Daniel Davis, Jane Burry, Mark Burry
dmmd123@gmail.com, jane.burry@rmit.edu.au, mark.burry@rmit.edu.au

Yeti: Designing geometric tools with interactive programming

Abstract
Designers scripting geometric tools have had two options: either use an interactive visual script, or forgo interactivity to use a text-based script. Within this paper we consider a third option: interactively writing text-based scripts. Described is an interactive scripting environment created for this purpose, which manages geometry with a Directed Acyclic Graph generated from the text-based relational markup language, YAML. The environment is compared to the two existing scripting options by using them to draw three geometric compositions. We argue it is possible to interactively script geometric tools, and that interactivity is a vital component in making scripting intuitive.

Keywords
Interactive programming; End-user programming; Design computation; Parametric modelling.

1 Introduction
Since Sutherland's digital Sketchpad, designers have aspired to make coding more like sketching. Prior to Sutherland, computer programs were manually executed in 'batches'. The designer would compile the code, define inputs and parameters, run the program and wait – often a long time – for the result. When Sutherland developed Sketchpad in 1963 he sought to overcome the delays in batch processing and allow "man and a computer to converse rapidly." [1]. Sketchpad was one of the first interactive computer programs. It displayed the results of the designer's actions immediately, which facilitated feedback and reflection in a conversation between designer and computer. Almost 50 years after Sketchpad, interactive graphics programs have all but replaced the drawing boards they once imitated. These design programs each offer a prescribed palette of design tools and often afford designers the ability to script their own customised tools. A script defines a list of operations for the computer to carry out. Yet when designers attempt to design their own design tools with scripts, they must once again design using a batch-processed system. This is because scripting in its current form involves writing out a procedural script, pressing compile, setting the inputs for the script, and waiting for the computer to draw the result – like the programs prior to Sutherland. Unlike sketching, or even digital drafting, with scripting there is a pronounced delay between the action of the user (changing the code) and the reaction of the system (redrawing the geometry). Such a delay can slow the pace of iteration in the design process and hinder feedback reaching the designer in a timely manner. Recently a number of scripting languages have emerged for musicians that enable the interactive modification of text-based scripts. Using these interactive programming languages the musician can immediately hear how changes to a script driving a musical performance will sound. These languages appear to be a viable method

for achieving a similar level of interactivity in geometric design. However, as will be outlined in this paper, the emphasis musicians place on tempo and timing makes their techniques unsuitable for the computationally intensive task of generating geometry.

With no existing interactive scripting tools for describing geometry, this research seeks to better understand the technical and cultural limits of designing geometry with interactive scripts. This paper begins with an outline of an interactive scripting technique that overcomes some of the computational impediments associated with the interactive scripting of geometric tools. 'Yeti' is an interactive scripting environment developed to utilise this technique, the implementation details of which are explained in this paper. Using a reflective practice methodology, Yeti is tested in a pilot study by applying it to three geometric design problems and comparing its performance to that of non-interactive text-based scripts and interactive visual scripts. The three design problems are taken from an architectural context, although it is anticipated this research will be of interest to designers outside the field of architecture, particularly those describing geometry with their own scripted design tools or parametric models. The paper begins by describing some of the existing scripting environments designers utilise.

2 Existing Design Scripting Methods

A script defines a list of actions for the computer to carry out. In contemporary usage, scripting is essentially synonymous with programming. As such a script can automate tasks that would otherwise be performed through the user interface and it can also define entirely new actions. For designers the primary motivations to script are: productivity (doing tasks that would take too long otherwise), and control (linking various actions together to create customised tools) [2]. Design software packages often encourage scripting through inbuilt scripting interfaces, and there are also applications (like Processing) that are created explicitly as standalone scripting interfaces for designers. With design increasingly being conducted on computers, so too scripting has increasingly become a way for designers to automate and control the design process.

To run a script, the computer generally has to interpret (or compile) the script into a machine-readable set of instructions. This is supported in scripting interfaces through an 'Edit-Interpret-Run' loop, whereby the designer edits the text of the script, presses a button to activate the script, and waits first as the computer validates the script, then waits as the computer interprets the script into a machine-readable set of instructions and finally waits as the computer runs this set of instructions. The notable exception is some visual programming languages, like Max-MSP, which will be discussed in the subsequent section. As a consequence of the Edit-Interpret-Run loop, there is a pronounced delay between the action of the user (editing the script) and the reaction of the system (redrawing the geometry). This delay impacts the rate of iteration since each variation of the script the designer tests goes through the Edit-Interpret-Run loop, often with the designer manually deleting the geometry of the previous loop between iterations.

3 Interactive Scripting

Interactive programming (also known as live-programming) is a method for editing and interpreting scripts while they run. To the end user there appears to be no Edit-Interpret-Run loop because any edit they make is automatically incorporated with the already running instance of the script. Behind the scenes there is still an Edit-Interpret-Run loop, where the computer automatically interprets an edit and in real-time invisibly transitions the running instance of the script to the new edited version. The net effect is that the end-user can experience in real-time the consequences of editing a script – closing the gap between action and reaction.

The crux of creating an interactive programming environment is smoothly transitioning a running script between different versions of the script. The most obvious method is to abandon the currently running script whenever it is edited, and automatically interpret and run the updated version of the script. For certain applications, such as SimpleLiveCoding for Processing [3], this is effective. However for the computationally taxing task of drawing geometry this is less desirable since it involves abandoning all the previous calculations and recalculating the geometry every-time the script is modified, even if the modification only changed a small and discrete part of the geometry. The method typically employed in interactive debugging is to maintain the state of the code – through a call stack – allowing the code to be rewound to the site of the edit [4], however all subsequent code still needs to be recalculated, even if it is not affected by the modification.

Perhaps one of the most developed systems for transitioning scripts has been developed by musicians, for whom interactive programming allows modification of scripts driving a musical composition while immediately experiencing the sonic implications.
The first performance with an interactive script was by Ron Kuivila at STEIM in 1985 [5]. In 2000, Supercollider led a revival of text-based interactive programming for musicians, and was followed by a number of similar languages like ClanK and Impromptu. All of these languages share in common the need from musicians to invoke actions relative to an underlying time signature. Typically this occurs through scheduling a reoccurring sequence of actions to be performed, and whenever the script is modified, adding the modified actions to the queue [6]. These musical environments have been adapted to generate geometry but the repetitive cycling of actions makes it unsuitable for generating anything other than basic geometry [7].
Therefore despite the scattered implementations of interactive programming, few – if any – are suitable for the unique challenges designers face when shaping geometry with scripts. Designers desiring the interactive feedback of sketching while they script currently have to make do with Edit-Interpret-Run loops. This is primarily due to the difficulty of editing and updating an already running script while handling the computational intensity of geometric calculations.

4 Introducing Yeti

The problem of editing a script while it runs geometric calculations has been elegantly overcome by the interactive visual scripting environments GrasshopperTM, HoudiniTM and Generative ComponentsTM. These three environments use Directed Acyclic Graphs (DAG) to represent relationships between geometry [8].
A relationship may be that a line is tangent to a circle (the circle is a parent of the line) and whenever the circle is adjusted, the line moves to satisfy the tangential relationship. From these geometric relationships the dependencies of the geometry can be extracted. Thus when part of the DAG is edited, the only recalculation required is to the geometry dependent (and therefore affected) by the edited part of the DAG [8]. Since a node within the DAG is directly associated with the geometry it creates, the node can manage the creation and deletion of geometry without the designer needing to remove old instances of the geometry. The DAG is defined through visual interfaces in Grasshopper, Houdini and Generative Components. Text-based scripts within these environments cannot not be interactively edited and still use the Edit-Interpret-Run loop.
Yeti is a text-based interactive scripting environment developed for the interactive creation of geometric tools. Yeti uses a DAG to manage the editing and calculation of geometry, but the DAG is defined through a text-based script rather than the visual interfaces used by Grasshopper, Houdini and Generative Components.
The language of Yeti's script is based on the relational mark-up language YAML [9]. The syntax consists of 'key: value' pairs, where the key is assigned the value to the right (the 'x:' in Table 1, has a value of −10). More complex values can be assigned through a list of 'key: value' pairs, separated from the parent key with indentation (the 'point:' in Table 1, has been assigned 'key: value' pairs for x, y & z). Relationships are defined by naming keys (names start with the '&') and referencing them as a value (references start with the '*').

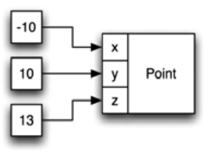
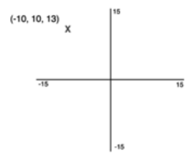
Yeti (YAML)	Directed Acyclic Graph	Geometry
point: ››x: −10 ››y: 10 ››z: 13		

Table 1. A simple Yeti script in YAML (left) and the corresponding DAG (centre) with the geometry it produces (right). Note all keys in the Yeti script map directly to nodes in the DAG.

The advantage of using YAML is that the 'key: value' pairs map directly into a Directed Acyclic Graph, where the key defines a node and the value defines either: the property of the node, or its relationship to other nodes (see Table 1). Whenever a script is modified in Yeti, the underlying DAG is automatically updated in the following process:

1. The edited script is tokenised into keys and values.
2. For every key, a corresponding node is generated in the DAG.
3. The node is assigned properties and related to other nodes based on the values assigned to the corresponding key.

4. Once the DAG is created, all nodes dependent upon deleted, added or modified nodes are recalculated, creating a new instance of the geometry.

In addition to interactive editing of running scripts, the YAML language and underlying DAG enable a number of unique features in Yeti:

• *Error handling*: The interpretation of code while it is being written frequently causes errors because the computer is often unable to resolve the ambiguity of partly written code. Yeti interprets and run scripts with errors by ignoring ‘key: value’ pairs with errors in them. Typically errors cannot be ignored in other languages because it interrupts the top-to-bottom progression of logic.
• *Interactive debugging*: Clicking a key in the code highlights the geometry controlled by the key. This helps clarify the often-enigmatic connection between code and geometry that characterises scripting. Yeti is able to do this because keys in the script are directly associated with parts of the model’s geometry via nodes in the DAG.
• *Instancing of objects:* The YAML language can be extended to include new keys. The user does this by creating a prototype object for the key through a list of ‘key: value’ pairs. When the new key is used in the script, a new instance of the prototype object is created and modified for the unique properties of that object instance. This is a common feature in text-based scripts but one that visual scripts often do not support.

Python	Yeti (YAML)
P1 = Rhino.Geometry.Point3d(0,10,13)	line: &myLine
P2 = Rhino.Geometry.Point3d(P1.X, P1.Y+20,0)	start: &p1
myLine = Rhino.Geometry.Line(P1,p2)	x: 0
doc = Rhino.RhinoDoc.ActiveDoc	y: 10
doc.Objects.AddLine(myLine)	z: 13
	end: &p2
	x: *p1.x
	y: (*p1.y + 20)

Table 2. Comparison of scripts to draw the same constrained line in Rhino Python and Yeti.

YAML also has its drawbacks. The definition of geometric properties and relationships in YAML is a significantly different method of scripting compared to the ordered list of procedural actions familiar to many scripters (the two paradigms are compared in Table 2). Similarly the recursion offered in procedural languages is not yet possible in Yeti due to the difficulty of representing recursion in a DAG. For this reason Yeti is not Turing-complete, and therefore unsuitable for certain operations like L-systems and cellular automata. Despite these quirks and limitations, YAML and the underlying DAG is fundamental to empowering the interactive scripting of geometric calculations, along with a number of other advantages like interactive debugging and error handling.

5 Designing Geometry with Interactive Scripts

5.1 Method

To explore the viability of interactively generating geometric tools with text-based scripts, we carried out three design projects with the iterative scripting environment Yeti (version 0.3). As a benchmark we repeated the work with two established methods of scripting: interactive visual scripting in Grasshopper (version 0.8.0052), and text-based scripting with Rhino Python (In Rhino5, version 2011-11-08). The three design problems have an architectural bias but the focus of the analysis is towards the shaping of geometry and designers who do so already through scripting or parametric modelling. Since this is the first time interactive scripting has applied in this context, the investigation is a pilot study designed to identify the major issues with interactive scripting in anticipation of refining Yeti further. The three projects used in the study are:

Project 1 & 2: Axel Kilian’s Roofs

Axel Kilian developed a pair of tutorials in 2005 to teach the then highly experiential visual scripting software, Generative Components. The tutorials demonstrate how to developed a customised geometric tool with scripting and introduce “several key parametric modelling concepts,” such as: geometric constraints, data arrays, modularity, and aggregate difference from topological similarity [10]. These two roofs form an interesting benchmark, both because they employ essential scripting techniques, and because they hold some historic credence with which it is possible to track the development of parametric modelling.

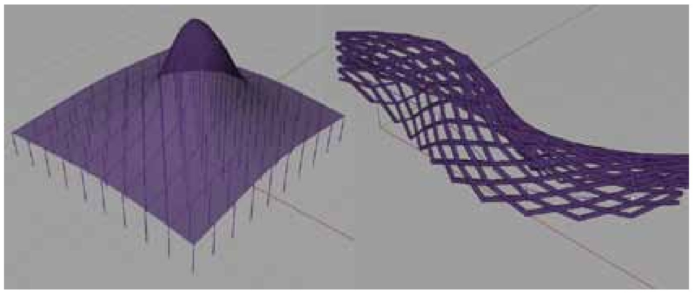


Fig. 2. Project one in Yeti (left) the more complicated project two in Yeti (right).

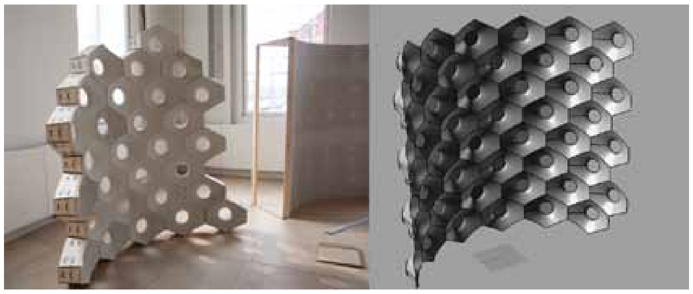


Fig. 3. Full-scale prototype at Smart Geometry (left) and associated digital model (right).

Project 3: Smart Geometry 2011
As part of the Responsive Acoustic Surfaces workshop at Smart Geometry 2011, two acoustic walls were developed to test the sound scattering of various plaster hyperboloid tile configurations [11]. Originally the wall was designed with the interactive visual scripting environment Grasshopper, used alongside Digital Project and Open Cascade. From the workshop it is known the project pushes the limits of interactive design through the computationally expensive calculation of the hyperboloid intersections, where very subtle nuances in the planarity of the intersections determine the project's viability.

5.2 Differences between scripting environments

The following section broadly describes the main differences between the three scripting environments, with a focus on the technical capacity of each environment.
Geometric output. The geometric library for Yeti is still being developed but it was capable of creating the geometry of the Kilian Roofs and creating the geometry of the hyperboloid wall, as was Grasshopper and Python. In all the environments the most challenging geometric task was to encode the reasoning for which side of the hyperboloid intersection to keep in project three. The difficulty of expressing this indicates that certain types of architecture are more amenable than others to the logic of scripting, a logic Yeti follows.
Script length. The number of lines of code in the Yeti scripts were essentially identical to the Python scripts, although the lines of the Yeti scripts tended to be sparser containing just ten characters on average, whereas the lines in Python contained 25 characters on average. The Grasshopper schemas are not directly comparable to text-based scripts, but it should be noted that the interface for Grasshopper did many of the tasks that needed to be explicitly defined in the Python and Yeti scripts, such as making geometry visible. In the Python scripts, significantly more of the script was involved with managing arrays of data, but in Grasshopper and Yeti arrays of data were resolved by the software rather than the user [12].
Speed of execution. Yeti remained responsive throughout the two roof projects. On the more complex roof an update cycle typically took 100ms, which was faster than one can type. This is comparable to Grasshopper and faster than the Python script, which took 2 seconds to execute (Python's biggest hindrance seems to be the way it draws geometry). The intersections in the geometry of the hyperboloid wall were too complex to calculate in real-time with either Grasshopper or Yeti. It was only possible to complete the project by disabling the interactivity and reverting back to the manual Edit-

Interpret-Run loop employed by scripts like Python. While interactive programming is useful on simple projects, batch-processing is still a useful paradigm to grind out computationally expensive geometry and a useful paradigm for Yeti to fall back on.

5.3 Discussion: Intuition and Interactivity

In creating Sketchpad, Sutherland not only created the first interactive CAD tool but also one of the first programs to "eliminate typed statements (except for legends) in favor of line drawings" [1]. Sutherland described controlling a computer with text as "writing letters to rather than conferring with our computers" [1]. It is an argument about whether designers find interactive drawing more intuitive than writing code. In the past 50 years, despite the increasing prevalence of scripting, overwhelmingly CAD software consists of interactive visual interfaces activated with mouse and keyboard shortcuts.

However for certain types of geometry, like the geometry in the three projects above, scripting is the only method of productively generating and controlling the geometry. For these projects designers have no option but to 'write letters' to the computer sent via the Edit-Interpret-Run loop. In writing these letters, some of the intuitiveness is bound to the language it is written in. The scripts from Python and Yeti, while of a similar length, are strikingly different in approach (See Table 2) the Python scripts methodically working through a list of actions while the Yeti scripts begin with the outcome and describe the necessary parameters. For this reason Yeti may seem unintuitive to designers already conversant with procedural scripting languages like Python [12]. Whether users new to scripting experience this difference in intuition remains to be studied.

Another factor in the intuitiveness of letters written to the computer is the speed they are returned. In carrying out the three projects above, it is clear intuition and interactivity are tightly coupled. Being able to click on words in the Yeti script and see the geometry they control highlighted, helps clarify their purpose. Similarly being able to edit a parameter and instantly see the geometry respond, makes manipulating the parameters feel more intuitive.

While writing code often feels like 'writing letters', the three projects above begin to uncover how interactivity can make scripting a more conversant and therefore intuitive experience. It remains to be seen if the advantages interactivity brings are enough to overcome the hindrances of needing to use a language like YAML. The cultural implications of such a change could be profound, particularly if scripting became intuitive enough to use on projects other than those that can only be productively generated and controlled with scripts.

6 Conclusion

Sutherland's digital Sketchpad placed interactivity at the foundation of digital design. When scripting designers have had two options: either use an interactive visual script, or forgo interactivity in favour of writing the script with text. This paper has articulated a third option: writing a text-based script in an interactive programming environment. Significantly this research has demonstrated it is possible to interactively program computationally-intensive geometric tools. This can be achieved by managing the geometry with a Directed Acyclic Graph, which can be generated from a text-based relational markup language like YAML. The markup language used to attain the performance necessary for interactive scripting may seem unusual compared to the conventions of established methods of design scripting, however there is a real benefit to being able to instantly see how a change to the script will affect the model's geometry. In the future interactive programming may make the act of writing code as responsive for designers as the act of sketching in a Sketchpad.

Acknowledgments

This research was funded as part of the Australian Research Council Discovery Grant 'Challenging the inflexibility of the flexible digital model' lead by Mark Burry, John Frazer and Jane Burry.

References

[1] Sutherland, I. (1963). Sketchpad: A Man-machine Graphical Communication System. PhD Thesis, Massachusetts Institute of Technology.

[2] Burry, M. (2011). Scripting Cultures. West Sussex: John Wiley & Sons Ltd.

[3] Simple Live Coding. Retrieved November 19, 2011 from https://github.com/fjenett/simplelivecoding.

[4] Johansson, O. (2011). Describing Live Programming Using Program Transformations and a Callstack Explicit Interpreter. Masters Thesis, Linkoping University.

[5] Roads, C. (1986). The second STEIM symposium on interactive composition in live electronic music. Computer Music Journal, 10(2),.

[6] Wang, G., & Cook, P. (2004). On-the-fly programming: Using code as an expressive musical instrument. In Proceedings of the 2004 International Conference on New Interfaces for Musical Expression (NIME). Hamamatsu: Shizuoka University.
[7] Sorensen, A. (2005). Impromptu : An interactive programming environment for composition and performance. In Paper presented to the Australasian Computer Music Conference 2005. Brisbane: ACMA.
[8] Woodbury, R. (2010). Elements of Parametric Design. Oxon: Routledge.
[9] Ben-Kiki, O., Evans, C., & Net, D. (2009). YAML Ain't Markup Language (YAML™) Version 1.2. 3rd. http://www.yaml.org/spec/1.2/spec.html
[10] Woodbury, R., Aish, R., & Kilian, A. (2007). Some patterns for parametric modeling. In Proceedings of the 27th Annual Conference of the Association for Computer Aided Design in Architecture. Halifax, Nova Scotia.
[11] Burry, J., Davis, D., Peters, B., Ayres, P., Klein, J., Pena de Leon, A., et al. (2011). Modeling hyperboloid sound scattering: The challenge of simulating, fabricating and measuring. Proceedings of Design Modeling Symposium Berlin. Berlin: Springer Verlag.
[12] Janssen, P., & Chen, K. (2011). Visual Dataflow Modeling: A Comparison of Three Systems. Proceedings of CAAD Futures 2011. Liège: Les Éditions de l'Université de Liège.

Daniel Davis,
Jane Burry,
Mark Burry
Spatial Information
Architecture
Laboratory, RMIT,
Australia