R. Stouffs, P. Janssen, S. Roudavski, B. Tunçer (eds.), *Open Systems: Proceedings of the 18th International Conference on Computer-Aided Architectural Design Research in Asia (CAADRIA 2013)*, 499–508.

DECODES

A platform-independent computational geometry environment

Kyle STEINFELD
University of California Berkeley, Berkeley, CA, United States
ksteinfe@berkeley.edu

and

Joy KO
Rhode Island School of Design, Providence, RI, United States
jko01@risd.edu

Abstract. This paper presents three strategies – host-independence, domain-specificity, and context-appropriate abstraction – for the design of a textual programming environment supporting computational architectural design that more effectively addresses the needs implied by common practices within this community. A survey of existing computational design environments is first presented through the lens of these three strategies. An outline is then presented for a platform-independent computational geometry library built upon each of these strategies, alongside a report of progress made on implementing this platform thus far. More information on this project may be found at http://decod.es/

Keywords. Computational geometry; programming environment; scripting; interoperability.

1. Introduction

In order to design a better programming environment for the architectural design community, it is prudent to first pose a few fundamental questions. What are the computational design environments currently employed in this community? Which ones are useful? Which present obstacles to the needs of common architectural situations?

Architectural design is not dominated by a single unified software platform, and, in stark contrast to other parts of the AEC industry, is characterized by a diversity of competing models, formats, and descriptions of the built environment.

This diversity not only reflects the architectural designers' predilection for moving between a variety of media and software platforms, but also of the naturally collaborative position of the architect who must regularly coordinate across disciplines and among scales. A host of geometric and algorithmic constructs have indeed been created to support these needs, some tailored to specific domains of practice – e.g. recursive tiling definitions and particle-spring systems – while others are more generalizable – e.g. NURBS surfaces and subdivision meshes.

To effectively support the full diversity of representations and operations required by architectural design practice, a computational design environment must meet the following, often competing, requirements:

- Provide access to a rich set of *primitive elements*: domain-specific geometric and algorithmic constructs that make up the world that defines the design.
- Provide appropriate *code structures*: conventions and built-in abstraction methods that allow the user to easily manipulate, modify and extend the set of primitive elements.

1.1. PRIMITIVE ELEMENTS

Most computational designers regularly employ scripts that operate within a host CAD environment, and rely upon the primitive elements offered by this environment. Access to the set of primitive objects and commands within CAD environments is provided via an application programming interface (API). Examples are AutoCAD's "object model" and Rhino's "OpenNURBS" and "RhinoCommon". Most APIs provide access to an interchangeable set of basic primitives such as points, vectors, lines, and meshes, as well as a more specialized set of complex geometric entities like surfaces, solids, or procedurally-based geometry such as subdivision surfaces. More sophisticated CAD environments may address non-geometric primitives, including relationships and constraints governing the parametric definition of geometry, and analytical routines that provide numeric analysis.

The advantages of scripting within such a host environment is demonstrated by the diversity of designs possible through this approach. There are, however, associated costs that may be easily overlooked. For example, there are significant barriers to modifying routines or developing new primitives when adopting an existing CAD library. Primitive elements in such libraries are often bloated with extraneous features and complexity. In RhinoCommon, for instance, there are 13 distinct constructs (RhinoCommon Software Development Kit, 2013) for describing a point object, each one deployed only in particular situations. In addition, more specialized constructs are usually less generalizable, which increases a user's reliance on a specific host environment (Leitão et al., 2012) and reduces the flexibility to operate between multiple host environments and deploy a diverse set of descriptions of the built environment.

Only a small minority of computational design activity takes place outside of an existing CAD environment. By necessity, designs operating independently of an existing CAD library must define the primitive elements from which design iterations are assembled from relatively low-level elements. For example, Kilian and Ochsendorf (2005) have constructed a sophisticated particle-spring system for structural form-finding based only on a rudimentary physics solver. It should be noted that in cases such as this, the libraries developed are intended for a targeted purpose, and are not applicable to computational design problems at large.

1.2. CODE STRUCTURES

All programming environments, whether textual or visual, present a set of code structures that facilitate the manipulation, combination, and modification of their set of primitive elements. Computational design in architecture is carried out by scripting using a variety of programming languages. Thus, it is useful to compare the code structures that make up these languages and the level of domain-specific functionality and abstraction mechanisms that these structures allow.

For textual languages, a number of foundational concepts in the general categories of object representation and manipulation (ex: variables), control flow (ex: logic gates, loops), and code organization and modularization (ex: functions, modules, classes) are needed before one can start creating or manipulating primitive elements. Generally, textual languages have a low level of domain specific functionality in contrast to the high level provided by many visual programming languages. For example, in the Grasshopper visual environment, structures such as m-d sliders, range components, built-in mapping functions for collections of objects, and "wires" that allow for the visual control of data – all provide access to complex geometric constructs in a way that resonates with visual designers, without detailed knowledge of the inner workings of the software. Since the textual languages widely used for computational design today – e.g. Rhinoscript, AutoLISP, and MEL – were developed before visual programming environments gained prominence, they typically make poor companions for visual programming (Celani, 2012). They neither integrate well with their visual counterparts nor do they leverage the advantages of textual programming (Leitão et al., 2012) – such as the robust abstraction mechanisms that facilitate the effective handling of complex problems and the ability to define bespoke routines and geometric descriptions that are applicable across software platforms.

2. A Design-Centred Computational Geometry Environment

A critical reconsideration of what constitutes an effective computational design environment ultimately requires a proper understanding of the target user base.

Members of the computational design community are typically seasoned users of CAD software, and possess an intermediate to advanced level of understanding of computational geometry. However, lacking any direct experience in software development, they typically have a low to novice level of familiarity with algorithmic design. It is our assertion that this tension between a deep understanding of geometry and a shallow understanding of algorithm creates a limitation in the capabilities of our target audience, one that may be mitigated by the proper computational design environment.

This approach is sympathetic with other programming environments designed for this purpose. Autodesk's DesignScript, for example, replaces certain general-purpose functionality with features specific to the demands of a particular domain of use. In this environment, users are permitted to interchangeably pass both lists and singletons into certain methods for the creation of geometry. This feature, while not supported by most general-purpose programming languages and considered questionable practice by expert programmers, is championed by the authors of DesignScript (Aish, 2012) who argue that such features would be considered desirable by the target user base accustomed to an associative CAD-like approach to collections of objects. In what follows, we describe Decodes, a computational geometry environment in development that is built upon three principles that the authors consider critical for the specific needs of the target user base of the architectural design community.

2.1. HOST-INDEPENDENCE

Computational design in architecture requires the ability to work between diverse and often incongruous descriptions of the built environment. The negotiation of these incongruous models is typically handled within scripts that interoperate across software platforms. To more effectively support design, a computational environment should provide mechanisms for easily operating within and between a range of host environments, as well as the ability to act independently of any host environment. The authors seek to develop a computational design environment that better addresses the need for host-independence through the following tactics:

- **Building up from Scratch:** a tabula rasa approach that involves developing a set of primitive elements independent of any CAD environment, sidesteps the associated unpleasantries of working with the complexities of a full-featured CAD environment. The result is a stand-alone environment that is lightweight and focused on foundational concepts to computational design.
- **Broadening the User Base:** this environment is written in a non-compiled scripting language able to be interpreted on the most widely used operating systems (windows, OSX, Unix) in architectural practice, thus encouraging an active use culture that transcends the boundaries between software platforms and operating systems.

- **Interoperating:** a set of routines for translating geometry in and out of a selected set of CAD software facilitates integration into the native scripting environments in such a way that it effectively replaces host-native scripting.

The approach outlined above requires the creation of a core set of geometric objects and operations able to be translated into each member of a set of supported CAD platforms and operating systems. A cross-platform capable scripting language is needed for this purpose. Python was chosen as a base language for the Decodes library, due to its cross-platform capability, its current level of integration with existing CAD platforms, and the availability of abstraction structures appropriate to a computational design audience (as further elaborated in sections below).

To this end, a set of protocols must be developed and maintained for translating between each supported CAD environment or file type and core Decodes geometry. Every construct or property that does not easily translate into existing CAD software requires a decision as to how it will be interpreted. For example, meshes are very generalizable geometric descriptions that are supported by most CAD platforms, but the details of their implementation may vary slightly. In OBJ file formats, meshes may contain either triangular or quadrilateral faces, while certain routines in the Decodes library rely upon exclusively triangular faces. While mesh triangulation represents a relatively trivial translation, more complex geometric descriptions (such as subdivision surfaces and NURBS descriptions of solid-bodies) will require more elaborate translation protocols, or may not be supported at all. Naturally, there already exists many rich, host-dependent geometry libraries; we take advantage of these by wrapping features in these libraries that are well implemented and in-line with the values of our approach.

Protocols for translating from a CAD environment to Decodes are packaged and termed “innies”. Some innies allow for a limited set of user interactive geometry selection or parameter definition. Protocols for translating from Decodes to external environments are called “outies”. Outies allow for the definition of a limited range of geometric object properties, such as layer, colour, object name, etc. At the time of writing, innies and outies have been developed and are functioning at a basic level for a number of packages, while others are planned but not yet fully implemented. Table 1 describes the current state of development for a number of innies and outies. Context-free outies are also possible, either through a “window” (via OpenGL) or an output file (via DXF, OBJ, or SVG), allowing designs to be developed by operating purely within the Decodes library.

Figure 1, below, depicts the same script run in three distinct CAD environments (AutoCAD, Rhino, and Grasshopper) on two different operating systems (OSX and Windows 7). The script was run without modification, and is dependent only on a pre-defined environment variable that defines the scripting context.

Table 1. Translation protocol state of development (current at the time of writing).

Platform	Innie Implementation	Outie Implementation
Rhino (McNeel)	Initial	Full
Grasshopper (McNeel)	Full	Full
AutoCAD (Autodesk)	None	Partial
Sketchup (Google)	None	Partial
Maya (Autodesk)	None	Initial
SVG	N/A	Partial
OpenGL	N/A	Initial

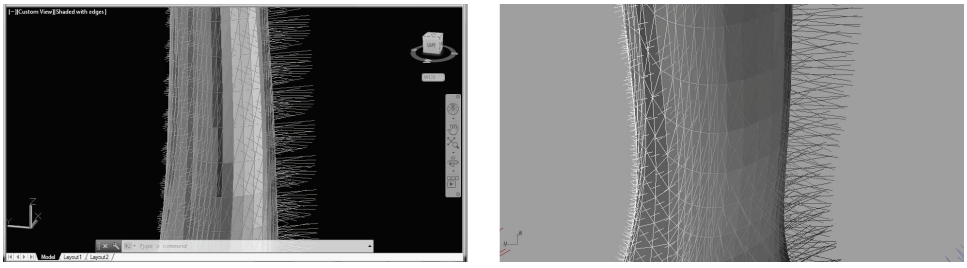

Figure 1. Results of a script, written in Decodes, that calculates the average yearly solar incident radiation across a user-defined mesh, as manifest in AutoCAD and Rhino.

Applying these tactics, the resulting environment is one in which scripts can be reused and easily shared across platforms through simple copy-paste operations. The anticipated impact of this feature is a user base that is expanded to include existing users of all of these platforms, and the creation of broader communities of like-minded users centred on specific computational design approaches rather than on CAD platforms.

2.2. DOMAIN-SPECIFICITY

Computational design depends upon a host of geometric and algorithmic constructs. Some, such as NURBS surfaces and subdivision meshes, are generalizable to other modes of practice and other disciplines, and are typically implemented well in existing CAD platforms. Such constructs are versatile and extensible, but also non-specific and do not themselves directly support high-level computational design procedures. Other constructs, such as recursive tiling definitions and particle-spring systems, are more idiosyncratic to specific domains of practice or to specific design approaches. Such constructs are very useful for the particular task

for which they were developed, but are often difficult for the end-user to modify. An effective computational geometry environment must balance two competing demands:

- **Richness** – access to a large set of commonly used domain-specific geometric and algorithmic constructs, implemented in a robust and flexible manner.
- **Extensibility** – the ease of understanding, modifying and extending this set to accommodate domain-specific tasks not supported by core functionality.

These competing needs are satisfied using the following tactics:

- **Remain Transparent:** the library is implemented within an end-user-accessible scripting language, removing the need for an API. This lowers the barrier for users to contribute extensions and modifications to the core functionality. An active user-base contributing extended libraries can override the natural limitations of a simple, less fully-featured library.
- **Invent Here, Find Elsewhere:** a stand-alone geometric library is developed with functionality specific to the computational design community. Three levels of extensibility are envisioned: a "core" set of basic primitives and functions applicable across CAD platforms, a "contributed" set of extension libraries that provide primitives and functions aimed at a more specific or idiosyncratic task, a set of "hooks" for taking advantage of primitives and functions already implemented in existing CAD platforms.

An approach based on these tactics requires the development of a geometric modelling kernel with functionality specific to the computational design community. Developing this "lean" geometry library from scratch offers the opportunity to reconsider conventional approaches to the core set of primitive objects and operations commonly implemented in CAD software (such as points, vectors, rays, meshes and surfaces), as well as to consider higher-level geometric types that might make beneficial additions to this core set.

The Decodes kernel accommodates the needs of the computational design community through two mechanisms. The primary mechanism is the natural extensibility of Decodes and the anticipated user community actively contributing extensions. As highly valuable extensions are developed and refined, they may be integrated into the core library. The secondary mechanism is the framework of the core library itself, which was developed through an examination of use-cases specific to architectural design. A concurrent thread of research examines situations commonly encountered in architectural design, and has identified seven broad categories of computational strategies that encapsulate tactics that are commonly employed in practice. These are: Aggregation and Tiling, Design Space Exploration, Geometric Decomposition, Agent-Based Forms, Optimization / Satisficing, Modelling for Production and Fabrication, and Interoperating.

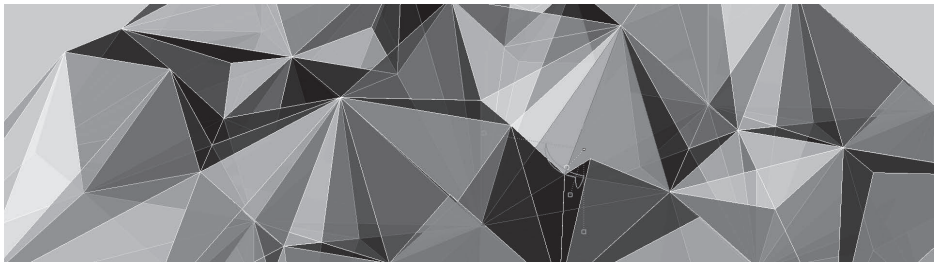

Figure 2. Results of a Decodes script that produces a set of Danzer tiles, and employs transformations through the definition of point bases.

This survey of common practices and methods forms the basis of the selection of core methods and features available in Decodes. At the time of writing, the following core geometric classes have been identified and developed to a basic functioning state: Vector, Point, Line, Ray, Line Segment, Mesh, Polygon, Polyline, Curve, Surface, Coordinate System, Transformation, Intersection. Considerations have been made for the multiple design paths that may be taken to similar solutions. For example, Figure 2, below, depicts the results of one of two built-in tiling definition – a 2d pinwheel tiling definition and a 3d Danzer tiling definition – each enacted using a similar means of structuring transformations. Either of these tiling systems may be defined in multiple ways (via Transformations, Coordinate Systems, or by manipulating the underlying Basis of point collections), conforming to the preferred framework brought to the problem by the user.

The result is a transparent, end-user-extensible, geometry library that better addresses the specific needs of the computational design domain. As this is being built from scratch, the geometry kernel currently trades domain-specificity and transparency for complexity and efficiency. On-going development and adoption by an active user base will naturally lead to an ever-richer library of geometric operations and more efficient implementations.

2.3. CONTEXT-APPROPRIATE ABSTRACTION

Designers as a group have been shown to benefit from a particular combination of abstraction mechanisms in their scripting and programming languages. A computational geometry environment for the design community should provide both access to geometric constructs at a lower-level than often found in CAD software, and access to code structures at a higher-level than often found in programming languages and older styles of scripting languages. The authors seek to develop a

computational design environment that offers a more appropriate level of abstraction for a design audience through the following tactics:

- **Providing Low-level Access to Primitive Geometry:** Following the principle of domain-specificity, a set of primitive geometric objects and operations have been developed that target approaches specific to design computation.
- **Exploiting High-Level Features of Modern Scripting:** Modern high-level scripting languages contain a host of features useful for designers without formal training in computer science, including list comprehension, dynamic typing, and support for multiple programming paradigms.

The approach outlined above is primarily manifest in the choice of base scripting languages used for Decodes, but also has ramifications for the coding style employed in development and the abstraction mechanisms that are available to the end user. Python was chosen as a base language due to its cross-platform capability and for the high-level features it offers such as list comprehension, dynamic typing, and support for multiple programming paradigms. Of these, support for dynamic typing has had the most impact in the development of the Decodes library.

A programming language is considered to be dynamically typed if type-checking is performed at run time rather than at compile-time. Under this model, variables are not typed but values are. Dynamically typed languages, such as Python, Ruby, Lisp, JavaScript, VB Script, and Lua, are often used in generative design applications. The popularity of this model may be due to the large number of novice programmers in the generative design community, and the perception that typed variables is a difficult concept for new users to apprehend. While dynamic typing is itself a context-appropriate feature for a design audience, as it removes an often-confounding obstacle to code development, it is the coding style that dynamic typing engenders that has enabled the most innovative feature of the Decodes library. “Duck typing” is a widely employed style of programming using a dynamically typed language. To conform to this coding style, a programmer should be less concerned with an object’s inheritance from a particular class or interface (what an object “is”) than the properties and methods available to that object at any given time (what an object “does”).

Applying a duck typing style to the development of the Decodes allows for the “flexible basis” abstraction mechanism. Point objects in architectural design applications are implicitly defined in Cartesian geometry – relative to an ortho-normal coordinate system in R3. Many computational design environments support a limited set of alternative definitions, including cylindrical coordinate systems, as well as points defined relative to parameterized objects such as curves and surfaces. We may understand a point object to be defined relative to any of these bases, where

the X,Y, and Z parameters of any given point are evaluated by its basis to produce a "world" point in R3. Using a static-typed language, only objects that explicitly announce themselves as deriving from a "basis" super class or interface may perform in this manner, implicitly limiting the set of objects which can act as the basis of a point. A duck typing programming style removes this limitation, allowing any user-defined object to act as a basis.

A "flexible basis" for points is a useful abstraction mechanism for the computational design context, as it allows for the implicit encapsulation of transformations and aggregation structures. A panelization study has been developed that demonstrates the use of a custom basis to define panel placement along a surface. In this situation the custom basis offsets the need for a surface definition, and provides the user with a more precise and reliable means of control when compared to NURBS surface definitions.

References

Aish, R.: 2012. DesignScript: Origins, Explanation, Illustration, *Computational Design Modelling. Design Modelling Symposium,* Berlin 2011, Berlin: Springer, 1–8.

Celani, G. and Vaz, C.E.V.: 2012, Programming Languages For Generative Design: A Comparative Study, *International Journal of Architectural Computing*, **10**(1), 121–138.

Kilian, A. and Ochsendorf, J.: 2005, Particle-spring Systems for Structural Form Finding, *Journal of the International Association for Shell and Spatial Structures*, **46**(2).

Leitão, A., Santos, L., and Lopes, J.: 2012, Programming Languages For Generative Design: A Comparative Study, *International Journal of Architectural Computing*, **10**(1), 139–162.

Pentilla, H.: 2003, Architectural-IT and Educational Curriculums – A European Overview, *International Journal of Architectural Computing*, **1**(1), 102–111.

"RhinoCommon Software Development Kit": 2013. Available from: <http://4.rhino3d.com/5/rhinocommon/> (Accessed February 17, 2013).