implementation of step a4, two ForEach node are used, one nested inside the other. The first ForEach is used to model the whole module. The second ForEach is used to generate the triangulated faces on each side of the module.

The inputs for the first ForEach node are a list of base polygons and a road curve. The node will take the base polygons from the first input, and feed them – one at the time – into the network defined inside the ForEach node. (The road curve from the second input is not iterated over.) Each iteration of the ForEach node will output the 13 new polygons. The ForEach node will automatically merge together all the output polygons into one large list, which is the final output of the ForEach node.

The process of generating the module occurs inside the first ForEach node. For step b3, the implementation is as follows. First, the top polygon is used to create a centre point. This point is then used to measure the minimum distance to the road curve, and this distance is stored inside the point entity as an attribute. The same point is then fed into a node with a stepped ramp function. This function is used to map the distance to one of the three scale factors. The resulting scale factor is

again stored inside the point entity as an attribute. The top polygon is then scaled using the scale factor stored in the point. In this case a scripted expression is used to extract the the scale attribute from the point.

The two nested ForEach nodes used in Houdini match the two custom nodes used in GC. However, the overall modelling approach in Houdini is more similar to Grasshopper, and as a result Houdini does not suffer from the reverse-order modelling method inherent in GC. The user can model using the forward-order method, starting with the overall surface. For the roof module, the user can insert a ForEach node, and then go inside to work on one module in isolation. For the triangulated faces, the user can insert another ForEach node, and then go inside that node to work on one side of the roof module in isolation. At any stage, the user can also come back out and see what the result looks like when applied to the overall surface.

Also, note that as with GC, Houdini also allows users to create custom nodes (which in Houdini are called *Digital Assets*) and scripted nodes (which is just a special type of custom node). However, in this case, neither of these were required. In Houdini, custom nodes are mainly used to enable collaboration and sharing of nodes between multiple users. Scripted nodes are only really required when developing custom nodes with advanced features.

3.3. Discussion

For Grasshopper and GC, iteration is achieved by nodes iterating over list data structures. Such a data-driven list-based construct can lead to networks with a very high level of complexity. This can to a certain extent be overcome if a construct for encapsulation is also provided. In the case of Grasshopper, such a construct is currently not available, and as a result, Grasshopper networks often become very difficult to understand and debug. (It should be noted that the developers of the Grasshopper plug-in are currently working on some kind of custom node functionality. At the time of writing, how exactly this functionality would work was not yet clear).

For GC, an encapsulation construct is provided that allows users to build custom nodes. This construct can significantly reduce the complexity of the list based iteration construct. However, on the downside, the GC encapsulation construct forces users to use a reverse-order modelling method, where they start with the deepest embedded custom nodes.

In Houdini, the list-based iteration construct is only used with one dimensional lists, and as a result the cognitive stress remains low. As well as list-based iteration, Houdini also provides an alternative node-based construct that combines iteration and encapsulation within one node. With the Houdini ForEach node, the process of iteration is defined explicitly by the user, and as a result, it is more intuitive and easier to understand than when using a list based iteration construct.

The ForEach node is also more expressive than the alternative list-based constructs, and can be used to define many kinds of complex iteration. For example, consider the problem of iteratively subdividing a surface 20 times. On the first iteration, if the surface area is greater than a certain value x , then it is subdivided into four. Each of these sub-surfaces are then also put through the same process, so that any of the four sub-surfaces with an area greater than x will again be subdivided. The sub-sub-surfaces are again put through the same process, and so on. This kind of model can easily be created in Houdini with two nested ForEach loops. However, in Grasshopper and GC, this can only be achieved by explicitly duplicating the sub-division nodes 20 times. For GC, the situation is slightly improved since a custom node could be used, but this node would also have to be duplicated 20 times.

4. Conclusions and future work

The research has identified iteration as a key area in VDM where cognitive stress tends to be high. The comparison of the three systems suggests that node-based iteration constructs have four advantages over list-based iteration constructs. First, node-based constructs can combine iteration and encapsulation within one node. Second, node-based constructs can support both forward-order and reverse-order modelling methods. Third, node-based constructs are more easily understood, since the iterative process is represented explicitly. Fourth, node-based constructs are more expressive, since they can be used to define computational procedures of greater complexity.

The current research is based on a qualitative assessment of the three VDM systems applied to one modelling task. In order to better understand the cognitive stress associated with each iteration construct, more users need to be tested on a wider variety of modelling tasks. In addition, in order to further explore possible iteration constructs, a wider range of VDM systems also need to be studied.

5. Acknowledgements

We received useful help from people on various newsgroups. In the Grasshopper discussion group, Danny Boyes and Matt Gaydon contributed revised versions of the example network and were helpful in answering various questions. On the Houdini forum, C. P. Brown (cpb) contributed a revised version of the example network, replacing scripts with VOP networks. On the GC forum, Xun Zhou made helpful suggestions how the network could be improved.

References

1. Hesselgren, L., Charitou, R. & Dritsas, S. (2008). Architectural Structure – computational strategies. In Littlefield, D. (ed.). *Spacecraft : Developments in Architectural Computing*, RIBA Publishing, London, pp. 3-12.
2. El-Ali, J. (2008). The efficiently formed building, in Littlefield, D. (ed.). *Spacecraft : Developments in Architectural Computing*, RIBA Publishing, London, pp. 13-20.
3. Whitehead, H. & Peters, B. (2008). Geometry, form and complexity. In Littlefield, D. (ed.). *Spacecraft : Developments in Architectural Computing*, RIBA Publishing, London, pp. 21-34.
4. Aish, R. (2005). Introduction to GenerativeComponents, A parametric and associative design system for architecture, building engineering and digital fabrication. White paper on <http://www.bentley.com>, accessed 2005.
5. Aish, R. & Woodbury, R. (2005). Multi-Level Interaction in Parametric Design. In *Lecture Notes in Computer Science*, 2005, Vol. 3638/2005, pp. 151-162.
6. Woodbury, R. (2010). *Elements of Parametric Design*, Routledge, NY.