Generative Design for Building Information Modelling

Bruno B. Ferreira

Instituto Superior Técnico, Universidade de Lisboa
bruno.b.ferreira@tecnico.ulisboa.pt
http://tecnico.ulisboa.pt/

Abstract. 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 recent Building Information Modelling (BIM) paradigm. This paper proposes a solution that extends Generative Design so that it can be used with BIM while preserving and taking advantage of BIM 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: BIM; Revit; Racket; Programming Languages; Generative Design; Rosetta

1 Introduction

Throughout the years, architects have used different tools to perform their job. In the beginning, they used simple ones like pen and paper, but as the scale and requisites of the buildings changed, the paper-based approach showed its limitations, such as the need to redraw an entire model to correct a mistake. With the advent of information technology, the computer proved to be a great aid for architects and this led to the development of a new and more powerful tool: Computer Aided Design (CAD).

CAD applications increased the efficiency of the 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 [1]. Some of these tools were also able to create tridimensional models of the buildings, enabling a more realistic preview. The design process was obviously changed by these tools, which opened new possibilities for more creative and artistic models.

However, errors were likely to occur and CAD tools were not able to detect them. Moreover, changing a very complex model was still a hard and time consuming task. Normally, several views of the building were created and changing one implied changing all the others. Also, this type of software is only able to deal with geometry, so all the information relevant for the construction of buildings could not be stored in the model itself, causing a documentation problem.

Building Information Modelling (BIM) appeared as a new paradigm shift that presented a solution to these problems.

BIM represents the process of development and use of a computer-generated model to simulate the planning, design, construction and operation of a building [2]. This leads to a model that contains not only the geometry of the building, but also information to support the construction, fabrication and procurement activities needed to realize it [3].

This is possible because BIM uses parametric objects that contains all the information needed. Parametric BIM objects are defined as a combination of geometric definitions and associated data and rules. They also have the following properties [3]:

- Geometry is integrated non-redundantly, and without inconsistencies. All views of an object are consistent with its shape.
- Parametric rules for objects automatically modify associated geometries when inserted into a building model or when changes are made to associated objects. For example, a door will fit automatically into a wall.
- Objects can be defined at different levels of aggregation, so a wall can be defined as well as its related components.
- Objects can be defined and managed at any number of hierarchy levels. For example, if the cost of a wall subcomponent changes, the cost of the wall should also change.
- Objects rules can identify when a particular change violates object feasibility.
- Objects have the ability to import or export sets of attributes such as structural materials, acoustic data and energy data to other applications and models.

The parameters and rules that describe these objects are defined by the user or the vendor and are created to fit the needs of the project in hands. Hence in models produced with this approach it is possible to find objects with contextual semantics, such as walls and doors, instead of simple geometry. For instance, instead of representing a wall with a simple geometric object, a wall object is used. This object has properties that describe not only the geometrical dimensions of the wall, such as length, width, and height, but also the materials, finishes, specifications, manufacturer, and price [4].

A BIM model contains information about the life cycle of the building and this can be easily extracted to produce several documents needed throughout the project duration.

The following list describes some of the uses of a BIM model [2]:

- **Visualization**: 3D renderings can be easily generated in-house with little additional effort.

– **Fabrication**: it is easy to generate shop drawings for various building systems. For example, the sheet metal ductwork shop drawing can be quickly produced once the model is complete.
– **Reviews**: fire departments and other officials may use these models for their review of building projects.
– **Forensic analysis**: a building information model can easily be adapted to graphically illustrate potential failures, leaks, evacuation plans, etc.
– **Facilities Management**: BIM can be used for renovations, space planning, and maintenance operations.
– **Cost estimation**: BIM tools have built-in cost estimating features. Material quantities are automatically extracted and changed when any changes are made to the model.
– **Construction sequencing**: a BIM can be effectively used to create material ordering, fabrication, and delivery schedules for all building components.
– **Conflict, interference and collision detection**: BIM models are created to scale, in 3D space, and all major systems can be visually checked for interferences. This process can verify, for example, that piping does not intersect with steel beams, ducts or walls.

BIM allows a more effective design process by facilitating the reuse and share of information. However, there are also disadvantages in this approach, such as the learning curve of BIM tools and the difficulty of changing the design process.

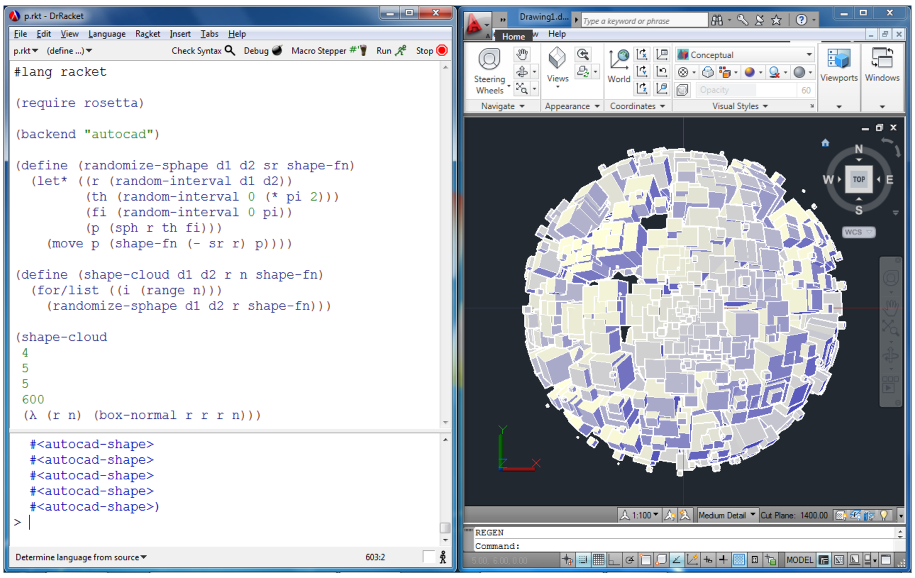
Still, modelling and producing complex and creative geometry can still be a challenge in a CAD or BIM tool and changing the model continues to present some difficulties since the degree of flexibility provided by these tools is not sufficient. Particularly, complex geometry modelling, repeated tasks, and exploration of different solutions are still difficult or cumbersome to implement in these tools. The concept of Generative Design (GD) is a solution to these problems [5].

GD can be described as form creation through algorithms [6]. This approach allows the generation of different solutions just by changing the constraints and the requirements implemented in a given program.

Writing a GD program that describes an architectural model requires a large initial effort, and might be less cost-effective than the typical digital modelling approach. However, the initial cost can be quickly recovered when it becomes necessary to incorporate changes in the design [7]. This flexibility is provided by the advantages that programming languages give to their users. A program does not allow ambiguity and describes the geometry to generate through parameters and algebraic formulas. These programs can also be reused if needed in a different project.

However, writing a GD program requires programming skills and knowledge of programming languages, a requisite that some potential users of this approach do not have. Moreover, each CAD tool must be programmed in its own programming language, which reduces the portability between CAD tools and might introduce difficulties for users who are not familiar with the language available in their preferred CAD tool.

Rosetta is a solution to overcome these problems. Rosetta is a programming environment for GD that allows the usage of different programming languages (called *front-ends*) to produce scripts that can generate models in different CAD tools (called *back-ends*) [8]. Some of the programming languages available, such as Scheme, Racket, and Python, are considered very good targets for architects that want to learn programming. The ability to portably generate models in different CAD tools is another advantage, as it makes the developed programs independent of the CAD tool being used and, thus, more reusable. **Fig. 1** shows a script written in Racket that uses Rosetta in order to produce the model in AutoCAD.



```
#lang racket

(require rosetta)

(backend "autocad")

(define (randomize-sphape d1 d2 sr shape-fn)
  (let* ((r (random-interval d1 d2))
         (th (random-interval 0 (* pi 2)))
         (fi (random-interval 0 pi))
         (p (sph r th fi)))
    (move p (shape-fn (- sr r) p))))

(define (shape-cloud d1 d2 r n shape-fn)
  (for/list ((i (range n)))
    (randomize-sphape d1 d2 r shape-fn)))

(shape-cloud
 4
 5
 5
 600
 (λ (r n) (box-normal r r r n)))
```

```
  #<autocad-shape>
  #<autocad-shape>
  #<autocad-shape>
  #<autocad-shape>
  #<autocad-shape>)
> |
```

```
REGEN
Command:
```

Fig. 1. Rosetta with the AutoCAD back-end

Using Rosetta it is possible to write programs that generate complex, but identical, geometry in several CAD families. These programs include parameters and constraints that define the geometry to create and how each of the elements in the design are related.

Nevertheless, the constraints used in GD programs are only present in the code. Although the design reflects those constraints, all the generated parts are disconnected in the CAD tool, and manually changing one does not change the others. However, BIM applications have several mechanisms that allow propagation of changes in one part to other connected parts.

Since BIM is so rich in information, its use would be preferable to normal CAD tools, and so the use of GD with BIM is a real need. GD would allow quicker and easier iterations in the development of a complex BIM model [9].

Developing a GD program for a BIM is a problem that exists nowadays but the solution offered by BIM tools is an Application Programming Interface (API) that gives users a way to use BIM with a programming approach. In the next section we briefly discuss the API of one of the most successful BIM tools.

1.1 RevitAPI

Revit has an API that allows its users to use programming in order to interact with the BIM. This API is written in C# and can be used with Microsoft Visual Studio.

This API gives the users primitive operations that can produce simple geometry and others that can explore several capabilities of this BIM. This gives the users the opportunity to use GD principles in order to interact with the tool.

However, for several reasons, the API is not suitable for architects that want to start to program. The first reason is the programming language that is recommended to use when developing with the API: C#. This programming language is not suitable for beginners, as it requires that several concepts are already known by its users, such as transactions and polymorphism. The second one is the programming environment: Visual Studio. Although this is a very powerful Integrated Development Environment (IDE), it is not the most suited for newcomers because it overwhelms the user with the amount of functionalities and options that it presents. Finally, the API itself. The RevitAPI has a documentation that is difficult to use for beginners since they might have problems understanding all the concepts that it contains. Also, due to the changes recently made to the API, the documentation and some of the examples found in the Internet are obsolete, which might confuse and create a barrier for users that want to try examples in order to learn how to use the API for simple tasks.

One of our goals is to produce a tool that will solve these problems and give an easier way for users to interact with Revit and other BIM tools.

This paper presents a solution that allows the usage of the GD mechanisms available in Rosetta in a back-end that is a BIM application, instead of a traditional CAD application.

In the next sections, we describe the objectives of this work as well as the related work and the general architecture of our solution. Finally, we discuss the evalutation of the solution.

2 Objectives

The main objective of this thesis is to develop an approach that allows the usage of GD with a BIM tool. This requires the definition of new programming abstractions that allow users to write GD programs that generate models in BIMs and that take advantage of the features that are unique to this type of tool.

In order to do this, the operations needed for the creation of BIM objects must be identified and implemented in such a way that makes the interaction with the BIM as simple as possible, mimicking the process that would be done directly in the BIM.

To evaluate the ideas developed in this thesis, we will implement them in the context of Rosetta, by connecting it to Autodesk's Revit, a popular BIM tool. To this end, we will also develop a simpler Application Programming Interface (API) to interact with Revit. The solution to be developed will then introduce Revit as an additional back-end for Rosetta, making it possible for users to write programs using functions that take advantage of the BIM capabilities.

3 Related Work

In this section the most relevant languages and applications that are a source of ideas for this work are described.

3.1 Grasshopper 3D and Lyrebird

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

Programs written in this language represent a data flow 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 between each other. This allows the user to specify which parameter component connects to or from each of the function components, allowing the implementations of complex algorithms that create and transform geometry.

They also provide the user some of the most common data types like numerical data types and booleans, and data structures such as arrays and vectors. Some functions are also available, such as mathematical functions (sine, cosine and others) and also conditional statements (equality, similarity, larger than and smaller than). The user may also define function blocks that evaluate conditional statements or solve complex mathematical algorithms [10].

Grasshopper also has a slider component that allows the user to easily change the values that are inputted to a specific program and see the change occur in their model.

It is important to notice that this language's functionality can also be extended using scripting components to write code using VB.NET, C# or Python programming languages [10].

Because it is a visual language, Grasshopper is easy to learn and start using, which made it popular amongst architects. However the visual aspect of the language is also a disadvantage because, as programs grow, the amount of connections between components makes them difficult to understand and even features such as the sliders stop working as intended due to performance problems. **Fig. 2** shows a complex program that illustrates the mentioned problems.

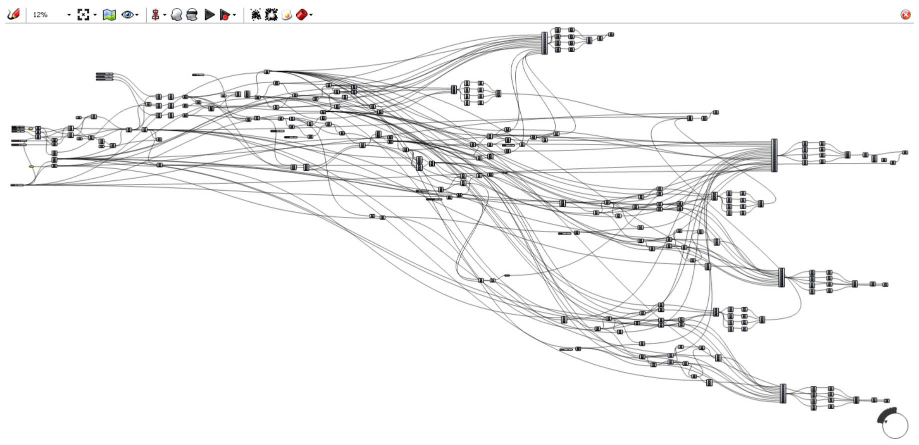


Fig. 2. An example of a complex parametric model in Grasshopper. Retrieved from http://workshopsfactory.wordpress.com/files/2009/08/parametric-table-grasshopper.jpg (retrieved December 2014)

Although Grasshopper was designed to be used only with Rhinoceros, it can also be extended with plug-ins, such as Lyrebird, that allows it with other tools. Lyrebird is a plug-in developed by LMN Architects as an interoperability tool between Grasshopper 3D and Revit [11]. This plug-in enables the usage of Grasshopper to structure the information needed in order to produce the desired model in Revit. To do this, a Grasshopper component receives as input the family type and parameters that represent the Revit object to be instantiated, or the object from which the user wants to receive information. That information is then sent to Revit through a pipe as a 2D array.

On the Revit side, there are several commands available, and based on the function executed on the Grasshopper side, the corresponding command is executed in the BIM tool. These operations might include the creation of new instances of a certain object, the modification of an existing instance, or might simply collect data of created instances, such as the family type or the parameters that define that family.

This software is of particular interest for this thesis because of the connection established with Revit and its ability to use several objects that this BIM tool has available. Nonetheless, not all of its functionalities are available in Lyrebird.

3.2 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 modelling tool but also a full-fledged programming language and a pedagogical tool [12]. This language was supposed to be primarily a textual language but ended up to be a more visual one due to Grasshopper's popularity.

As a programming language, DesignScript is seen as an *associative* language as it mantains 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. In order to be more flexible, it aims to be:

- focused on the end-user because it is meant to be used by programmers of different skill levels.
- multi-paradigm.
- host-independent, making it usable with different back-ends.
- extensible, which gives the programmer freedom to add new functions and classes.

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 user 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 [12].

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 [13]. 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.

This tool is very interesting to this thesis for its ambition to be useful not only to different skilled users, but also for its ability to be accessible for newcomers and flexible enough for them to learn and explore several programming skills. Also, the idea of reproducing the designer's way of thought in the way the program is written is also an objective to achieve as described previously. Yet, the fact that it uses a visual language similar to Grasshopper, makes it suffer from the same scalability problems. A program created in DesignScript can be seen in **Fig. 3**.

3.3 Dynamo

Dynamo is a plug-in for Revit that is strongly influenced by visual programming languages like Grasshopper for Rhino. As a matter of fact, the first version of Dynamo used DesignScript in an attempt to make it a more visual language.

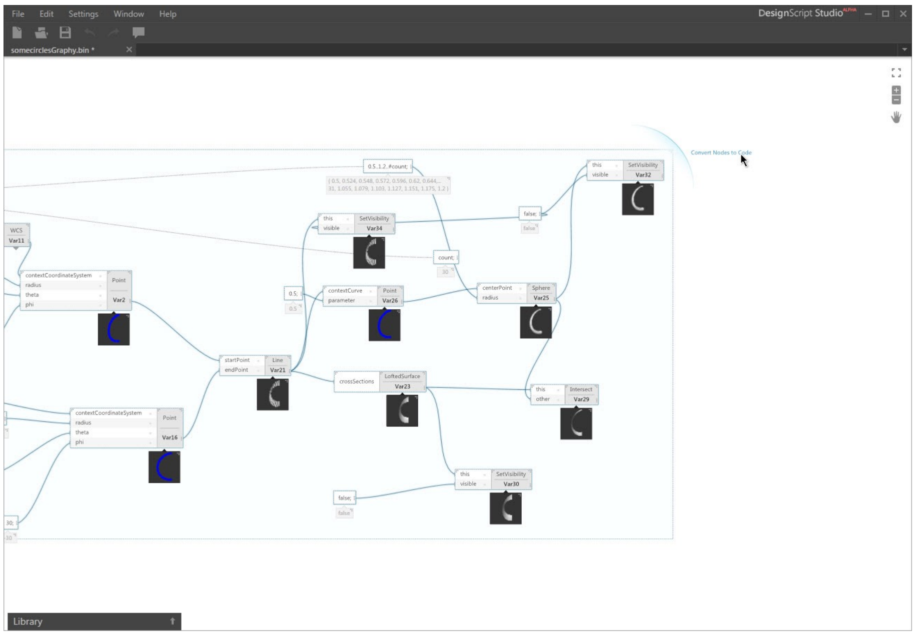


Fig. 3. An example of a program created in DesignScript. Retrieved from http://through-the-interface.typepad.com/.a/6a00d83452464869e20192ac16a8d2970d-pi (retrieved December 2014)

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 [14]. A port from an element can only be connected to another port of a matching type, which means, a port must be connected to a port with an input type that matches its output type.

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

Fig. 4 shows a program written with Dynamo nodes and its result.

Dynamo also allows the usage of code blocks, which are elements that can be used to write programs with a textual programming language, being Python one possible language to use [15]. These code blocks allow the creation of small algorithms that introduce more complex functionalities that are not possible to create with the other nodes.

This tool is another one that communicates with a BIM and presents a way to create GD programs for it, based on a visual programming language.

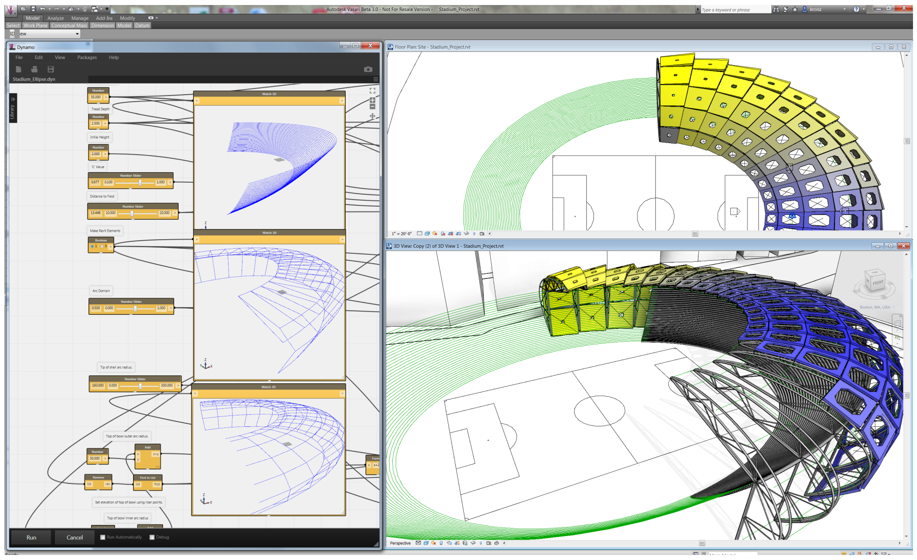


Fig. 4. A program written with Dynamo. Retrieved from http://inthefold.autodesk.com/.a/6a017c3334c51a970b019b01bc21de970c-pi (retrieved December 2014)

3.4 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 several scripts to be defined that include the model description and the parameters of the new object [16].

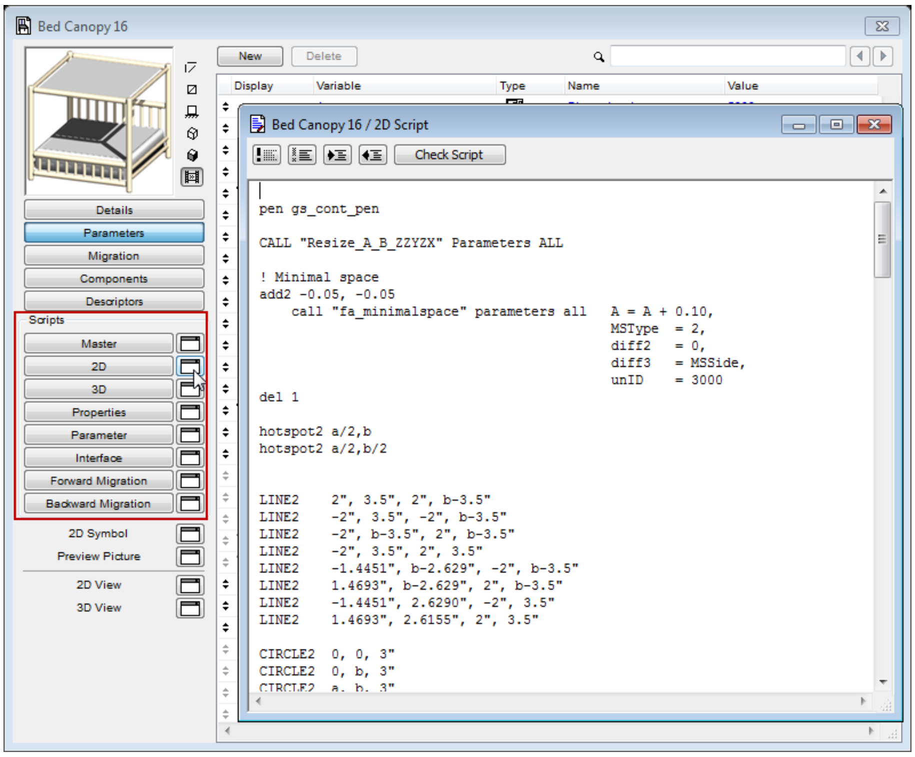
Each object is described with a sequence of commands that describe its geometry. Like OpenGL, a matrix stack implements transformations like translations, rotations and scales. Users use push and pop to introduce or remove a matrix associated with a transformation in the stack. The transformations that are in the stack when creating a shape are the ones that are going to influence it.

Fig. 5 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.

GDL is an interesting point of comparison for this work, since it was a solution introduced in another BIM tool. However, the language, which is similar to BASIC, is obsolete and offers poor abstraction mechanism compared to other languages. Also, GDL does not take advantage of the BIM components of ArchiCAD and produces only simple geometry.

3.5 GenerativeComponents

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



```
pen gs_cont_pen

CALL "Resize_A_B_ZZYZX" Parameters ALL

! Minimal space
add2 -0.05, -0.05
    call "fa_minimalspace" parameters all   A = A + 0.10,
                                            MSType  = 2,
                                            diff2   = 0,
                                            diff3   = MSSide,
                                            unID    = 3000
del 1

hotspot2 a/2,b
hotspot2 a/2,b/2


LINE2    2", 3.5", 2", b-3.5"
LINE2    -2", 3.5", -2", b-3.5"
LINE2    -2", b-3.5", 2", b-3.5"
LINE2    -2", 3.5", 2", 3.5"
LINE2    -1.4451", b-2.629", -2", b-3.5"
LINE2    1.4693", b-2.629", 2", b-3.5"
LINE2    -1.4451", 2.6290", -2", 3.5"
LINE2    1.4693", 2.6155", 2", 3.5"

CIRCLE2  0, 0, 3"
CIRCLE2  0, b, 3"
CIRCLE2  a, b, 3"
```

Fig. 5. A GDL program that creates a bed canopy.

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 [17].

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#. This last one allows the definition of complex algorithms.

GC shows that a Visual Language might be easier to learn but as the user learns and wants to produce more complex models, he will eventually start to produce scripts and eventually will write programs in a textual language.

3.6 RevitPythonShell

RevitPythonShell is a plug-in developed for Revit that allows users to take advantage of the RevitAPI but using Python. This tool was developed by Daren

Thomas to simplify the workflow that is needed in order to develop with the RevitAPI [18].

This plug-in embeds IronPython as language and uses a code editor provided by Python Tools. With this editor, developers have access to a Read-Eval-Print-Loop (REPL) that will let them experiment the API in an easier approach. They only need to type a statement, hit enter, see the results and carry on [18]. This is a great advantage in comparison with the normal workflow of the API when a change is needed. If anything is wrong in an add-on developed with the RevitAPI and a change is needed, the application must be close, the code modified and recompiled.

In addition to this, the fact that it uses Python, a very popular language nowadays, makes it easier to use. However, the functions available in this plug-in are almost equal to the ones available in the RevitAPI and require the user to start and commit a transaction to produce the results in the graphical interface of the BIM tool.

The code needed to create a reference point with the RevitPythonShell is the following:

```
import clr
clr.AddReference('RevitAPI')
clr.AddReference('RevitAPIUI')
from Autodesk.Revit.DB import *

app = __revit__.Application
doc = __revit__.ActiveUIDocument.Document

t = Transaction(doc, 'create␣a␣single␣reference␣point')

t.Start()

#define x, y, and z variables
x = 10
y = 10
z = 0

#declare a XYZ variable
myXYZ = XYZ(x,y,z)

#use XYZ to define a reference point
refPoint = doc.FamilyCreate.NewReferencePoint(myXYZ)

t.Commit()

__window__.Close()
```

The interactivity that the RevitPythonShell gives is a feature that facilitates the interaction with the API and allows more experimental approaches. This is very important when developing a GD program.

3.7 Comparison

Table 1 is a comparison between the tools mentioned above taking into consideration if they allow the usage of visual or textual languages.

	Visual Programming Languages	**Textual Programming Languages**
Lyrebird	Yes	No
DesignScript	Yes	Yes
Dynamo	Yes	Only small scripts
GDL	No	Yes
GC	Yes	Yes
RevitPythonShell	No	Yes

Table 1. Comparison between tools

As seen in the table, there is almost an even distribution between the tools. However, as mentioned before, visual programming languages have their disadvantages, specially when algorithms become very complex. And the tools that support textual languages are either using obsolete languages like BASIC, in the case of GDL, or have complex APIs that create a barrier to architects who are starting to learn how to program.

However there are many advantages to all of these tools that must be taken into consideration. The interactivity and explorable nature of tools like the RevitPythonShell are extremely important for an architect to experiment and learn how to program and how to make use of their creativity. And the fact that the workflow and primitives of tools like Dynamo are easy to learn, motivates architects to start programming.

So taking this information into consideration, the solution to be developed will make use of a textual programming language in order to not suffer from the disadvantages of tools like Grasshopper. But since textual languages present a bigger barrier to newcomers, the primitives and workflow must be as easy to learn as possible. This, in combination with a pedagogical IDE, allows users to learn and explore with ease.

Finally, all these tools can only be used with a specific BIM. The solution that is being developed will not only enable the usage of Revit, but also define an approach that can be explored with other BIM tools as well.

4 Solution

The proposed solution consists of two major modules:

– An Abstraction Layer made of a set of functions written in Racket that will be provided by Rosetta in order to write the GD programs.
– A plug-in that communicates with a BIM tool. In this case, the plug-in is written in C# and uses the RevitAPI in order to produce the desired models in the BIM tool.

4.1 General Architecture

Fig. 6 shows the connection between each of the modules of the solution. In the next sections the purpose of each module will be explained.

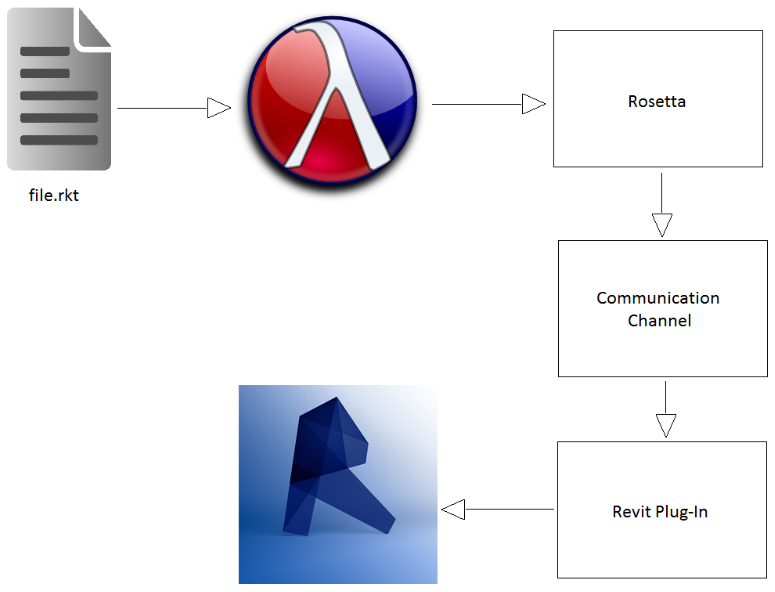


Fig. 6. A general view of the architecture of the solution. The arrows show the flow of information between each module of the solution.

4.1.1 Racket and Rosetta

The program will be written in Racket making use of the functions already defined in Rosetta and new ones specially defined for the new back-end, Autodesk Revit.

At the start of the program execution, a connection is established with the Revit plug-in, creating a communication channel. Afterwards, this channel is used to transmit information between Rosetta and Revit so that Revit's functionality can be accessed from Rosetta.

4.1.2 Communication Channel

Rosetta will send the necessary information through the channel in order to select the desired functionality on the plug-in side.

Right now, this channel is implemented with sockets and the messages are serialized using Google Protocol Buffers. The Protocol Buffers were selected because they are known for their performance and there are implementations available for Racket and for C#, the languages used in Rosetta and in the plug-in. The socket server is initialized when the plug-in is started and the Racket program creates the client and connects to the server when it starts to execute. Information will then be exchanged between them until the user closes the session.

4.1.3 Revit Plug-In

The plug-in will receive the necessary information through the channel in order to create the desired objects or to obtain information about the objects already created. This works like Remote Procedure Call (RPC). The information received is deserialized with the Protocol Buffers in order to correctly reconstruct the information that was sent.

It is important to notice that this plug-in can be non-blocking or blocking. Non-blocking means that the user can still use Revit while the program is executing. This is specially useful if the user wants to change the view while the model is being created to check if it has the desired look.

Blocking, on the other hand, does not allow any type of interaction with the BIM until the program reaches its end.

Both approaches have advantages and disadvantages. The first one allows interaction with Revit while the program is executing but that means a more complex implementation and a loss in performance. The second one is easier to implement and it has better performance but reduces the user's interactivity in the process.

In the current prototype we experimented both approaches and due to the interactivity allowed by the first one we decided to make it the default behaviour. This behaviour is achieved by using the idle function of the RevitAPI. When the user is not using the graphical interface, the idle function will execute and see if new information is in the socket. In case it is, the information will be deserialized and processed in order to invoke the desired functionality.

4.2 Abstraction Layer

The main objective to accomplish is to create an approach that allows users to use GD but taking into account the different paradigm that BIM defines.

This leads to the creation of a set of functions that will not only create geometry but will also take advantage of the functionalities present in the BIM, in this particular case, Revit.

These functions can be divided into two categories: the geometry functions and the BIM functions.

Another objective of this layer is to simplify the functions and reduce the complexity that is introduced by the programming languages available to communicate with a BIM.

4.2.1 Geometry Functions

The creation of complex geometry to use in the models is still a desired feature in BIM. Revit, for example, offers a type of document, called a Family document, in which a user can create a new instance of a specific family of objects, or create a new family for a type of objects that is not provided yet.

For this reason, a set of geometry functions was created in order to produce complex geometry in Family documents. These functions include the creation of cylinders, boxes and spheres, among others.

This set of functions is simply implementing the creation of geometry like in a normal CAD and will enable the production of the same models that were possible to create in other back-ends, such as AutoCAD.

However, it will then be possible to introduce a series of parameters like restrictions and properties (material, price and others) converting the geometry into a proper BIM object. A set of functions to introduce this parameters will be made available and must be used when creating models in order to create objects that can be used in a Revit Project.

Functions that translate and rotate objects and operations like extrusions, revolutions, blends and sweeps are also included in this set.

4.2.2 BIM functions

Another problem that we want to address is the lack of architectural concepts in GD programs written for traditional CAD tools. As an example, in these tools, the concept of levels or floors does not exist, and the walls have the correct height simply because the code has the mathematical values required to make them that way. Similarly, doors and windows have no relation between them and the walls. They can even be created before the wall in which they are placed. These restrictions are present in the code but not in the generated model.

This is where the differences between BIM and traditional CAD tools are noticed. With BIM, all of these relationships should also be present in the generated model. Moreover, levels must be created and floors, ceilings and walls must be associated with specific levels, and doors and windows cannot be created without the indication of which wall is their host. Indeed, a door is created *in* a wall and a wall is placed *on* a specific floor. This is very important considering that in BIM a change to a wall that contains a door must propagate to the door itself and deleting a wall will also delete the door it contains.

To support these relations and restrictions, several functions must be created in our solution that will insert elements into other elements, like a door into a wall. At the present time, most of these functions are still to be defined and the current approach to provide them to the user still needs refinement. However, an example that uses some of these functions is already working and it will be shown in the next section.

4.3 Examples

In this subsection a set of examples will be presented in order to demonstrate what is already working in the solution. The first three examples use code that was originally written in order to produce models in AutoCAD and the result of executing the exact same code in Revit will be shown. The last example will show the result of using functions that make use of BIM objects.

4.3.1 City

In this example, simple geometry objects are used in order to create an abstract city. The code used for this example was created as a simple exercise for the students of Design Programming and Computing at IST. The code is the following:

```
(define (malha-predios p n-ruas m-predios l h s)
  (if (= n-ruas 0)
    #t
    (begin
      (rua-predios p m-predios l h s)
      (malha-predios (+y p (+ l s))
                     (- n-ruas 1)
                     m-predios
                     l
                     h
                     s))))

(define (rua-predios p m-predios l h s)
  (if (= m-predios 0)
    #t
    (begin
      (predio p l h)
      (rua-predios (+x p (+ l s))
                   (- m-predios 1)
                   l
                   h
                   s))))

(define (predio-blocos n p0 c0 l0 h0)
  (if (= n 1)
    (box p0 c0 l0 h0)
    (begin
      (let ((c1 (* (random-range 0.7 1.0) c0))
            (l1 (* (random-range 0.7 1.0) l0))
            (h1 (* (random-range 0.2 0.8) h0)))
        (let ((p1 (+xyz p0
                        (/ (- c0 c1) 2.0)
                        (/ (- l0 l1) 2.0)
                        h1)))
          (box p0 c0 l0 h1)
```

```
         (predio-blocos (- n 1) p1 c1 l1 (- h0 h1)))))))

(define (predio0 p l h)
  (predio-blocos (random-range 1 6)
               p
               l
               l
               (* (random-range 0.1 1.0) h)))

(define (predio1 p l h)
  (cylinder
   (+xy p
       (/ l 2.0)
       (/ l 2.0))
   (/ l 2.0)
   (* (random-range 0.1 1.0) h)))


(define (predio p l h)
  (if (= (random 5) 0)
    (predio1 p l h)
    (predio0 p l h)))

(malha-predios (xyz 0 0 0) 10 10 100 600 40)
```

Cylinders and boxes are used in order to simulate different types of buildings that exist in a real city. In order to give it a more realistic look, the box buildings are made of several boxes of different size and the size of all buildings is random. This example illustrates the `box` and `cylinder` functions.

The result of this program in Revit can be seen in **Fig. 7**.

4.3.2 Sphere of Cylinders

The objective of this example is to produce a sphere made of cylinders. The cylinders are created with several orientations. The code produced for AutoCAD is the following:

```
(define (cilindros-esfera p r rc n)
  (if (= n 0)
    #t
    (let ((phi (random-range pi/4 (- 2pi pi/4)))
          (psi (random-range 0 pi)))
      (cylinderb (+sph p r phi psi)
               rc
               (+sph p
                     r
                     (+ phi (random-range -pi/4 pi/4))
                     (+ psi (random-range -pi/4 pi/4))))
      (cilindros-esfera p r rc (- n 1)))))

```

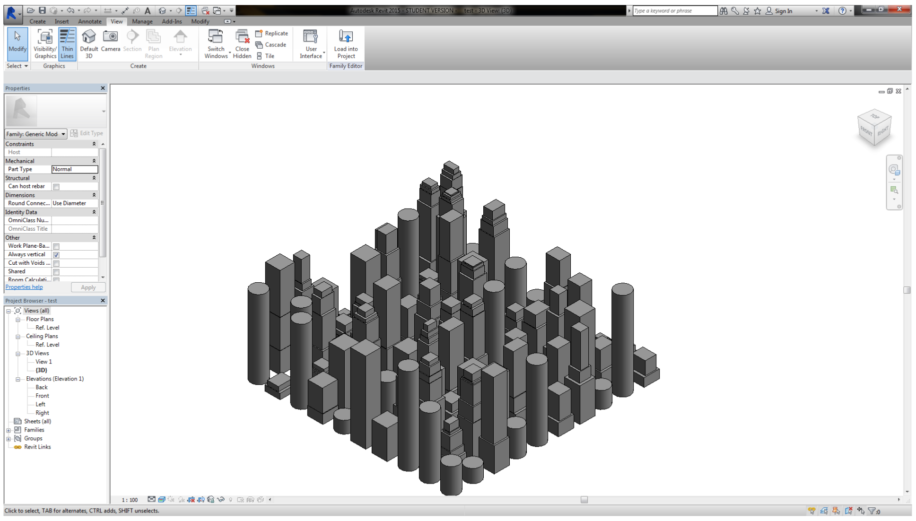


Fig. 7. An example of a city made of boxes and cylinders in Revit

```
(cilindros-esfera (xyz 0 0 0) 3.0 0.1 400)
```

The model obtained from the program execution, but targeting Revit, can be seen in **Fig. 8**.

4.3.3 Trusses

Trusses are models that are relatively hard to design in a CAD tool. However with GD this task is much easier. The following code produces an example of a truss whose nodes are distributed according to a bi-dimensional sinusoid:

```
(define raio-no-trelica (make-parameter 0.1))

(define (no-trelica p)
  (sphere p (raio-no-trelica)))

(define raio-barra-trelica (make-parameter 0.03))

(define (barra-trelica p0 p1)
  (if (=c? p0 p1)
      (empty-shape)
      (cylinder p0 (raio-barra-trelica) p1)))

(define (nos-trelica ps)
  (map no-trelica ps))

(define (barras-trelica ps qs)
  (for/list ((p (in-list ps))
```

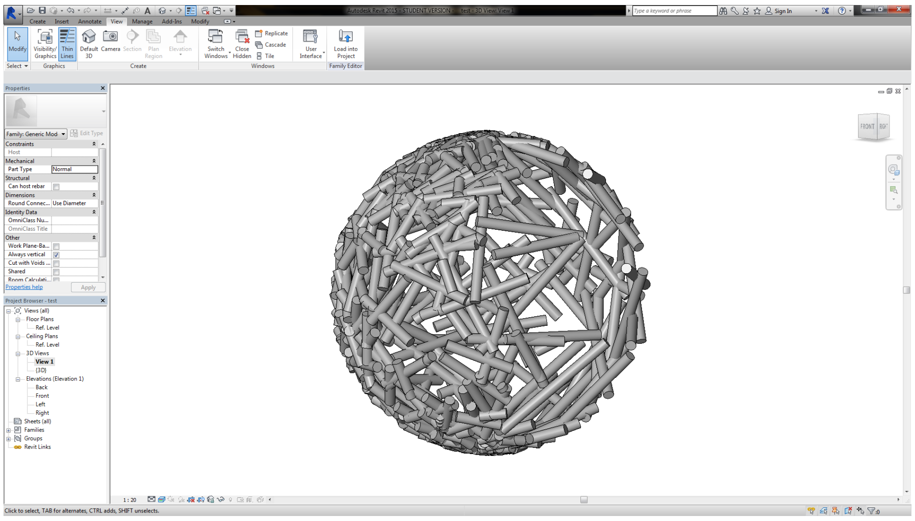


Fig. 8. Geometric model in Revit with cylinders placed in order to form a sphere

```
            (q (in-list qs)))
    (barra-trelica p q)))

(define (trelica as bs cs)
  (nos-trelica as)
  (nos-trelica bs)
  (nos-trelica cs)
  (barras-trelica as cs)
  (barras-trelica bs as)
  (barras-trelica bs cs)
  (barras-trelica bs (cdr as))
  (barras-trelica bs (cdr cs))
  (barras-trelica (cdr as) as)
  (barras-trelica (cdr cs) cs)
  (barras-trelica (cdr bs) bs))

(define (trelica-espacial curvas)
  (let ((as (car curvas))
        (bs (cadr curvas))
        (cs (caddr curvas)))
    (nos-trelica as)
    (nos-trelica bs)
    (barras-trelica as cs)
    (barras-trelica bs as)
    (barras-trelica bs cs)
    (barras-trelica bs (cdr as))
    (barras-trelica bs (cdr cs))
    (barras-trelica (cdr as) as)
    (barras-trelica (cdr bs) bs)
    (if (null? (cdddr curvas))
        (begin
          (nos-trelica cs)
          (barras-trelica (cdr cs) cs))
        (begin
          (barras-trelica bs (cadddr curvas))
          (trelica-espacial (cddr curvas))))))

(define (render-trelica malha)
  (let* ((p0 (caar malha))
         (p1 (caadr malha))
         (p2 (cadadr malha))
         (p3 (cadar malha))
         (d (min (distance p0 p1) (distance p0 p3))))
    (parameterize ((raio-no-trelica (/ d 9.0))
                   (raio-barra-trelica (/ d 19.0)))
      (trelica-espacial
       (insere-vertice-piramide
        malha)))))

(define (sin-u*v n)
  (map-division
    (lambda (u v)
      (xyz u
           v
           (* 0.4 (sin (* u v)))))
    (* -1 pi) (* 1 pi) n
    (* -1 pi) (* 1 pi) n))

(render-trelica (sin-u*v 10))
```

The nodes are spheres and the bars uniting them are cylinders whose base and top are the center of the two nodes that they unite. The result is seen in **Fig. 9**.

4.3.4 Floor Example

This last example does not include simple geometry and uses features that are only available in a BIM tool. The most relevant one is the fact that it uses BIM objects instead of simple geometry. By using these objects, the code is simplified because users do not need to implement all of the details of a door like in a CAD tool. The selected door family has all the details and properties that a door needs.

The code used to produce the model is the following:

```

(define floor-points
  (list (xyz -10 20 0)
```

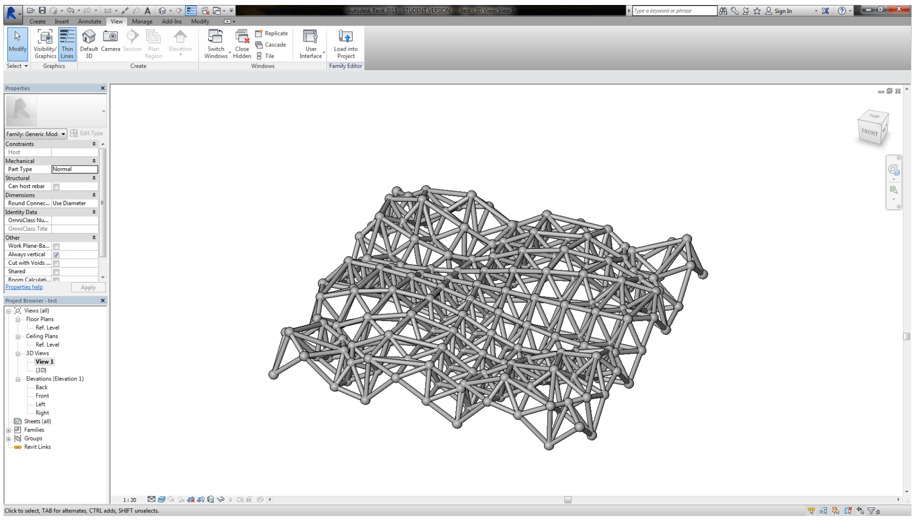


Fig. 9. A Truss made of cylinders and spheres

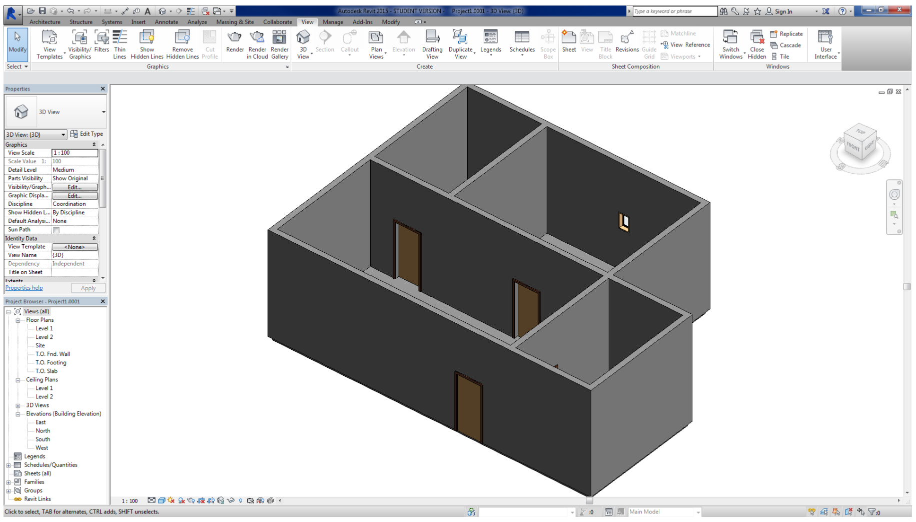
```
        (xyz 10 20 0)
        (xyz 10 5 0)
        (xyz 20 5 0)
        (xyz 20 -10 0)
        (xyz -20 -10 0)
        (xyz -20 20 0)
        (xyz -10 20 0)))

(define walls-ids (list))

(define (floor-walls points level)
  (for/list ((pt0 points)
             (pt1 (cdr points)))
    (set! walls-ids (append walls-ids (list (wall pt0 pt1 level))))))

(define f1 (floor-from-points floor-points "Level␣1"))
(floor-walls floor-points "Level␣1")
(insert-door-relative (car (cddddr walls-ids)) 15 0)
(define w1 (wall (xyz -10 5 0) (xyz 10 5 0) "Level␣1"))
(insert-door-relative w1 10 0)
(define w2 (wall (xyz -10 5 0) (xyz -10 20 0) "Level␣1"))
(define w3 (wall (xyz -20 5 0) (xyz -10 5 0) "Level␣1"))
(insert-door-relative w3 5 0)
(define w4 (wall (xyz 10 5 0) (xyz 10 -10 0) "Level␣1"))
(insert-door-relative w4 10 0)
(insert-window (car walls-ids) 10 5)
```

In this example, all the objects use their standard family but it is possible to choose other ones with different properties that fit different situations. Also, in the case of walls, BIM automatically joins them since they are objects with that property. The result can be seen in **Fig. 10**.



3D View: {3D}	Edit Type
Graphics	
View Scale	1 : 100
Scale Value 1:	100
Detail Level	Medium
Parts Visibility	Show Original
Visibility/Graph...	Edit...
Graphic Displa...	Edit...
Discipline	Coordination
Show Hidden L...	By Discipline
Default Analysi...	None
Sun Path	☐
Identity Data	
View Template	<None>
View Name	{3D}
Dependency	Independent
Title on Sheet	

Fig. 10. A 3D view of a floor created in Revit

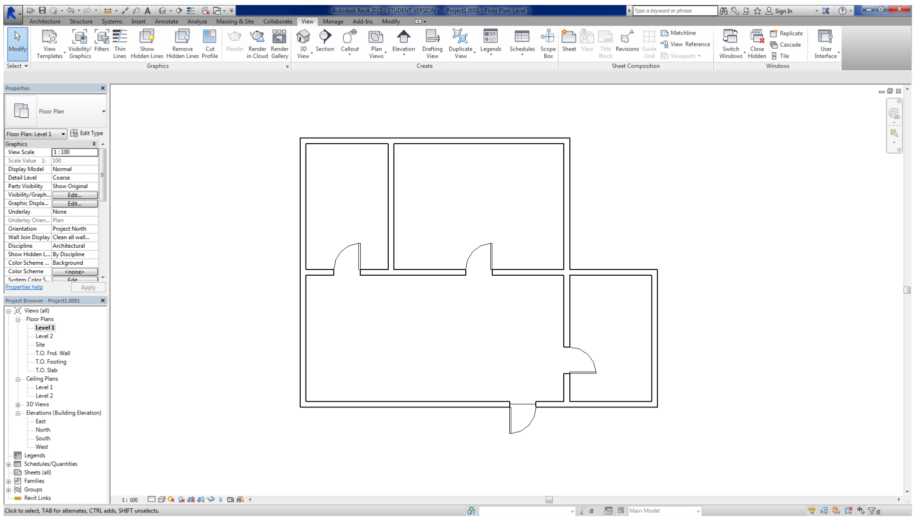
This example has all the properties of a model created with the graphical interface of the BIM tool. The advantage of this model is that it can fully explore the capabilities of the BIM tool, namely, it can be used to compute the needed materials and it can be used for structural analysis. Also, plan views of the model can be automatically generated, as visible in **Fig. 11**.

5 Evaluation

The solution will be evaluated taking into consideration (1) its capability for creating the complete desired model, and (2) the correctness of the model, which means, if the model has the properties expected of a proper BIM model.

To do this, models that are possible to create in CAD tools must be produced in Revit, as well as models that explore features of the BIM tool. Several BIM experts will help us in the creation of these models by showing us how they would create them with a manual approach. In the end, their solution will be compared with a solution that creates the same model but using a programming approach.

Additionally, the time needed to create and modify the model in both the manual and the programming approaches will also be taken into account.



Graphics	
View Scale	1 : 100
Scale Value 1:	100
Display Model	Normal
Detail Level	Coarse
Parts Visibility	Show Original
Visibility/Graph...	Edit...
Graphic Displa...	Edit...
Underlay	None
Underlay Orien...	Plan
Orientation	Project North
Wall Join Display	Clean all wall...
Discipline	Architectural
Show Hidden L...	By Discipline
Color Scheme ...	Background
Color Scheme	<none>
System Color S...	Edit

Floor Plan: Level 1

Click to select, TAB for alternates, CTRL adds, SHIFT unselects.

Fig. 11. Representation of the floor in plan view

6 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 are implementing takes into consideration generic BIM concepts and not the concepts that are exclusive to a given tool, we expect that users will be able to create portable programs that explore different BIM tools.

The solution will continue to be developed and tested with architects that are also novice programmers. Their feedback will be used to define and improve the abstractions and functions needed to work with a BIM tool.

References

1. Fernandes, R. (2013). Generative Design: a new stage in the design process. Master thesis, Instituto Superior Técnico.
2. Azhar, S., Hein, M. and Sketo, B.(2008). Building Information Modeling (BIM): Benefits, Risks and Challenges. In *Proceedings of the 44th ASC National Conference*, Auburn, Alabama, USA.
3. Eastman, C., Teicholz, P., Sacks, R., & Liston, K. (2008). *BIM Handbook: A Guide to Building Information Modeling for Owners, Managers, Designers, Engineers, and Contractors*, Hoboken, New Jersey: John Wiley & Sons, Inc.

4. Ibrahim, M., Krawczyk, R., & Schipporeit, G. (2004). Two Approaches to BIM: A Comparative Study. In *eCAADe Conference* (volume 22, pp. 610-616).
5. McCormack, J., Dorin, A., & Innocent, T. (2004). Generative Desing: a paradigm for design research. In *Proceedings of Futureground, Design Research Society*, Melbourne.
6. Terdizis, K. (2003). Expressive Form: A conceptual Approach to Computational Design. London and New York: Spon Press.
7. Leitão, A., Santos, L., & Fernandes, R.. Pushing the Envelope: Stretching the limits of Generative Design.
8. Lopes, J., & Leitão, A. (2011). Portable Generative Design for CAD Applications. In *Proceedings of the 31st annual conference of the Association for Computer Aided Design in Architecture* (pp. 196-203).
9. Fernando, R., Drogemuller, R., & Burden, A. (2012) *Parametric and Generative Methods with Building Information Modelling: Connecting BIM with explorative design modelling* In *Beyond Codes and Pixels: CAADRIA 2012: Proceedings of the 17th International Conference on Computer Aided Architectural Design Research in Asia* (pp. 537-546).
10. Payne, A., & Issa, R. (2009) *The Grasshopper Primer, Second Edition - for version 0.6.0007*, `http://www.liftarchitects.com/blog/2009/3/25/grasshopper-primer-english-edition` (retrieved on August 2014)
11. Logan, T. (2014, February) Superb Lyrebird. `http://lmnts.lmnarchitects.com/bim/superb-lyrebird/` (retrieved on November 2014)
12. Aish, R. (2012) DesignScript: origins, explanation, illustration. In *Computational Design Modelling* (pp. 1-8). Springer Berlin Heidelberg.
13. Aish, R. (2013) DesignScript: Scalable Tools for Design Computation. In *eCAADe 2013: Computation and Per- formance–Proceedings of the 31st International Conference on Education and research in Computer Aided Architectural Design* (pp. 18 - 20)
14. Dynamo Source Code `https://github.com/DynamoDS/Dynamo` (retrieved on November 2014)
15. Learn Dynamo `http://dynamobim.com/learn/` (retrieved on December 2014)
16. Nicholson-Cole, D. (2004) *Introduction to Object Making with Archicad: GDL for Beginners* Graphisoft R&D Rt
17. Aish, R., & Woodbury, R. (2005) *Multi-level interaction in parametric design* In *Smart Graphics* (pp. 151-162), Springer Berlin Heidelberg.
18. Thomas, D. (2009) *Introducing RevitPythonShell* `http://darenatwork.blogspot.pt/2009/12/introducing-revitpythonshell.html` (retrieved on April 2015)

A Planning

Planning		
Tasks	**Details**	**Duration**
Related Work Research	- Conclude the research	July (3 weeks)
Attribute introduction for Family Documents	- Study the API and tutorials - Implementation - Testing	July (1 week) August (4 weeks) September (2 week)
BIM functions for the abstraction layer	- Study the API and tutorials - Implementation - Testing	September (2 weeks) October (4 weeks) November (2 weeks)
Implementation of the case study	- Definition of the case study with a BIM expert - Implementation - Testing - Review the results with the BIM expert	November (2 week) December (4 weeks) January (2 weeks) January (2 weeks)
Evaluation	- Evaluate the solution with the case study results - Perform other tests needed for evaluation	February (1 week) February (3 weeks)
Thesis final report	- Write thesis report	March (4 weeks) April (2 weeks)
Review and Submission	- Report review and submission	April - May (3 week)

Table 2. Planning schedule