Generative Design for Building Information Modeling

Bruno Ferreira[1], António Leitão[2]
[1,2]INESC-ID/Instituto Superior Técnico
[1]bruno.b.ferreira@tecnico.ulisboa.pt
[2]antonio.menezes.leitao@ist.utl.pt

Generative Design (GD) is a programming-based approach for Architecture that is becoming increasingly popular amongst architects. However, most Generative Design approaches were thought for traditional Computer Aided Design (CAD) tools and are not adequate for the Building Information Modeling (BIM) paradigm. This paper proposes a solution that extends GD to be used with BIM applications while preserving and taking advantage of its ideas. The solution will be evaluated by developing a connection between Revit, a well-known BIM tool, and Rosetta, a programming environment for GD, and by implementing the necessary programming language features that allows GD to be used in the context of BIM tool.

Keywords: *Generative Design, BIM, Revit, Rosetta, Racket, Programming Languages*

INTRODUCTION

Computer Aided Design (CAD) applications increased the efficiency of design activities and allowed architects to produce more accurate and precise drawings that could be more easily edited without the need of manually erasing and redrawing parts of the original design.

Nevertheless, modeling complex and creative geometry can still be a challenge in a CAD tool and changing the model continues to present some difficulties as the degree of flexibility provided by these tools is not sufficient. Generative Design (GD) is used as a solution to these problems (McCormack et al. 2004).

GD can be described as form creation through algorithms (Terdizis 2003). This approach allows the generation of different solutions just by changing the constraints and the requirements implemented in a given program. These programs can include complex algorithms that generate geometry that is very difficult to create by manual means.

Recognising the advantages of the GD approach, many tools were developed that allow the creation of GD programs. These tools were also tailored for architects with basic programming experience, reducing the programming skills needed to use them.

Rosetta is one of those tools. It provides a programming environment for GD, allowing the development of scripts using different programming languages. These scripts can then be used to generate models in different CAD tools.

Nowadays, Building Information Model (BIM) tools are replacing the traditional CAD tools. Firstly, because they offer a set of features that go beyond CAD, and, secondly, because many governments are making the use of BIM obligatory in projects.

Unfortunately, tools like Rosetta were developed for CAD applications, and thus, are not adequate for the BIM paradigm.

This happens because CAD tools only deal with geometry while BIM objects are far more than that. They are defined by the parametric rules they contain as well as by their properties like materials, finishes, manufacturer specifications and even price.

These properties allow BIM tools to detect problems in the design, such as a pipe that is placed in the same location as a window. The parametric rules allow the object to adapt to its usage when inserted in a project.

Finally, all the information about the project life cycle is stored in these objects and can be used to create documents related to fabrication, cost estimation and even building management.

CAD tools do not require this information but BIM does. This fact alone changes the way users interact with BIM and also changes how GD programs have to be written. GD tools that only work with CAD tools must allow all these features in order to communicate with a BIM tool but most of them do not have the proper support for that.

Developing a GD program for BIM is a problem that exists nowadays and the solution offered by BIM tools, like Revit, is an Application Programming Interface (API). The API provides a way to use BIM with a programming approach. Unfortunately, the use of the API requires knowledge of programming languages, like C# or C++, and computer science concepts, such as transactions and polymorphism, which assumes considerable programming experience.

The objective of this paper is to present a solution that allows novice programmers to write GD programs for BIM applications.

RELATED WORK

Several tools were analysed to guide us in the development of our solution. The main focus was on tools that allow users to write GD programs for BIM applications. This includes tools that are already available in BIM applications as well as plug-ins developed for the same purpose.

Grasshopper 3D and Lyrebird

Grasshopper 3D is a graphical programming language developed for architects as a plug-in for Rhinoceros 3D CAD.

Programs written in this language represent a data flow graph that consists of a group of components and the connections between them. These components can be selected from a series of menus and dragged to the working environment. The components can represent functions, parameters or even geometry and they are connected with lines. This allows the users to create complex algorithms by combining components with the connectors.

It is important to notice that the components are not the only way to use Grasshopper's functionalities. They can also be extended by using scripting components to write code using VB.NET, C# or Python programming languages (Payne and Issa 2009).

Because it is a graphical language, Grasshopper is easier to learn and start using, which made it popular amongst architects. However, the graphical aspect of the language is also a disadvantage because, as programs grow, the amount of connections between components makes the program difficult to understand and even features such as sliders stop working as intended due to performance issues. **Figure 1** shows a complex program that illustrates the problems mentioned.

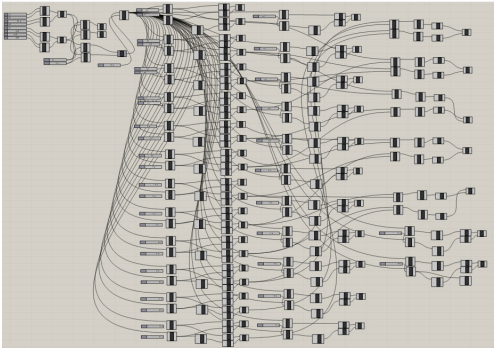

Figure 1
An example of a complex parametric model in Grasshopper. Retrieved from [3] on June 2015.

Although Grasshopper was designed to be used only with Rhinoceros, it can also be extended with plug-ins. This feature allowed the development of plug-ins, such as Lyrebird, that made possible the usage of the language with other applications, including BIM tools.

Lyrebird is a plug-in developed by LMN Architects as an interoperability tool between Grasshopper 3D and Revit. This plug-in enables the usage of Grasshopper to structure the information needed to produce the desired model in Revit. This information is then used to identify and instanciate the correct families with the correct rules and parameters.

As a result, Lyrebird is more focused on sending the correct data between the applications, instead of translating the geometry between them [1]. There is only one component on the Grasshopper side that receives the information and then sends it to the Revit side. There, a command is selected to add a new object or manage an existing one.

For example, to create a column, a line is used as input. Next, on the output of the component, the family to use in Revit is specified. On the Revit side, the information is received and the user confirms the creation of the new element. A column of the specified family is then created, using the line as input. The line is used as abstract information that, combined with the family of the BIM object, allows the creation of the actual object.

DesignScript

DesignScript is a programming language that is heavily influenced by design principles. It was created by Robert Aish to be, not only a production modeling tool, but also a full-fledged programming language and a pedagogical tool (Aish 2012).

As a programming language, DesignScript is seen as an *associative* language as it maintains a graph of dependencies between the variables used in a program. Any change in one of the variables is propagated throughout the program. This means that if we have a variable *a*, and if *b* is defined as *a* + *1*, a change in *a* will also modify the value of *b*. This is a *change-propagation* mechanism, similar to the update mechanisms available in associative CAD tools. The language is also a domain-specific language as it contains primitives for design and geometry.

As a modeling tool, DesignScript tries to introduce concepts that are easily understood by users that are not accustomed to design with the help of a programming language. This is achieved by allowing the developers to use its logical framework in order to produce the design models, and also facilitating an exploratory approach to the tool involving refactoring of the produced models (Aish 2012).

Finally, DesignScript aims to be a pedagogical tool, as it allows the evolution of the programming skills of its users. Users unfamiliar with programming are able to use a direct approach with a graph node diagramming interface that is simple and requires little to no understanding of programming concepts (Aish 2013). **Figure 2** shows a program created with the graph approach. However, as their design becomes more complex, users might feel the need to learn more advanced programming concepts. The node-to-code functionality of DesignScript allows a transition between the graph representation and a script that initially presents a logic very similar to the original graph. For users that desire to produce more complex programs, this script might be changed into a normal script.

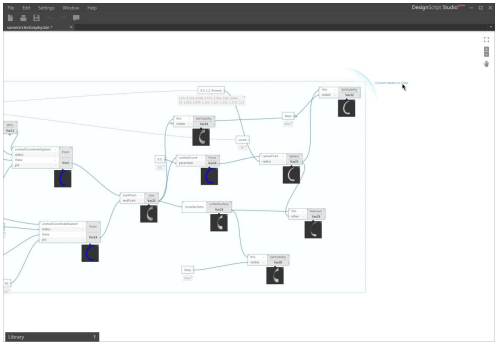

Figure 2
An example of a program created in DesignScript. Retrieved from [4] on December 2014.

Dynamo

Dynamo is a plug-in for Revit that is strongly influenced by graphical programming languages such as Grasshopper for Rhino.

Just as Grasshopper, users create a workflow by introducing nodes that are connected to each other through wires associated with the ports that each node contains. A port from an element can only be connected to another port of a matching type. This means that the input and output port must have compatible types.

Nodes can represent several Revit elements, such as lines, or functions, such as mathematical functions. Users can also define custom nodes in order to extend the functionality provided by Dynamo.

Figure 3 shows a program written with Dynamo nodes and its result.

Dynamo also supports the use of code blocks, which are elements containing small scripts written in a textual programming language, such as Python. These code blocks allow the creation of small algorithms that introduce more complex functionalities that are not possible to create with the other nodes.

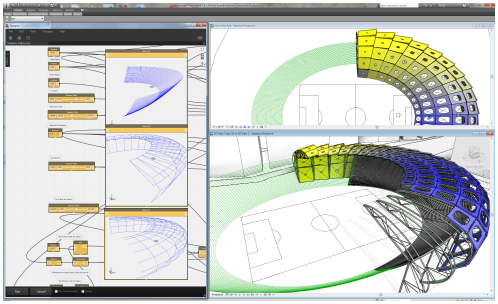

Figure 3
A program written with Dynamo and its result. Retrieved from [5] on December 2014.

Geometric Description Language

Geometric Description Language (GDL) is a parametric programming language for ArchiCAD that allows the creation of scripts that describe objects, which are called library parts.

This language, similar to BASIC, requires the definition of several scripts that include the model description and the parameters of the new object (Nicholson-Cole 2004).

Each object is described with a sequence of commands that describe its geometry. In similarity to OpenGL, a matrix stack implements transformations like translations, rotations and scales. The transformations currently in the stack when creating a shape are the ones that are going to influence it.

Figure 4 shows a program written in GDL. The editor shows the sequence of commands that produces the result, visible on the top left corner of the image.

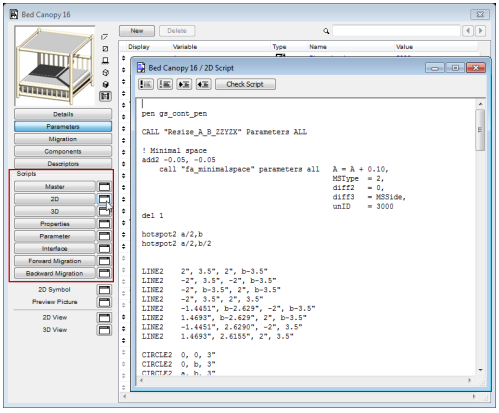

Figure 4
A GDL program that creates a bed canopy. Retrieved from [6] on May 2015.

However, GDL only allows the creation of simple geometry and does not take advantage of the BIM functionalities of ArchiCAD.

GenerativeComponents

GenerativeComponents (GC) is a parametric and associative system developed for Bentley's Microstation.

This system is propagation-based so the user has to determine the rules, relationships and parameters that define the desired geometry. This propagation-based system consists of an acyclic directed graph that is generated by two algorithms: one that is responsible for ordering the graph and the other that propagates values through it (Aish and Woodbury 2005).

GC has several ways of user interaction, taking into consideration his skills. The first one is a Graphical User Interface (GUI) that allows direct manipulation of geometry. The second one is by defining relationships among objects with simple scripts in GCScript. The third and final one, is by writing programs in C#, allowing the definition of complex algorithms.

GC shows that a graphical language might be easier to learn but as the user wants to produce more complex models, he will start to produce scripts in a textual programming language.

GENERATIVE DESIGN FOR BIM

Most of the tools analysed in the previous sections have disadvantages. The major one is the fact that they are associated with a specific application. The programs created with them are not portable, which is something that we want to solve with our solution.

Another disadvantage is the programming language that they use. Lyrebird and Dynamo, for example, primarily use Graphical Programming Languages. These languages, although easy to learn, do not scale well with the program complexity.

A Textual Programming Language is more flexible but might introduce a barrier for newcomers. The chosen language must be easy to learn and fit for beginners. GC, for example, uses C# which is a very complex language, and GDL uses a language similar to BASIC which is obsolete. This is something that we address in our approach.

In the following subsections our proposed solution is described. It aims to give an approach that allows users to write portable GD programs for BIM tools.

The primary components of the solution are an Integrated Development Environment (IDE), an abstraction layer and a communication component. We will now discuss these components.

Integrated Development Environment

Our solution uses a programmatic approach based on textual programming languages since they are more flexible and scale better with the program complexity. For this reason users need an editor in which they write their code.

The language of the editor must be fit for a beginner. For example, the Python language is very popular and is being used to teach beginners how to program. Also, languages like Python have an IDE that help users write and debug their programs. These features are very important for beginners since they have difficulties writing their first programs and these IDEs help them overcome some initial barriers.

Abstraction Layer

To write GD programs, users need functions that give access to BIM functionality. These functions must allow users to instantiate BIM objects and define all the information needed in order to produce an accurate project.

For example, to produce a wall with a door in it, there must be a function that creates walls. This function might receive the height, length, position and the type of wall in order to create the correct BIM object. Then, a function that creates the door is used and it must receive, not only the position, height and type of door but also the wall that will serve as host of this object.

In addition to this, to indicate positions and create lines and arcs, abstractions for these concepts are needed as well as functions that allow their creation and usage.

All these functions as well as others that create and manipulate BIM objects are included in an abstraction layer, so that they can be used in all supported BIM tools.

However, some features are unique to certain BIM tools, so we will offer them as functions that will be available only to that specific BIM. The user can choose to use them, giving up the portability of their programs, but gaining the ability to take advantage of the specific features the tool has to offer.

BIM Communication

Finally, in order to execute the user programs and generate the result in the desired BIM tool, a component that communicates with the tool is needed.

Since most of the BIM tools have an API that exposes their features in a programatic way, this component can be a plug-in for the tools written with the aid of that API.

All the functions of the abstraction layer have a correspondent one in this component. When one of them is used, the needed information is sent to this component and then the correspondent function is executed. Taking advantage of the API, these functions will then produce their results in the BIM tool.

Figure 5 shows all the components of our solution and how they are related.

EVALUATION

In this section, we perform an evaluation of our solution resorting to an implementation that uses Revit. This implementation takes advantage of Rosetta and Revit connecting the two of them and introducing the latter as a new back-end for Rosetta.

Rosetta Programming Environment

Rosetta already has a development environment designed for beginners and provides many concepts that architects already use on a day to day basis.

By using Rosetta, users are able to choose the language in which they want to program. Racket, Python, Javascript and Processing are some examples of languages that are already available. This helps users to overcome the need of learning a new programming language if they already know one that is available. If they have yet to learn how to program they can choose any of the languages available since most of them are pedagogical languages. The Rosetta IDE also supports debugging and syntax highlighting for all the languages available as front-ends.

Also, by using Rosetta, we can take advantage of many functionalities without the need of re-implementing them. One example are the coordinate systems.

Of course, in order to add a BIM, such as Revit, as a back-end, new functions and abstractions must be created in Rosetta since the current ones were developed for CAD tools. These new functions and abstractions must be included in an abstraction layer explained in the following section.

Abstraction Layer

This layer contains all the functions and abstractions that can be used to produce results in BIMs, in this case Revit. All these were created taking into consideration the way architects do their work in order to facilitate their understanding and usage when writing their programs.

Since Revit can produce Project and Family documents, we created different functions that can be used in each one.

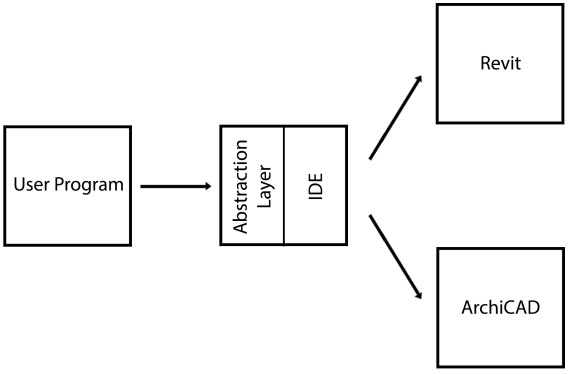

Figure 5
Overview of the architecture of the solution.

For Family documents, we have functions that create simple geometry. These functions are very similar to the ones available for CAD tools in Rosetta. This allows users to port some of their algorithms directly to the new back-end and produce their complex geometry in it. This geometry can then be used to define new families and family instances that can be used in Project files.

Figure 6 shows a program that was originally written for a CAD, now being used with Revit. To do this we simply re-implemented the geometry functions to work with BIM tools.

The functions and abstractions for Project files are where the main differences between BIM and CAD are felt. In these files the user creates instances of families and uses them to develop their project. So these are BIM objects and not simple geometry.

To have BIM objects, concepts like walls, doors and windows must be introduced. Also, the functions that deal with these concepts are defined taking into consideration how these objects are created by the user in the graphical interface. To create a simple wall, for example, the user draws a line between two points and a wall of the selected type is created along that line. So, the function that creates the wall is used in a similar manner, receiving points that define the line where the wall will be created.

Another significant difference between the CAD and BIM tools is the restrictions and the relationships between objects. When creating a wall with a door, we know that they are related because of their placement. In a CAD tool, the door is placed in a position where the wall is but they are not related. When creating the GD program for CAD tools, we use restrictions that make everything fit in the right place but these restrictions are only in the code and not in the generated model. In BIM tools these relationships are present in the generated objects. For example, a door cannot be created without a host. A wall must be given as a host and the BIM tool will create and modify the necessary properties for the door to fit perfectly.

So all the operations take this into account and require all the elements necessary to create these re-

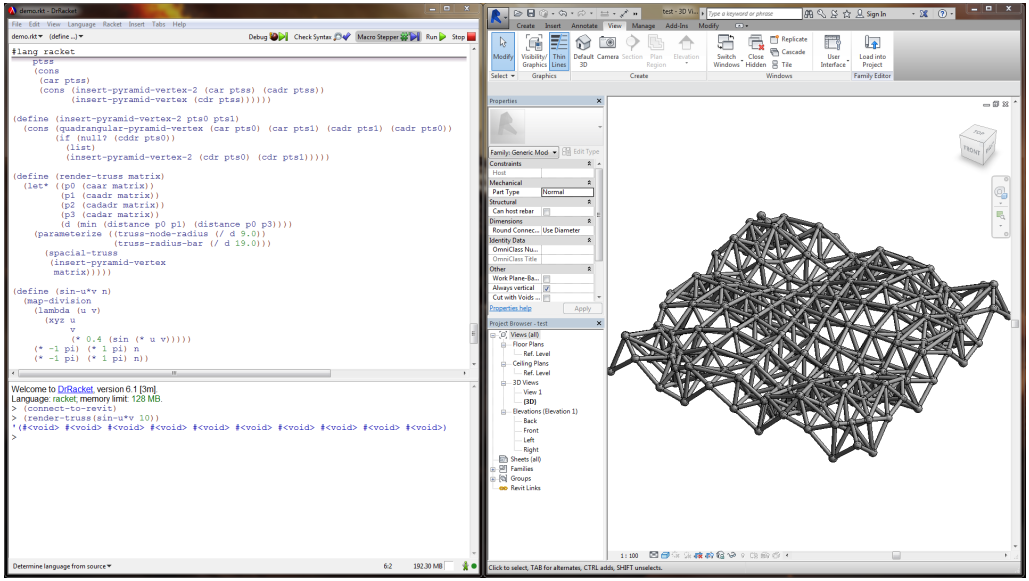

Figure 6
A truss generated in Revit with a GD program.

lationships. To create a floor the user must indicate in which level it will be placed. To create a door, he must say in which wall it will be hosted.

Figure 7 shows a script on the left that creates a tower with slabs and columns to support them. The result in Revit can be seen on the right side of the image.

This example uses functions included in the abstraction layer that create BIM objects. The levels are created with the height specified by the user. After that, the floors are created in those levels with a function that creates round slabs taking into account the center of the slab and a radius. Finally, the columns are created in a specified position and their height is determined by the two levels indicated by the user. The first level is where the base of the column is and the other is on the top.

As seen in the example, all these functions have a visual result in Revit. In order to achieve this, Rosetta must comunicate with the desired BIM tool and send all the information that is necessary to produce the model. All the information is serialized using Google Protocol Buffers and transmitted to the BIM tool via sockets. This guarantees that all information is sent in a format that can be easily reconstructed on the other side. There, the information is used in a plug-in that was developed for Revit, which we will explain in detail in the following section.

Revit Plug-In

The plug-in was created using the RevitAPI made available by Revit. This plug-in communicates with Rosetta in order to receive all the information needed to create the objects.

This plug-in is written in C# and was developed using the Microsoft Visual Studio programming environment. The abstraction layer previously mentioned was developed to hide the complexity of the API.

All the functions in Rosetta have a correspondent one in the plug-in and these are the ones that communicate directly with the BIM and create all the needed elements. In order to produce these elements, the information received from Rosetta

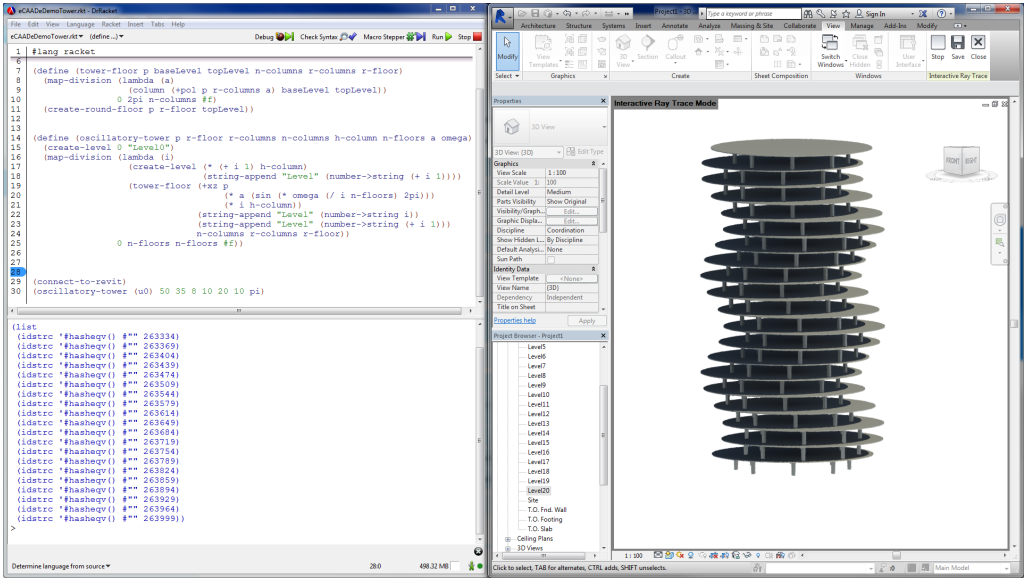

Figure 7
A tower generated with BIM objects with a GD program.

must be read from the socket and deserialized with the Protocol Buffers to reconstruct the information. Then, this information is used as parameters for the API functions that create the objects in the BIM. These functions return an identifier that is serialized and sent to Rosetta allowing the user to reference those objects in the future.

To create a more explorative interaction, this plug-in is non-blocking which means that Revit will not block while the user is executing his program. This means that it is possible to interact with Revit while the program is running, for example, to see the results using a different perspective. However, this comes with a penalty in performance. If the user wishes better performance, a blocking mode can be used that does not allow to interact with Revit until the program finishes.

An important feature provided by Rosetta is the Read-Eval-Print-Loop (REPL). This is an interactive window visible in the bottom left of **Figure 6**, where the user can test functions and see the results in the BIM without the need of writing another script. The REPL receives the user input, evaluates it and shows the result to the user, then becoming ready to receive another input. If the user wants to see how a certain function works with a specific parameter, he can use the REPL and quickly see if the function behaves as he expects.

CONCLUSIONS

The GD approach created for CAD applications proved very useful but nowadays these tools are being replaced with BIM. This paradigm is very different from CAD, so the approach originally created no longer fits.

To solve this problem we propose a solution that allows users to create GD programs with a set of abstractions designed for BIM tools. Our solution extends Rosetta, a IDE for GD. By taking advantage of Rosetta, users have a development environment fit for the needs of beginners. Due to the many front-ends available, users can pick the programming language they prefer. Also, since the abstraction layer we implemented takes into consideration generic BIM concepts and not the concepts that are exclusive of a given tool, we expect that users will be able to create portable programs that explore different BIM tools.

We are currently evaluating our solution with a a group of architects that are also novice programmers. In future work we plan to expand this solution adding more functionality to the BIM abstraction layer. Also, we plan to support aditional BIM back-ends such as the ArchiCAD back-end, which is already being developed.

ACKNOWLEDGEMENTS

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013, and by the Rosetta project under contract PTDC/ATP-AQI/5224/2012.

REFERENCES

Aish, R. 2012 'DesignScript: origins, explanation, illustration.', *Computational Design Modelling*, Springer Berlin Heidelberg, pp. 1-8

Aish, R. 2013 'DesignScript: Scalable Tools for Design Computation', *eCAADe 2013: Computation and Performance–Proceedings of the 31st International Conference on Education and research in Computer Aided Architectural Design*, Delft, The Netherlands, pp. 18-20

Aish, R. and Woodbury, R. 2005 'Multi-level interaction in parametric design', *Smart Graphics*, Springer Berlin Heidelberg, pp. 151-162

Azhar, S., Hein, M. and Sketo, B. 2008 'Building Information Modeling: Benefits, Risks and Challenges', *Proceedings of the 44th ASC National Conference*, Auburn, Alabama, USA

Eastman, C., Teicholz, P., Sacks, R. and Liston, K. 2008, *BIM Handbook: A Guide to Building Information Modeling for Owners, Managers, Designers, Engineers and Contractors*, John Wiley & Sons, Hoboken, New Jersey

Ibrahim, M., Krawczyk, R. and Schipporeit, G. 2004, 'Two Approaches to BIM: A comparative study', *eCAADe Conference*, 22, pp. 610-616

Leitão, A., Santos, L. and Fernandes, R. 2014 'Pushing the Envelope: Stretching the Limits of Generative De-

sign', *Blucher Design Proceedings*, pp. 235-238
Lopes, J. and Leitão, A. 2011 'Portable Generative Design for CAD Applications', *Proceedings of the 31st annual conference of the Association for Computer Aided Design in Architecture*, pp. 196-203
McCormack, J., Dorin, A. and Innocent, T. 2004 'Generative design: a paradigm for design research', *Proceedings of Futureground, Design Research Society*, Melbourne
Nicholson-Cole, D. 2004, *Introduction to Object Making with Archicad: GDL for Beginners*, Graphisoft R&D
Payne, A. and Issa, R. 2009, *The grasshopper primer. Zen 'Edition*, Robert McNeel & Associates
Terzidis, K. 2003, *Expressive form: a conceptual approach to computational design*, Spon Press, London and New York
[1] http://lmnts.lmnarchitects.com/bim/superb-lyrebird/
[2] http://dynamobim.com/learn/
[3] https://11arch461.files.wordpress.com/2011/10/def_extent.jpg
[4] http://through-the-interface.typepad.com/.a/6a00d83452464869e20192ac16a8d2970d- pi
[5] http://inthefold.autodesk.com/.a/6a017c3334c51a970b019b01bc21de970c-pi
[6] http://helpcenter.graphisoft.com/guides/archicad-18-int-reference-guide/user-interface-reference/dialog-boxes/gdl-geometric-description-language/gdl-object-editor/