Linguagem de Programação para Modelação Geométrica

Gualter Magnusson Gonçalves Semedo
gualter.semedo@tecnico.ulisboa.pt

Instituto Superior Técnico
Universidade de Lisboa

Resumo As ferramentas para Desenho Assistido por Computador (CAD) fornecem Interfaces Gráficas do Utilizador (GUI) que permitem aos utilizadores gerarem modelos geométricos. No entanto, dada a baixa flexibilidade e precisão das GUI, muitos são os que recorrem a programação para gerar modelos geométricos, ou seja, ao Desenho Generativo. Actualmente quem recorre ao Desenho Generativo para gerar modelos geométricos tem de lidar com linguagens de programação e plataformas de desenvolvimento criadas para propósito geral e pensadas para serem utilizadas por programadores profissionais, linguagens de difícil aprendizagem e utilização e que não fornecem mecanismos de suporte ao Desenho Generativo. Python, JavaScript e AutoLisp são exemplos de algumas dessas linguagens. Este trabalho define uma linguagem que programação e um conjunto de operadores geométricos que devem ser implementados de forma a solucionar estes problemas.

1 Introdução

Com o intuito de superar as limitações e a falta de expressividade dos ambientes gráficos das ferramentas de Desenho Assistido por Computador (CAD), muitos são os que recorrem à programação para gerar modelos geométricos (Desenho Generativo). A construção de modelos geométricos através da escrita de programas é uma actividade delicada e complexa, tendo por isso surgido ao longo dos anos linguagens de programação cujo objectivo é facilitar esta tarefa.

Neste trabalho analisa-se as várias linguagens criadas e/ou utilizadas para modelação geométrica e os problemas que os seus utilizadores/programadores enfrentam. A falta de ambientes de desenvolvimento e linguagens de programação que sejam: de fácil aprendizagem e utilização, expressivas, e que estejam adaptadas às necessidades do domínio, são os principais problemas enfrentados.

Como solução a estes problemas, define-se neste trabalho uma linguagem de programação e um conjunto de operadores geométricos que devem ser implementados pelos ambientes de desenvolvimento de modo a facilitar a descrição de modelos geométricos através da escrita de programas, bem como resolver os problemas subjacentes.

Para a avaliação deste trabalho utilizar-se-á a plataforma Rosetta (Lopes, 2012). A plataforma será estendida para incluir a nova linguagem e os operadores geométricos propostos. O resultado será comparado com a versão original do Rosetta, bem como, com outras linguagens e plataformas desenvolvidas e/ou utilizadas no contexto do Desenho Generativo.

No que diz respeito à organização do resto deste documento, na secção 2 apresentam-se os conceitos fundamentais, na secção 3 descrevem-se os objectivos deste trabalho, na secção 4 apresenta-se uma análise das linguagens de programação criadas e/ou utilizadas no contexto do Desenho Generativo, na secção 5 apresenta-se a *Proposta de Tese*, na secção 6 descreve-se o processo a ser seguido para validar a *Proposta de Tese* e na secção 7 apresenta-se um resumo e as principais conclusões deste trabalho.

2 Conceitos Fundamentais

Nesta secção apresentam-se alguns conceitos do domínio em que este trabalho se insere, ou seja, arquitectura, design e linguagens de programação, conceitos esses que serão utilizados ao longo deste relatório.

2.1 Desenho Generativo

Desenho Generativo (GD) é a aplicação de métodos computacionais para desenhar estruturas ou objectos arquitecturais (Krause, 2003). Por outras palavras, Desenho Generativo permite a um designer escrever programas que quando executados produzem modelos geométricos.

2.2 Desenho Assistido por Computador

Desenho Assistido por Computador (CAD) é o nome dado a um sistema computacional (software) utilizado na criação e/ou manipulação de modelos arquitecturais.

2.3 Linguagem Específica do Domínio

Linguagem Específica do Domínio é uma linguagem de programação que fornece notação adaptada para um domínio de aplicação e que se baseia nos conceitos e características relevantes desse domínio (Arie Deursen, 2002).

3 Objectivos do Trabalho

O objectivo deste trabalho consiste em analisar as linguagens de programação e plataformas para Desenho Generativo de modo a apresentar uma solução para os problemas que enfrenta quem recorre ao Desenho Generativo para criar modelos geométricos, bem como um plano de validação da proposta apresentada.

4 Trabalhos Relacionados

Ao longo dos anos algumas linguagens de programação e ferramentas foram desenvolvidas para dar suporte ao Desenho Generativo, sendo ainda algumas linguagens e ferramentas de propósito geral adoptadas para este domínio. Nesta secção apresenta-se uma análise de algumas dessas linguagens. A análise dá ênfase aos operadores geométricos fornecidos, apresentando para cada linguagem uma análise critica da mesma.

As linguagens que serão analisadas são: PLaSM, GML, GDL e Dynamo (linguagens específicas do domínio), MEL, AutoLisp, RhinoScript (Linguagens de propósito geral utilizados para dar suporte ao Desenho Generativo nas ferramentas de CAD Maya, AutoCad, Rhrinoceros3D, respectivamente) e Processing (Linguagem de propósito geral utilizado em desenho electrónico e artes gráficas). No final desta secção apresenta-se uma análise da ferramenta Rosetta (Lopes, 2012).

A análise apresentada nesta secção excluiu as linguagens de programação visuais, uma vez que trabalhos recentes demonstraram a falta de flexibilidade deste tipo de linguagens, bem como a impossibilidade de tirar proveito de elementos de extrema importância para o Desenho Generativo, como é o caso da recursão e das funções de ordem superior (Leitão, 2014),(Lopes, 2012) e (Cabecinhas, 2010).

4.1 PLaSM

PLaSM (de Programming Language for Solid Modeling) (Paoluzzi, 2003) é uma linguagem para Desenho Generativo, criada pelo grupo de CAD das universidades Italianas “La Sapienza” e “Roma Tre”. A linguagem é considerada (pelo seu autor) uma extensão orientada à Geometria de um subconjunto da linguagem FL (de Function Level), uma linguagem criada pelo grupo de Programação Funcional da divisão de pesquisa da IBM em Almaden, Estados Unidos (John Backus, 1989) e (John Backus, 1990).

A linguagem foi desenvolvida tendo em vista a seguinte assumpção: “Ambiente de computação funcional é o ambiente natural para a computação geométrica, bem como, para a geração de modelos e formas geométricas”. Essa assumpção justifica-se pelas seguintes propriedades da programação funcional (Paoluzzi, 2003):

- As funções podem ser utilizadas como programas ou como dados;
- Os programas são facilmente conectados por concatenação e/ou nidificação;
- O código dos programas é conciso e claro;
- O comportamento dos programas é de fácil compreensão uma vez que os programas não guardam estado.

4.1.1 Operadores Geométricos da Linguagem

A linguagem PLaSM dispõe de um conjunto de operadores geométricos predefinidos. A maioria dos operadores são independentes da dimensão, ou seja, podem ser aplicados a objectos de qualquer dimensão (1D, 2D, 3D, ou mesmo superior).

4.1.1.1 CUBOID

A primitiva *CUBOID* permite a construção de segmentos de recta (1D), rectângulos (2D), paralelepípedos (3D) e hiper-paralelepípedos (4D ou mais). Exemplos:

```
1 CUBOID:2 ≡ segmento de recta (1D) de comprimento 2
2 CUBOID:<2,4> ≡ retângulo (2D) de área 2 x 4
3 CUBOID:<1,2,3> ≡ paralelepípedo (3D) de volume 1 x 2 x 3
4 CUBOID:<1,1,1,1> ≡ hiper-paralelepípedo (4D) de volume 1
```

4.1.1.2 SIMPLEX

A primitiva *SIMPLEX* permite a construção de segmentos de recta (1D), triângulos (2D), tetraedros (3D), etc. Exemplos:

```
1 SIMPLEX:2 ≡ segmento de recta (1D) de comprimento 2
2 SIMPLEX:<1,1> ≡ triângulo (2D) de área 1/2!
3 SIMPLEX:<1,g,1> ≡ tetraedro (3D) de volume g/3!
```

4.1.1.3 QUOTE

A primitiva *QUOTE* permite a construção de polígonos a partir de uma sequência de números positivos e negativos. Os números positivos são utilizados para criar segmentos e os negativos são utilizados para criar intervalos vazios. Exemplo:

```
1 QUOTE:<5,-3,5,-2,5,-1,5>
```

Script 4.1.1: Exemplo de utilização da primitiva QUOTE.

Figura 4.1.1: Polígono resultante da execução do script 4.1.1.

4.1.1.4 MKPOL

MKPOL (de MaKe POLyhedron) é o construtor geométrico básico da linguagem PLaSM, permitindo a construção de poliedros [1] complexos de qualquer dimensão. A assinatura da função é:

$$MKPOL :< verts, cells, pols >$$

Onde: *verts* é uma sequência de pontos que representa os vértices do poliedro, *cells* indica a sequência dos pontos que formam as arestas e *pols* indica a sequência das arestas que formam o poliedro.
Exemplo:

```
1 DEF Lshape = MKPOL<verts, cells, pols>
2 WHERE
3   verts = <<0,0>, <2,0>, <2,1>, <1,1>, <1,2>, <0,2>>,
4   cells = <<1,2,3,4>, <4,5,6,1>>,
5   pols = <1,2>
6 END;
```

Script 4.1.2: Exemplo de utilização da primitiva MKPOL.

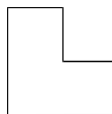


Figura 4.1.2: Forma geométrica *Lshape* resultante da execução do script 4.1.2.

4.1.1.5 UKPOL

UKPOL é a operação inversa da operação *MKPOL*, ou seja, permite, dado um poliedro qualquer, obter a representação externa da estrutura de dados do poliedro. Exemplo:

```
1 UKPOL: (CUBOID:<1,2,3>) ≡ <
2   < <0.0,2.0,3.0>,<1.0,2.0,3.0>,<0.0,0.0,3.0>,<1.0,0.0,3.0>,
3     <0.0,2.0,0.0>,<1.0,2.0,0.0>,<0.0,0.0,0.0>,<1.0,0.0,0.0> >,
4   < <1,2,3,4,5,6,7,8> >,
```

[1] http://en.wikipedia.org/wiki/Polyhedron

```
5   < <1> > >
```

4.1.1.6 STRUCT

A primitiva *STRUCT* permite a criação de formas geométricas complexas a partir da assemblagem de formas geométricas mais simples. Exemplo:

```
1 DEF Leg = CUBOID:<0.1,0.1,0.7>;
2 DEF Plane = CUBOID:<1,1,0.2>;
3 DEF Table = STRUCT:<Leg, T:1:0.9:Leg, T:<1,2>:<0.9,0.9>:Leg, T:2:0.9:Leg, T:3:0.7:Plane>;
```

Script 4.1.3: Exemplo de utilização da primitiva STRUCT.

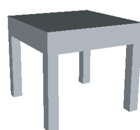


Figura 4.1.3: Forma geométrica *Table* resultante da execução do script 4.1.3.

4.1.1.7 SKELETON

O operador *SKELETON* permite extrair o "esqueleto" de dimensão d de uma certa forma geométrica de uma dimensão D (d < D), ou seja, a operação permite obter as células das fronteiras de uma forma geométrica e cujas dimensões sejam menor ou igual a d. A operação é definida por: *@d:form*, onde d corresponde à dimensão do esqueleto que queremos extrair e *form* corresponde a forma geométrica original. Por exemplo: *@0:f* retorna os vértices da forma geométrica *f*, *@1:f* retorna os vértices e as arestas da forma geométrica *f*, *@2:f* retorna os vértices, as arestas e as faces da forma geométrica *f*. Na figura 4.1.4 apresenta-se a forma geométrica resultante da extracção dos vértices e arestas (*@1*) da forma geométrica *Cub* definida no script 4.1.4.

```
1 DEF Cub = CUBOID:<1,1,1> STRUCT SIMPLEX:3
2 DEF Squeleton = @1:Cub
```

Script 4.1.4: Exemplo de utilização do "Squeleton".

4.1.1.8 Produto

O produto (representado pelo símbolo *) é o operador da linguagem PLaSM que permite criar uma nova forma geométrica a partir de duas formas geométricas originais, onde, cada ponto da nova forma geométrica é determinado pelo produto cartesiano dos pontos das formas geométricas originais. No script 4.1.5 apresenta-se um exemplo de utilização do operador:

```
1 QUOTE<10,-10,10>
2 QUOTE<10,-10,10> * QUOTE<10,-10,10>
3 QUOTE<10,-10,10> * QUOTE<10,-10,10> * QUOTE<3>
```

Script 4.1.5: Exemplo de utilização do Produto.

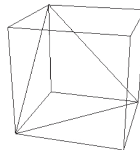


Figura 4.1.4: Forma geométrica *Squeleton* definida no script 4.1.4 como resultado da extração do "esqueleto" 1D da forma geométrica *Cub*.

Figura 4.1.5: Formas geométricas resultantes da execução do script 4.1.5.

4.1.1.9 Transformações Geométricas

A linguagem PLaSM define as seguintes transformações geométricas:

$$T : coords : params \equiv \text{Translação}$$
$$R : coords : params \equiv \text{Rotação}$$
$$S : coords : params \equiv \text{Escala}$$

Onde *coords* corresponde as coordenadas afectadas pela transformação e *params* aos parâmetros dessa transformação. Exemplos:

```
1 T:1:4.5  ≡ Translação de 4.5 unidades sobre a 1ª coordenada (xx)
2 R:2:(PI/4)  ≡ Rotação de 45º sobre a 2ª coordenada (yy)
3 T:<2,3>:<2,-2>  ≡ Translação de 2 e -2 unidades sobre a 2ª e a 3ª coordenada, respectivamente
```

4.1.1.10 Operadores Booleanos

A linguagem fornece os seguintes operadores booleanos para a modelação e/ou manipulação de formas geométricas: união (+), intersecção (&), diferença (-) e diferença exclusiva (∧). Exemplo:

```
1 DEF a = T:<1,2>:<-0.5,-0.5>:CUBOID<1,1,1>;
2 DEF b = R:<1,2>:(PI/4):a;
3 STRUCT<a + b,T:1:2,a & b,T:1:2,a ∧ b,T:1:2,a - b>;
```

Script 4.1.6: Exemplo de utilização dos operadores booleanos.

4.1.1.11 Operadores de Posicionamento Relativo

Os operadores *ALIGN* (aplicável a objectos de qualquer dimensão), *TOP*, *BOTTOM*, *LEFT*, *RIGHT*, *UP*, *DOWN* (aplicáveis apenas a objectos 3D), permitem o posicionamento relativo de objectos.

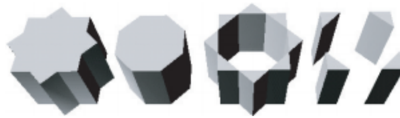


Figura 4.1.6: Formas resultantes da execução do script 4.1.6. Correspondendo a: união, intersecção, diferença exclusiva e diferença, respectivamente.

Na figura 4.1.7 apresenta-se o resultado da exportação para um ambiente gráfico do objecto *out* definido no script 4.1.7. A mesa apresentada no exemplo foi construída utilizando as operações: *QUOTE* (secção 4.1.1.3), *** (secção 4.1.1.8) e *TOP*. As cadeiras foram construídas aplicando a operação *eScala* (secção 4.1.1.9) sobre a mesa. O output apresentado na figura 4.1.7 corresponde ao posicionamento clássico das cadeiras em torno de uma mesa e foi conseguido recorrendo aos operadores de posicionamento relativo.

```
1 DEF legs = QUOTE:<0.1,-0.8,0.1> * QUOTE:<0.1,-0.8,0.1> * QUOTE:<0.7>;
2 DEF plane = QUOTE:<1> * QUOTE:<1> * QUOTE:<0.2>;
3 DEF table = Legs TOP Plane;
4 DEF chair = S:<1,2,3>:<0.4,0.4,0.5>:Table;
5 DEF out = chair RIGHT table RIGHT chair UP chair DOWN chair;
```

Script 4.1.7: Exemplo de utilização dos operadores de posicionamento relativo.

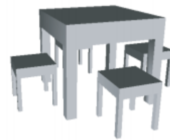


Figura 4.1.7: Forma geométrica correspondente ao objecto *out* definido em 4.1.7.

4.1.2 Conclusões

A linguagem PLaSM descrita pelos seus criadores como sendo uma "extensão geométrica" de um subconjunto da linguagem FL (de Function Level), ou seja, para a programação funcional, foi desenhada com a assunpção que o ambiente funcional é o ambiente natural para o Desenho Generativo. A linguagem permite a combinação de funções para a construção de outras funções, ou seja, funções de ordem superior, a composição de funções e a utilização de funções parciais (Currying [2], em inglês).

No entanto, apesar da sua grande expressividade (como pode perceber-se pelo exemplo 4.1.7), a sua sintaxe e semântica são complicadas, mesmo para pessoas com boa experiência em programação e bons conhecimento de outras linguagens e paradigmas de programação (como se pode concluir pelo exemplo 4.1.2).

No entanto, a linguagem contém um conjunto de operadores geométricos (secção 4.1.1) muito poderosos, correspondendo assim a uma fonte de inspiração quando se trata de criar uma nova linguagem de programação para Desenho Generativo.

[2] `http://en.wikipedia.org/wiki/Currying`

4.2 GML

GML (de Generative Modeling Language) (Havemann, 2003) é uma linguagem de programação de baixo nível criada para Desenho Generativo de objectos 3D. A linguagem é baseada na linguagem PostScript (Adobe, 1999) da Adobe[3] e encontra-se implementada como um interpretador "stack-based" (baseado em pilha). A pilha é uma pilha de operandos, ou seja, a pilha é utilizada para passar argumentos entre funções.

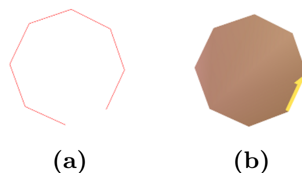
A linguagem GML (tal como a linguagem PLaSM) não se encontra integrada com nenhuma ferramenta de CAD, utilizando por isso a biblioteca OpenGL[4] para o processo de renderização.

A linguagem utiliza malhas poligonais (James Foley, 1982) e operadores de Euler (Charles Eastman, 1979) para guardar e manipular formas geométricas. A complexidade desses mecanismos de implementação é encapsulada pelos operadores geométricos da linguagem.

4.2.1 Operadores Geométricos da Linguagem [5]

4.2.1.1 Poly2doubleface

O operador *poly2doubleface* permite converter um polígono representado por um conjunto de pontos 3D numa figura geométrica de duas faces. Na figura 4.2.1 apresenta-se: (a) um polígono (b) o resultado da aplicação do operador *poly2doubleface* a esse polígono.



(a) **(b)**

Figura 4.2.1: Exemplo de aplicação do operador *poly2doubleface*.

4.2.1.2 Bridgerings

O operador *bridgerings* permite, dadas duas faces de um sólido ou dois sólidos, criar faces que unem as arestas das faces do(s) sólido(s) correspondentes.

Na figura 4.2.2 apresenta-se: (a) duas faces com o mesmo número de vértices e (b) a figura geométrica resultante da aplicação da operação *bridgerings* às faces.

4.2.1.3 Extrude

O operador *extrude* é o operador que, dado um polígono (ou forma geométrica) de perfil e uma curva guia, efectua uma "varredura" ao longo da curva utilizando o polígono (ou forma geométrica), resultando essa "varredura" num novo polígono (ou forma geométrica).

[3] `http://www.adobe.com/`

[4] `https://www.opengl.org/`

[5] Os exemplos apresentados nesta secção foram adaptados a partir de (Havemann, 2003).

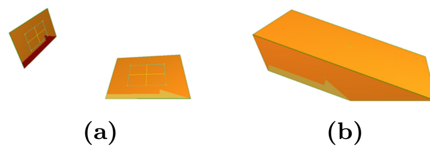


Figura 4.2.2: Exemplo de aplicação do operador *bridgerings*.

Na figura 4.2.3 apresenta-se: (a) uma forma geométrica de perfil (b) uma curva guia e (c) a forma geométrica resultante da aplicação da operação *extrude* a forma de perfil e a curva guia, ou seja, a forma geométrica resultante de uma varredura ao longo da curva utilizando a forma de perfil.

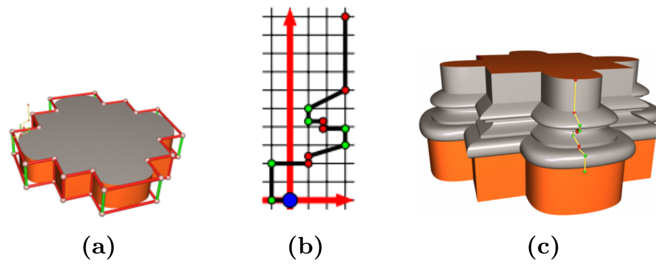


Figura 4.2.3: Exemplo de uma aplicação do operador *extrude*.

4.2.2 Exemplos Arquitecturais [6]

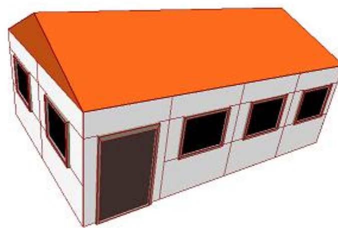


Figura 4.2.4: Forma geométrica resultante da execução do script 4.2.1.

```
1 deleteallmacros
2 newmacro clear
3 pnm-clear
4 bns-clear
5 usereg ioremoveall
6 dict !myRules
```

[6] Os exemplos apresentados nesta secção foram adaptados a partir de (Havemann, 2003).

```
7  :myRules begin
8  /House [[/mass /S (30,20,10)] [/shape "cube"] [/comp "all" [/Front [3 /Side] /Wall /Roof]]] def
9  /Side [[/repeat "X" 7 /WTile]] def
10 /Front [[/subdiv "X" [7 [1]] [/DTile /Side]]] def
11 /WTile [[/subdiv "Z" [[2] 4 [1]] [/Wall /WTile2 /Wall]]] def
12 /WTile2 [[/subdiv "X" [[1] 5 [1]] [/Wall /Window /Wall]]] def
13 /DTile [[/subdiv "Z" [8 [1]] [/DTile2 /Wall]]] def
14 /DTile2 [[/subdiv "X" [[1] 5 [1]] [/Wall /Door /Wall]]] def
15 %%Terminal Symbols
16 /Wall [[/object "Wall"]] def
17 /Roof [[/mass /S "Z" 4][/object "Roof_Gabled"]] def
18 /Door [[/object "Door"]] def
19 /Window [[/object "Window"]] def
20 end
21 %%Execute
22 [/House] [:myRules] Daidalos_SG.SGModel
```

Script 4.2.1: Construção generativa da estrutura de uma casa e das suas fachadas.

4.2.3 Conclusões

Tendo em vista os exemplos apresentados na secção 4.2.1, pode-se concluir que a linguagem GML é muito expressiva, oferecendo aos programadores operadores que permitem uma fácil modelação de formas geométricas que de outra forma seria muito complicada ou mesmo impossível.

No entanto, a linguagem é baseada em PostScript, uma linguagem baseada em pilha que foi originalmente criada para representar a formatação de dados para impressão e não para ser utilizada por programadores. GML tal como o seu predecessor é baseada em pilha e fornece pouca abstracção aos programadores, facto que torna a sua aprendizagem complexa e os programas de difícil compreensão (tal como se pode verificar no código apresentado em 4.2.1).

4.3 GDL

GDL (de Geometric Description Language) é a linguagem de suporte ao Desenho Generativo do ArchiCAD [7]. A linguagem GDL foi fortemente influenciada pela linguagem BASIC (College, 1964), contendo as mesmas estruturas de controlo, lógica das variáveis, etc. No entanto, a linguagem adopta conceitos específicos do domínio, tais como: Edifícios, Colunas, Telhados, Parede, etc. como pode-se verificar nos exemplos das secções subsequentes.

4.3.1 Operadores Geométricos da Linguagem

A construção de formas geométricas de 2 e 3 dimensões pode ser feita recorrendo a inúmeros operadores da linguagem. São exemplos desses operadores: *BLOCK*, *CYLIND*, *SPHERE*, *CONE*, *PRISM* - 3D (Graphisoft, 2004). *LINE2*, *RECT2*, *POLY2*, *CIRCLE2* e *ARC2* - 2D (Graphisoft, 2004). Na figura 4.3.1 apresenta-se o resultado da aplicação de alguns desses operadores no script 4.3.1.

```
1 BLOCK 1.0, 2.0, 3.0
2 CYLIND 1.0, 1.0
3 SPHERE 1.0
4 CONE 4, 0.5, 0.1, 90, 90
```

Script 4.3.1: Exemplos de utilização de construtores geométricos.

[7] http://www.graphisoft.com/archicad/

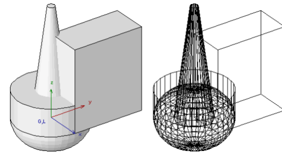


Figura 4.3.1: Forma geométrica resultante da execução do script 4.3.1.

4.3.2 Navegação no espaço 2D e 3D

Como se pode verificar na figura 4.3.1, todas as formas geométricas definidas no script 4.3.1 têm a mesma origem. A linguagem fornece alguns operadores que permitem a criação de formas geométricas em diferentes espaços. São exemplos desses operadores: *ADD*, *ADDX*, *ADDY*, *ADDZ*, ROT, *ROTX*, *ROTY*, *ROTZ*, MUL, *MULX*, *MULY*, *MULZ* - 3D (Graphisoft, 2004). *ADD2*, *ROT2* - 2D (Graphisoft, 2004).

No script 4.3.2 aplicam-se algumas dessas transformações às formas geométricas definidas em 4.3.1. Na figura 4.3.2 apresentam-se as formas resultantes.

```
1  BLOCK 1.0, 2.0, 3.0
2  ADD 1.0, 1.0, 1.5
3  ROTy 90
4  CYLIND 1.0, 1.0
5  ADDz 2.0
6  MULz 0.5
7  SPHERE 1.0
8  ADDz -2.0
9  CONE 5, 0.5, 0.1, 90, 90
10 DEL 5
```

Script 4.3.2: Exemplos de utilização das transformações geométricas.

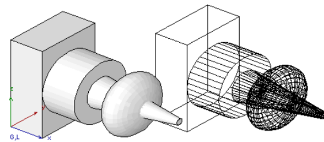


Figura 4.3.2: Formas geométricas resultantes da execução do script 4.3.2.

A última expressão do script 4.3.2 (*DEL 5*), remove as cinco últimas transformações efectuadas ao "cursor" (linhas: 2, 3, 5, 6 e 8), retornando assim a sua posição inicial (denominada "Origem Global").

Além da operação *DEL n*, onde *n* é o número de transformações que pretendemos descartar (da mais recente para a mais antiga), pode-se ainda utilizar as seguintes operações predefinidos para gerir a pilha de transformações: *DEL TOP* e *NTR*, correspondendo a operação que remove todas as transformações efectuadas no script actual e a operação que devolve o número actual de transformações, respectivamente.

4.3.3 Exemplos Arquitecturais

A linguagem GDL dispõe de uma grande quantidade de operadores que permitem a criação de figuras geométricas, algumas muito complexas (Graphisoft, 2004). Nesta secção apresentam-se alguns exemplos:

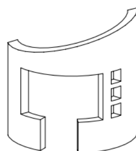


Figura 4.3.3: Forma geométrica resultante da execução do script 4.3.3.

```
1  ROTZ -60
2  BWALL_ 1, 1, 1,
3      4, 0, 6, 6, 0,
4      0.3, 2,
5      15, 15, 15, 15,
6      5,
7      1, 1, 3.8, 2.5, -255,
8      1.8, 0, 3, 2.5, -255,
9      4.1, 1, 4.5, 1.4, -255,
10     4.1, 1.55, 4.5, 1.95,-255,
11     4.1, 2.1, 4.5, 2.5, -255,
12     1, 0, -0.25, 1, 3
```

Script 4.3.3: Construção de uma forma geométrica recorrendo a uma das variações da primitiva WALL.

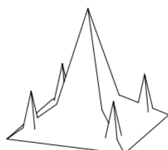


Figura 4.3.4: Forma geométrica resultante da execução do script 4.3.4.

```
1  PYRAMID 4, 1.5, 1+4+16, -2, -2, 0, -2, 2, 0, 2, 2, 0, 2, -2, 0
2  PYRAMID 4, 4, 21, -1, -1, 0, 1, -1, 0, 1, 1, 0, -1, 1, 0
3  ADDX -1.4
4  ADDY -1.4
5  GOSUB 100
6  ADDX 2.8
7  GOSUB 100
8  ADDY 2.8
9  GOSUB 100
10 ADDX -2.8
11 GOSUB 100
12 END
13 100:
14 PYRAMID 4, 1.5, 21, -0.25, -0.25, 0, 0.25, -0.25, 0, 0.25, 0.25, 0, -0.25, 0.25, 0
15 RETURN
```

Script 4.3.4: Exemplo utilizando transformações e sub-rotinas.

4.3.4 Conclusões

Depois dos exemplos aqui apresentados conclui-se facilmente que a linguagem GDL é uma linguagem muito poderosa. A linguagem define inúmeros conceitos específicos do domínio - Design e Arquitectura.

No entanto, a linguagem contém demasiadas primitivas, com sintaxe e semântica complexas. Um exemplo disso é a primitiva *PRISM* e suas variações (*CPRISM*, *BPRISM*, *FPRISM*, *HPRISM*, *SPRISM*, etc). Essas múltiplas primitivas (muitas delas com inúmeros parâmetros) são um entrave à assimilação e à utilização da linguagem.

Sendo a linguagem da família do BASIC uma linguagem muito antiga (1964), GDL herdou conceitos e semânticas que em nada facilitam a prática da programação. É o caso do *GOTO* (script 4.3.4).

Outro aspecto propício a erros é a gestão da pilha de transformações geométricas, toda ela efectuada pelo programador (secção 4.3.2). O autor de um dos livros de introdução à linguagem GDL alerta: "Because you can also Rotate and Multiply, you are in danger of getting lost – after a few lines you could be 'deep in spaghetti'. After each set of moves, you should return to the origin using DEL before doing the next job." (Nicholson-Cole, 2000).

4.4 MEL

MEL (de Maya Embedded Language) é uma linguagem de script descendente do shell scripting do UNIX e encontra-se presente na ferramenta de CAD Maya[8] da Autodesk.

MEL é utilizada para a criação da interface utilizador do Maya e pode ser utilizada para estender as suas funcionalidades.

MEL é uma linguagem de propósito geral típica. Os elementos básicos da linguagem são semelhantes aos que podemos encontrar em linguagens como o C e o Java.

A linguagem é fortemente tipificada, suportando os seguintes tipos básicos: int, float, string, array, matrix e vector, correspondendo a: números inteiros, números fraccionários, cadeias de caracteres, lista de valores (todos do mesmo tipo), tabela de duas dimensões de números fraccionários e ao triplo de números fraccionários, respectivamente.

4.4.1 Conclusões

Sendo uma linguagem de propósito geral, a linguagem MEL embora seja utilizada para criar a interface gráfica do Maya e possa ser utilizada para estender as funcionalidades da ferramenta, não fornece suporte ao Desenho Generativo, no entanto, a linguagem pode utilizar a API do Maya[9] para manipular objectos do domínio.

Uma vez que a manipulação de objectos de 3 dimensões implica frequentemente a manipulação de números fraccionários, o *float* é o tipo básico da linguagem. O tipo *vector* presente na linguagem corresponde a um triplo de números fraccionários e abstrai a noção de ponto nas coordenas X, Y, Z.

O facto de a linguagem ser descendente do shell scripting do UNIX torna a linguagem complexa e confusa na medida em que é fortemente baseada na execução de comandos em vez da necessária abstracção fornecida pela chamada de funções ou a utilização de métodos orientado a objectos; permite tanto a utilização da sintaxe baseada em comandos e a baseada em funções.

[8] `http://www.autodesk.com/products/maya/overview`

[9] `http://help.autodesk.com/view/MAYAUL/2016/ENU/`

4.5 AutoLisp

AutoLisp é uma linguagem da família Lisp, muito antiga, criada para estender as funcionalidades do AutoCAD[10], permitindo o Desenho Generativo a partir da sua ligação com o AutoCAD.

Sendo o AutoCAD uma das mais antigas e utilizadas ferramentas de CAD, o AutoLisp é uma linguagem muito conhecida por essa comunidade, existindo muito código disponível.

A ligação entre a linguagem com o AutoCad é feita através da submissão de comandos para serem executados na consola da ferramenta. A submissão de comandos é feita utilizando a função *command*, uma função genérica que aceita qualquer número de argumentos. No excerto de código em 4.5.1 apresenta-se exemplos de utilização desse mecanismo.

```
1 (command "circle" "0,0" "3,3")
2 (command "thickness" 1)
3 (setq p1 (list 1.0 1.0 3.0))
4 (setq rad 4.5)
5 (command "circle" p1 rad)
```

Script 4.5.1: Exemplo de utilização da função *command*.

Além dos comandos disponibilizados pela API do AutoCad (Autodesk, 2012), a linguagem fornece um conjunto de operadores geométricos (Autodesk, 2012). Esses operadores facilitam a criação, a manipulação e a obtenção de informações de formas geométricas.

4.5.1 Operadores Geométricos da Linguagem

4.5.1.1 Angle - A função *angle* - *(angle pt1 pt2)* - determina o ângulo (em radianos) entre uma linha e o eixo dos *XX*

4.5.1.2 Distance - A função *distance* - *(distance pt1 pt2)* - determina a distância entre dois pontos.

4.5.1.3 Polar - A função *polar* - *(polar pt ang dist)* - determina um ponto a uma certa distância e que forma um certo ângulo com o ponto original.

4.5.1.4 Inters - A função *inters* - *(inters pt1 pt2 pt3 pt4 [onseg])* - determina a intersecção entre duas linhas.

```
1 (setq pt1 (list 3.0 6.0 0.0))
2 (setq pt2 (list 5.0 2.0 0.0))
3 (setq ang (angle pt1 pt2)) ; angulo entre a linha que une p1 e p2 e o eixo dos xx.
4 (setq dst (distance pt1 pt2)) ; distancia entre os pontos p1 e p2.
5 (setq pt3 (polar pt1 ang dst)) ;ponto que forma um angulo ang com o ponto p1 e que dista desse de dst.
```

Script 4.5.2: Exemplo de utilização das funções: angle; distance e polar.

[10] http://www.autodesk.pt/products/autocad/

4.5.2 Conclusões

Sendo o AutoLisp um dialecto da linguagem Lisp, uma linguagem muito antiga, a linguagem é muito limitada no que diz respeito a alguns elementos presentes nas linguagens de programação modernas, como é o caso das estruturas de dados e das excepções.

Estudos recentes demonstraram a importância das funções de ordem superior para o Desenho Generativo (Leitão, 2014), no entanto, AutoLisp apresenta problemas graves no que diz respeito ao suporte dessas funções (Cabecinhas, 2010). São exemplos desses problemas o *downward* e o *upward funarg problems* (Moses, 1970), problemas resultantes da utilização de *scope dinâmico* na implementação da linguagem.

A par dos problemas já mencionados, a linguagem é muito limitada no que diz respeito a funções geométricas (secção 4.5.1), logo, limitada no suporte ao Desenho Generativo.

4.6 RhinoScript

RhinoScript (Rutten, 2007) é uma linguagem de script de suporte ao Desenho Generativo da ferramenta de CAD Rhrinoceros[11]. A linguagem é uma extensão da linguagem VBScript[12] da Microsoft, na medida em que implementa todos os elementos da linguagem, acrescentando-lhes elementos específicos do domínio (Design e Arquitectura).

O Rhinoceros inclui uma biblioteca chamada OpenNURBS[13]. OpenNURBS fornece um conjunto de funções geométricas e de Input/Output. RhinoScript, tal como outras ferramentas que se ligam ao Rhinoceros (Rutten, 2007), utiliza essa biblioteca para fornecer operadores geométricos aos programadores. No entanto, os programadores não têm de lidar directamente com a biblioteca, uma vez que RhinoScript encapsula essa biblioteca fornecendo aos programadores essas funcionalidades sobre a forma de uma *package*.

4.6.1 Exemplos Arquitecturais [14]

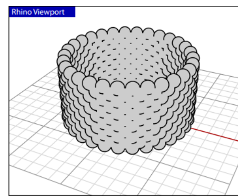


Figura 4.6.1: Forma geométrica resultante da execução do script 4.6.1.

```
1 Call TwistAndShout()
2 Sub TwistAndShout()
3   Dim z, a
4   Dim pi = Rhino.Pi()
5   Dim dblTwistAngle = 0.0
```

[11] https://www.rhino3d.com

[12] https://msdn.microsoft.com/en-us/library/t0aew7h6(v=vs.84).aspx

[13] https://www.rhino3d.com/opennurbs

[14] Exemplos adaptados de “Rhinoscript 101 for Rhinoceros 4.0”

```
 6    Call Rhino.EnableRedraw(False)
 7    For z = 0.0 To 5.0 Step 0.5
 8      dblTwistAngle = dblTwistAngle + (pi/30)
 9      For a = 0.0 To 2 * pi Step (pi/15)
10        Dim x = 5 * Sin(a + dblTwistAngle)
11        Dim y = 5 * Cos(a + dblTwistAngle)
12        Call Rhino.AddSphere(Array(x,y,z), 0.5)
13      Next
14    Next
15    Call Rhino.EnableRedraw(True)
16  End Sub
```

Script 4.6.1: Construção de um cilindro a partir da combinação de esferas.

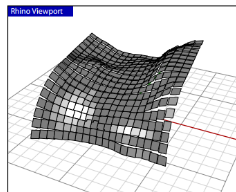


Figura 4.6.2: Forma geométrica resultante da execução do script 4.6.2.

```
 1  Call WhoFramedTheSurface()
 2  Sub WhoFramedTheSurface()
 3    Dim idSurface = Rhino.GetObject("Surface", 8, True, True)
 4    If IsNull(idSurface) Then Exit Sub
 5    Dim intCount = Rhino.GetInteger("# Iterations", 20, 2)
 6    If IsNull(intCount) Then Exit Sub
 7    Dim uDomain = Rhino.SurfaceDomain(idSurface, 0)
 8    Dim vDomain = Rhino.SurfaceDomain(idSurface, 1)
 9    Dim uStep = (uDomain(1) - uDomain(0)) / intCount
10    Dim vStep = (vDomain(1) - vDomain(0)) / intCount
11    Dim u, v, pt, srfFrame
12    Call Rhino.EnableRedraw(False)
13    For u = uDomain(0) To uDomain(1) Step uStep
14      For v = vdomain(0) To vDomain(1) Step vStep
15        pt = Rhino.EvaluateSurface(idSurface, Array(u, v))
16        If Rhino.Distance(pt, Rhino.BrepClosestPoint(idSurface, pt)(0)) < 0.1 Then
17          srfFrame = Rhino.SurfaceFrame(idSurface, Array(u, v))
18          Call Rhino.AddPlaneSurface(srfFrame, 1.0, 1.0)
19        End If
20      Next
21    Next
22    Call Rhino.EnableRedraw(True)
23  End Sub
```

Script 4.6.2: Construção de superfícies a partir de planos.

4.6.2 Conclusões

Sendo o RhinoScript uma extensão da linguagem VBScript (uma linguagem de propósito geral), RhinoScript não fornece o devido suporte ao Desenho Generativo (embora utilize a biblioteca OpenNURBS para disponibilizar funções de modelação geométrica). Prova disso é a adopção de inúmeras outras linguagens pela comunidade do Rhinoceros (Python, por exemplo).

RhinoScript, à semelhança do VBScript, possui uma sintaxe pouco rígida, bem como sintaxe diferente para a declaração e invocação de funções e sub-rotinas, o que constitui uma fonte de confusão e erros complicados de detectar.

As funções do RhinoScript devolvem *Null* quando falham, o que leva a que os programas em Rhinoscript tenham muitos blocos de teste do retorno dessas funções.

No entanto, a sintaxe da linguagem permitiu a eliminação da maioria dos parênteses e chavetas que "poluem" programas escritos em linguagens como o Lisp, C, o Java, etc.

4.7 Processing

Processing (Casey Reas, 2014) é uma linguagem de código aberto (*open source*) criada em 2001 pelos investigadores Casey Reas e Benjamin Fry do MIT Media Lab. O propósito da linguagem é dar suporte ao desenho electrónico, desenho visual e artes gráficas. A linguagem corresponde a uma sintaxe simplificada do Java (linguagem na qual esta implementada), adicionando elementos para computação gráfica recorrendo a biblioteca OpenGL[15] para o processo de renderização.

No entanto, a linguagem expandiu-se para outras plataformas, como é o caso dos browsers através do projecto Processing.js[16] e dos dispositivos equipados com o sistema operativo iOS através do projecto iProcessing[17].

4.7.1 Estrutura dos programas

Uma vez que o Processing recorre ao OpenGL para o processo de renderização e não a uma ferramenta de CAD, os programas tem uma estrutura predefinida, tendo o programador ao seu dispor algumas funções para controlar esse processo.

4.7.1.1 Função setup()

A função *setup()* é a função que é invocada pelo sistema no início da execução de um programa, por isso, as configurações iniciais devem ser efectuadas dentro dessa função, apenas podendo existir uma única função *setup()* em cada programa.

São exemplos de funções de configuração do ambiente as seguintes funções[18]:

– size(int width, int height [, String renderer])
– frameRate(int framesPerSecound)
– background(...)

4.7.1.2 Funções draw() e redraw()

A função *draw()* é invocada pelo sistema na sequência da execução da função *setup()*. A função é invocada continuamente enquanto o programa estiver em execução ou até a invocação da função *noLoop()*. A frequência de invocação dessa função pode ser alterada recorrendo a função *frameRate(int framesPerSecound)*. A função *draw()* não deve nunca ser invocada explicitamente, em vez disso, deve-se utilizar as funções *loop()* e *noLoop()* para controlar a sua execução. Por outro lado, a invocação da função *redraw()* força uma única execução da função *draw()*.

[15] `https://www.opengl.org/`

[16] Implementação JavaScript do Processing que utiliza o elemento Canvas do HTML no processo de renderização.

[17] `http://luckybite.com/iprocessing/`

[18] A referência completa pode ser consultada em `https://processing.org/reference`.

4.7.2 Operadores Geométricos da Linguagem

Além das funções de configuração e controlo do processo de renderização apresentadas na secção 4.7.1, Processing estende a sintaxe do Java adicionando operadores geométricos. Nesta secção apresentam-se exemplos de alguns desses operadores[18].

4.7.2.1 createShape()

A função *createShape* permite criar uma nova figura geométrica. A figura geométrica criada é do tipo *PShape*. *PShape* é a estrutura de dados que permite guardar e manipular figuras geométricas. A função *shape()* permite fazer a renderização de um objecto do tipo *PShape*. Exemplo:

```
1 PShape square;
2 void setup() {
3   size(100, 100, P2D);
4   square = createShape(RECT, 0, 0, 50, 50);
5   square.setFill(color(0, 0, 255));
6 }
7 void draw() {
8   shape(square, 25, 25);
9 }
```

Script 4.7.1: Utilização da função *createShape* e *shape* para construir e fazer a renderização de um rectângulo, respectivamente.

4.7.2.2 loadShape()

A função *loadShape* permite carregar uma figura geométrica a partir de um ficheiro. A figura geométrica carregada será do tipo *PShape*. O ficheiro de origem deve ser to tipo SVG ou OBJ. A função devolve *null* no caso de não conseguir carregar o ficheiro. Exemplo:

```
1 PShape s;
2 void setup() {
3   size(100, 100);
4   s = loadShape("bot.svg");
5 }
6 void draw() {
7   if(s != null)
8     shape(s, 10, 10, 80, 80);
9 }
```

Script 4.7.2: Utilização da função *loadShape* e *shape* para carregar e fazer a renderização da figura guardada no ficheiro bot.svg, respectivamente.

4.7.2.3 point()

A função *point(float x, float y)* permite desenhar um ponto na posição em *(x,y)* e a função *point(float x, float y, float z)* permite desenhar um ponto na posição *(x,y,z)*.

4.7.2.4 line()

A função *line(float x1, float y1, float x2, float y2)* permite desenhar uma linha que une os pontos *(x1,y1)* e *(x2,y2)* e a função *line(float x1, float y1, float z1, float x2, float y2, float z2)* permite uma linha que une os pontos *(x1,y1,z1)* e *(x2,y2,z2)*.

4.7.2.5 arc()

A função *arc(float x, float y, float width, float height, float start, float stop[, int mode])* permite desenhar um arco no ponto *(x,y)*, a partir do ângulo *start* até ao ângulo *stop* e com largura *width* e altura *height*. Exemplos:

```
1  // bloco (a)
2  arc(50, 55, 50, 50, 0, HALF_PI);
3  noFill();
4  arc(50, 55, 60, 60, HALF_PI, PI);
5  arc(50, 55, 70, 70, PI, PI+QUARTER_PI);
6  arc(50, 55, 80, 80, PI+QUARTER_PI, TWO_PI);
7  // bloco (b)
8  arc(50, 50, 80, 80, 0, PI+QUARTER_PI, OPEN);
9  // bloco (c)
10 arc(50, 50, 80, 80, 0, PI+QUARTER_PI, CHORD);
11 // bloco (d)
12 arc(50, 50, 80, 80, 0, PI+QUARTER_PI, PIE);
```

Script 4.7.3: Exemplos utilização da função *arc*.

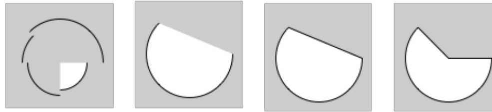


Figura 4.7.1: Formas geométricas resultantes da execução dos blocos de código: (a), (b), (c) e (d) do script 4.7.3, respectivamente.

4.7.2.6 ellipse()

A função *ellipse(float x, float y, float width, float height)* permite desenhar uma elipse na posição *(x,y)* com largura *width* e altura *height*.

4.7.2.7 rect()

A função *rect(float x1, float y1, float width, float height)* permite desenhar um quadrado com vértice superior esquerdo no ponto *(x1,y1)*, largura *width* e altura *height*. Pode-se ainda indicar um ângulo para os cantos ou um ângulo específico para cada canto.

4.7.2.8 quad()

A função *quad(float x1, float y1, float x2, float y2, float x3, float y3, float x4, float y4)* permite desenhar um quadrilátero com vértices nos pontos *(x1,y1)*, *(x2,y2)*, *(x3,y3)* e *(x4,y4)*.

4.7.2.9 triangle()

A função *triangle(float x1, float y1, float x2, float y2, float x3, float y3)* permite desenhar um triângulo com vértices nos pontos *(x1,y1)*, *(x2,y2)*, *(x3,y3)*.

4.7.2.10 box()

A função *box(float w, float h, float d)* permite desenhar um paralelepípedo. Correspondendo os parâmetros *w*, *(h)* e *d* à largura, altura e profundidade do paralelepípedo, respectivamente.

No caso dos parâmetros *w*, *(h)* e *d* serem todos iguais a figura resultante será um cubo, podendo nesse caso utilizar a função *box(float size)*, correspondendo o parâmetro *size* a dimensão dos lados do cubo.

4.7.2.11 sphere()

A função *sphere(float r)* permite desenhar uma esfera de raio *r*.

4.7.3 Exemplos Arquitecturais [19]

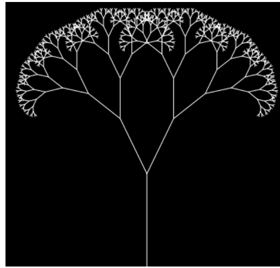


Figura 4.7.2: Forma geométrica resultante da execução do script 4.7.4.

```
1  float theta;
2  void setup() {
3    size(640, 360);
4  }
5  void draw() {
6    background(0);
7    frameRate(30);
8    stroke(255);
9    float a = (mouseX / (float) width) * 90f;
10   theta = radians(a);
11   translate(width/2,height);
12   line(0,0,0,-120);
13   translate(0,-120);
14   branch(120);
15 }
16 void branch(float h) {
17   h *= 0.66;
18   if (h > 2) {
19     pushMatrix();
20     rotate(theta);
21     line(0, 0, 0, -h);
22     translate(0, -h);
23     branch(h);
24     popMatrix();
25     pushMatrix();
26     rotate(-theta);
27     line(0, 0, 0, -h);
28     translate(0, -h);
```

[19] Exemplos adaptados a partir de `https://processing.org/reference/`.

```
29      branch(h);
30      popMatrix();
31    }
32  }
```

Script 4.7.4: Construção recursiva de uma árvore.

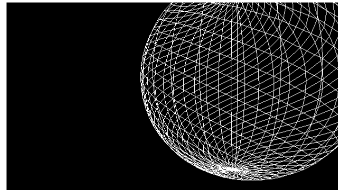


Figura 4.7.3: Forma geométrica resultante da execução do script 4.7.5.

```
1  void setup(){
2    size(640, 360, P3D);
3    background(0);
4    lights();
5    noLoop();
6  }
7  void draw(){
8    noStroke();
9    pushMatrix();
10   translate(130, height/2, 0);
11   rotateY(1.25);
12   rotateX(-0.4);
13   box(100);
14   popMatrix();
15   noFill();
16   stroke(255);
17   pushMatrix();
18   translate(500, height*0.35, -200);
19   sphere(280);
20   popMatrix();
21 }
```

Script 4.7.5: Exemplo de utilização de primitivas 3D e transformações geométricas.

4.7.4 Conclusões

A linguagem Processing é de domínio vasto uma vez que a sua criação teve como objectivo dar suporte ao desenho electrónico, desenho visual e artes gráficas, facto que faz com que a linguagem forneça pouco suporte ao Desenho Gerativo.

A linguagem herdou toda a expressividade do Java, no entanto, a tentativa de simplificação da sintaxe do Java não fez com que a linguagem se torna-se mais fácil de aprender e utilizar.

No entanto, a linguagem tem tido muito sucesso e tem ganho utilizadores nos diferentes domínios. Tendo os seus criadores e desenvolvedores recebido muitos prémios ao nível internacional. De entre os prémios destacam-se o *"Golden Nica award"* na categoria de *"Net Vision"* atribuído pelo instituto Australiano *Ars Electronica* em 2005 e o *"USA Design Award"* na categoria de *"Interaction Design"* atribuído pelo museu do design Smithsonian Cooper-Hewitt em 2011.

4.8 Dynamo

Dynamo (Autodesk, xxxx) é uma ferramenta para programação visual. A ferramenta permite a especificação de scripts através da sua linguagem textual (*Dynamo textual language*, antigo *DesignScript*), bem como a visualização do comportamento dos scripts, ou seja, a ferramenta permite a mistura de elementos gráficos e textuais para a especificação de um programa.

A linguagem Dynamo[20] é uma linguagem orientada a objectos com uma sintaxe parecida com a sintaxe de linguagens como C, C++ ou Java. A linguagem dispõe de uma biblioteca que fornece operadores geométricos de suporte ao Desenho Generativo.

4.8.1 Operadores Geométricos da Linguagem

4.8.1.1 Construção de Pontos

Na linguagem Dynamo um ponto é representado pela classe *Point*. A classe dispõe de construtores para a construção de pontos. São exemplos desses construtores *Point.ByCoordinates* e *Point.BySphericalCoordinates*. Em 4.8.1 exemplifica-se a utilização desses construtores.

```
1  x = 10;
2  y = 2.5;
3  z = -6;
4  p1 = Point.ByCoordinates(x, y, z);
5
6  radius = 5;
7  theta = 75.5;
8  phi = 120.3;
9  cs = CoordinateSystem.Identity();
10 p2 = Point.BySphericalCoordinates(cs, radius, theta, phi);
```

Script 4.8.1: Construção de dois pontos.

4.8.1.2 Construção de Linhas

Uma linha é representada pela classe *Line*. A classe *Line* disponibiliza construtores para a criação de linhas. Um exemplo é o construtor *Line.ByStartPointEndPoint*. Em 4.8.2 exemplifica-se a utilização desse construtor.

```
1 p1 = Point.ByCoordinates(3, 10, 2);
2 p2 = Point.ByCoordinates(-15, 7, 0.5);
3 l = Line.ByStartPointEndPoint(p1, p2);
```

Script 4.8.2: Construção de uma linha a partir de dois pontos.

4.8.1.3 Construção de Curvas

A linguagem fornece duas formas distintas para a construção de curvas: (1) Através da especificação de um conjunto de pontos por onde a curva deve passar (construtor *NurbsCurve.ByPoints*) e (2) Através da especificação de “pontos de controlos” (construtor *NurbsCurve.ByControlPoints*).

[20] Por simplicidade quando se refere ao Dynamo refere-se à linguagem textual do Dynamo.

Para as curvas do tipo 1 (chamadas *Curvas interpretadas*) o Dynamo cria uma curva suave que une os pontos especificados. Este tipo de curva é útil quando o designer sabe exactamente a forma que quer dar à curva, ou quando sabe por onde a curva deve ou não passar.

Para as curvas do tipo 2 (chamadas *Curvas pontos de controlo*) o Dynamo cria linhas rectas que unem os pontos especificados. Na especificação da curva pode-se indicar o "grau de suavização" (um inteiro entre 1 e 20). Quanto maior for o "grau de suavização" menor será o declive dos segmentos de curva que unem os pontos. Este tipo de curva é útil quando o designer pretende explorar a forma a dar a uma curva.

4.8.1.4 Construção de Superfícies

A representação do Dynamo para uma superfície é a classe *Surface*. A construção de superfícies é efectuada através de construtores disponibilizados pela classe. Um exemplo é o construtor *Surface.ByLoft*. No código apresentado em 4.8.3 exemplifica-se a utilização desse construtor.

```
1  // pontos:
2  p1 = Point.ByCoordinates(3, 10, 2);
3  p2 = Point.ByCoordinates(-15, 7, 0.5);
4  p3 = Point.ByCoordinates(5, -3, 5);
5  p4 = Point.ByCoordinates(-5, -6, 2);
6  p5 = Point.ByCoordinates(9, -10, -2);
7  p6 = Point.ByCoordinates(-11, -12, -4);
8  // linhas:
9  l1 = Line.ByStartPointEndPoint(p1, p2);
10 l2 = Line.ByStartPointEndPoint(p3, p4);
11 l3 = Line.ByStartPointEndPoint(p5, p6);
12 // superficie:
13 surf = Surface.ByLoft({l1, l2, l3});
```

Script 4.8.3: Construção de uma superfície a partir de 3 linhas.

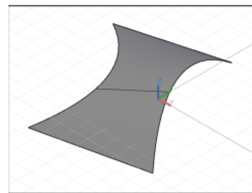


Figura 4.8.1: Superfície resultante da execução do script 4.8.3.

Tal como a classe *NurbsCurve* apresentada na secção 4.8.1.3 para a construção de curvas (1D), a classe *NurbsSurface* permite a construção de superfícies (2D). A construção é efectuada através da especificação de um conjunto de pontos por onde a superfície deve passar ou através da especificação de um conjunto de pontos de controlo.

A construção dessas superfícies é efectuada através do método *NurbsSurface.ByPoints*. Quando o método é invocado com apenas 1 argumento (os pontos) a superfície a ser criada será uma "superfície interpretada", e caso o método seja invocado com 3 argumentos (pontos e "grau de suavização" para cada uma das duas dimensões) a superfície a ser criada será uma "superfície pontos de controlo").

É ainda possível construir superfícies a partir da especificação de um conjunto de curvas (método *Surface.LoftFromCrossSections*) ou através da varredura com uma curva em torno do eixo central, resultando

no que se chama uma *superfície de revolução* (método *Surface.ByRevolve*). No script 4.8.4 apresenta-se um exemplo da construção de uma *superfície de revolução* e na figura 4.8.2 apresenta-se a figura geométrica resultante dessa construção.

```
1  pts = {};
2  pts[0] = Point.ByCoordinates(4, 0, 0);
3  pts[1] = Point.ByCoordinates(3, 0, 1);
4  pts[2] = Point.ByCoordinates(4, 0, 2);
5  pts[3] = Point.ByCoordinates(4, 0, 3);
6  pts[4] = Point.ByCoordinates(4, 0, 4);
7  pts[5] = Point.ByCoordinates(5, 0, 5);
8  pts[6] = Point.ByCoordinates(4, 0, 6);
9  pts[7] = Point.ByCoordinates(4, 0, 7);
10 crv = NurbsCurve.ByPoints(pts);
11 axis_origin = Point.ByCoordinates(0, 0, 0);
12 axis = Vector.ByCoordinates(0, 0, 1);
13 surf = Surface.ByRevolve(crv, axis_origin, axis, 0, 360);
```

Script 4.8.4: Construção de uma superfície de revolução.

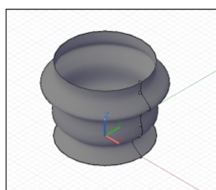


Figura 4.8.2: Superfície de revolução resultante da execução do script 4.8.4.

4.8.1.5 Construção de Cubos, Cones, Cilindros e Esferas

Dynamo fornece as seguintes representações (classes) para as figuras geométricas primitivas de 3 dimensões: Cubos (*Cuboid*), Cones (*Cone*), Cilindros (*Cylinder*) e Esferas (*Sphere*). Essas classes fornecem construtores para a criação dessas figuras geométricas. No script 4.8.5 apresenta-se exemplos da aplicação de alguns desses construtores.

```
1  // Paralelepipedo
2  cs = CoordinateSystem.Identity();
3  cub = Cuboid.ByLengths(cs, 5, 15, 2);
4  // Cones
5  p1 = Point.ByCoordinates(0, 0, 10);
6  p2 = Point.ByCoordinates(0, 0, 20);
7  p3 = Point.ByCoordinates(0, 0, 30);
8  cone1 = Cone.ByPointsRadii(p1, p2, 10, 6);
9  cone2 = Cone.ByPointsRadii(p2, p3, 6, 0);
10 // Cilindro
11 cylCS = cs.Translate(10, 0, 0);
12 cyl = Cylinder.ByRadiusHeight(cylCS, 3, 10);
13 // Esfera
14 centerP = Point.ByCoordinates(-10, -10, 0);
15 sph = Sphere.ByCenterPointRadius(centerP, 5);
```

Script 4.8.5: Exemplos da aplicação de construtores de primitivas 3D.

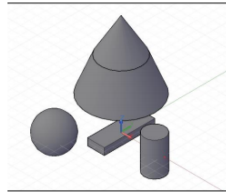


Figura 4.8.3: Formas geométricas resultantes da execução do script 4.8.5.

4.8.1.6 Thicken (Ampliar)

A operação *Thicken* é uma operação disponibilizada pela classe *Surface* e que permite ampliar / "engrossar" uma superfície. No script 4.8.6 apresenta-se um exemplo da aplicação do operador e na figura 4.8.4 a figura geométrica resultante dessa aplicação[21].

```
1 p1 = Point.ByCoordinates(3, 10, 2);
2 p2 = Point.ByCoordinates(-15, 7, 0.5);
3 p3 = Point.ByCoordinates(5, -3, 5);
4 p4 = Point.ByCoordinates(-5, -6, 2);
5 l1 = Line.ByStartPointEndPoint(p1, p2);
6 l2 = Line.ByStartPointEndPoint(p3, p4);
7 surf = Surface.ByLoft({l1, l2});
8 solid = surf.Thicken(4.75, true);
```

Script 4.8.6: Exemplo da aplicação da operação *Thicken* a duas faces de uma superfície.

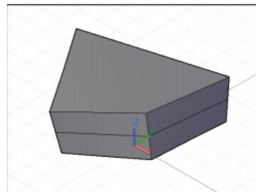


Figura 4.8.4: Figura geométrica resultante da execução do script 4.8.6.

4.8.1.7 Intersect (Intersecção)

A operação *Intersect* é uma operação que permite extrair figuras geométricas de dimensões inferiores a partir de figuras geométricas de dimensão superior.

4.8.1.8 Translate, Rotate e Scale

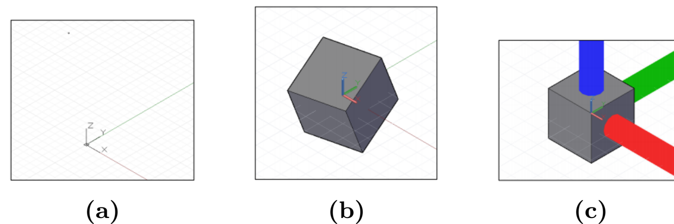
Dynamo fornece operadores que permitem aplicar transformações geométricas sobre objectos ou sobre o seu sistema de coordenadas. Exemplos dessas transformações incluem: translações (*Translate*), rotações

[21] O segundo parâmetro do método *Thicken* define se a operação deve ser aplicado ou não as duas faces da superfície

(*Rotate*) e escala (*Scale*). No código presente em 4.8.7 exemplifica-se a aplicação dessas 3 transformações e na figura 4.8.5 apresenta-se o resultado dessas transformações.

```
//Translacao
p = Point.ByCoordinates(1, 2, 3);
p = p.Translate(10, -20, 50);
//Rotacao
cube1 = Cuboid.ByLengths(CoordinateSystem.Identity(), 10, 10, 10);
newCs = CoordinateSystem.Identity();
newCs = newCs.Rotate(Point.ByCoordinates(0, 0),Vector.ByCoordinates(1,0,0.5), 25);
oldCs = CoordinateSystem.Identity();
cube1 = cube1.Transform(oldCs, newCs);
//Escala
cube2 = Cuboid.ByLengths(CoordinateSystem.Identity(), 10, 10, 10);
newCs = CoordinateSystem.Identity();
newCs = newCs.Scale(20);
oldCs = CoordinateSystem.Identity();
cube2 = cube2.Transform(oldCs, newCs);
```

Script 4.8.7: Exemplo da aplicação das operações *Translate*, *Rotate* e *Scale*.



(a) **(b)** **(c)**

Figura 4.8.5: Formas geométricas resultantes da execução do script 4.8.7. Correspondendo a: (a) translação do ponto p (b) rotação do cubo *cube1* (c) escala do cubo *cube2*.

4.8.1.9 Operadores Booleanos

Dynamo disponibiliza os seguintes operadores booleanos aplicáveis a sólidos geométricos (objectos derivados da classe *Solid*):

- União (método *Union*) - Dado um sólido base e um sólido de input, devolve um novo que corresponde a *união* dos dois;
- Diferença (método *Difference*) - Dado um sólido base e um sólido de input, devolve um novo que corresponde ao resultado de subtrair o espaço do sólido de input ao sólido base;
- Intersecção (método *Intersect*) - Dado um sólido base e um sólido de input, devolve um novo que corresponde a *intersecção* dos dois.

4.8.2 Conclusões

O Dynamo utiliza a expressividade de uma linguagem textual para colmatar as deficiências das linguagens gráficas. A linguagem é uma linguagem expressiva, contendo objectos específicos do domínio e disponibiliza muitas operações para a manipulação desses objectos.

No entanto, a linguagem exige muito tempo para a sua aprendizagem, sendo a sua sintaxe muito parecida com a sintaxe de linguagens como o C++ e o Java (linguagens de propósito geral). A mistura de elementos gráficos e textuais é uma possível fonte de confusão, tendo o "programador" que se preocupar com a ligação entre os dois tipos de componentes, além da preocupação com a sintaxe de cada um dos componentes (gráfico e textual).

4.9 Rosetta

Rosetta (Lopes, 2012) é um ambiente de desenvolvimento para Desenho Generativo. O sistema foi desenvolvido tendo em vista 3 princípios. (1) portabilidade dos programas, (2) rigor matemático e (3) forte relação entre os programas e os modelos gerados por estes.

O sistema suporta várias linguagens de programação (ou *frontends*) e várias ferramentas de CAD (ou *backends*). AutoLisp, JavaScript, Python, Processing, RosettaRacket[22] são exemplos de algumas das linguagens de programação implementadas pelo Rosetta e AutoCad, Rhinoceros3D e OpenGL são alguns dos *backends* suportados.

Além dos componentes de *frontend* e *backend*, Rosetta contém um componente chamado *core*, componente que dá suporte aos outros componentes. O componente *core* abstrai a utilização de vários *frontends* e *backends* fornecendo funcionalidades que são comuns ao ambiente de desenvolvimento e a todos os programas. Como é caso de: tipos de dados primitivos (vectores, listas, etc.), tipos de dados geométricos (formas geométricas, sistemas de coordenadas, matrizes, cores, etc.), construtores de formas geométricas (pontos, círculos, quadrados, etc.) e transformações geométricas (translações, rotações, *extrusion*, *loft*, *sweep*, etc.).

A arquitectura do Rosetta permite a inclusão de novos *frontends* (linguagens de programação) e *backends* (CADs) sem a necessidade de alterar os outros componentes ou os programas existentes.

4.9.1 Conclusões

Rosetta veio colmatar muitos dos problemas que afectam quem recorre ao Desenho Generativo para gerar modelos geométricos. Um dos problemas mais grave é a portabilidade, ou seja, a impossibilidade de migrar (ou reutilizar) o código desenvolvido para uma certa ferramenta de CAD para outra ferramenta de CAD.

Outro problema prende-se com a impossibilidade de haver uma escolha independente de uma ferramenta de CAD e de uma linguagem de programação, uma vez que, quando se escolhe uma ferramenta de CAD a linguagem de programação é (normalmente) "impingida" e vice-versa.

Contudo, Rosetta apenas fornece funcionalidades que são comuns à maioria das ferramentas de CAD. Esta limitação garante a portabilidade dos programas desenvolvidos, mas não resolve o problema da insuficiência de operadores (de alto nível) de suporte ao Desenho Gerativo.

As linguagens (frontends) implementadas pela ferramenta são linguagens de propósito geral (algumas delas discutidas nas secções anteriores), por isso, os designers continuam obrigados a lidar com linguagens de difícil aprendizagem e que não foram desenvolvidos tendo em vista o Desenho Generativo, como é o caso de linguagens como JavaScript, Python, Racket, por exemplo.

5 Proposta de Tese

Depois da análise dos trabalhos relacionados, isto é, linguagens de programação e ferramentas desenvolvidas (ou utilizadas) no contexto do Desenho Generativo, chegou-se à conclusão de que a fim de se conseguir

[22] RosettaRacket é uma extensão da linguagem Racket que facilita a iniciação da aprendizagem do Rosetta e inclui a definição de novas formas sintácticas.

ambientes para Desenho Generativo que sejam: (1) apelativos (2) de fácil aprendizagem e utilização (3) expressivo. Esses ambientes de desenvolvimento devem[23]:

1. Definir/Incluir operadores geométricos de alto nível que permitam criar e manipular formas geométricas de forma intuitiva e eficaz;
2. Definir/Incluir linguagens de programação com sintaxe e semântica adequada a prática do Desenho Generativo e ao público-alvo.

Sendo essa a *Proposta de Tese*.

Dado as características do Rosetta (secção 4.9), o trabalho a ser desenvolvido nesta tese de mestrado utilizará a ferramenta como ponto de partida, ou seja, o Rosetta será estendido tendo em vista a *Proposta de Tese*.

5.1 Operadores geométricos a definir/incluir

Nesta secção apresenta-se os operadores geométricos de alto nível que considera-se essencial definir/incluir num ambiente de desenvolvimento (ou linguagem de programação) para Desenho Generativo que se pretenda que tenha as características definidas na secção 5.

Estes operadores foram inspirados nas varias linguagens de programação analisadas e foram escolhidos tendo em vista a expressividade e utilidade de cada um. A validação desta lista de operadores junto de arquitectos, designers e comunidade do Desenho Generativo em geral corresponde a uma das primeiras tarefas a serem executadas na segunda parte deste trabalho.

5.1.1 Quote

Este operador foi inspirado no operador *QUOTE* presente na linguagem PLaSM (secção 4.1.1.3) e permite a construção de polígonos a partir da especificação de uma sequência de números inteiros. Os inteiros positivos são utilizados para criar segmentos e os negativos são utilizados para criar intervalos vazios ("saltos").

5.1.2 Skeleton

Este operador permite extrair o "esqueleto" de dimensão d de uma certa forma geométrica de uma dimensão D (d < D). O operador retorna os seguintes constituintes de uma forma geométrica:

- Vértices
- Vértices e arestas
- Vértices, arestas e faces

Conforme d (dimensão) seja: 1, 2, 3, respectivamente. Este operador foi inspirado no operador *Skeleton* do PLaSM (secção 4.1.1.7).

5.1.3 Struct

O operador *Struct* foi inspirado no operador com o mesmo nome da linguagem PLaSM (secção 4.1.1.6) e permite a criação de formas geométricas a partir da assemblagem de formas geométricas mais simples.

[23] A estas propriedades juntam-se os princípios por detrás da arquitectura do Rosetta (ver secção 4.9).

5.1.4 Product

Permite a criação de novas formas geométricas a partir de formas geométricas originais onde cada ponto da nova forma geométrica é determinado pelo produto cartesiano dos pontos das formas geométricas originais. O operador foi inspirado no operador * (Produto) da linguagem PLaSM.

5.1.5 Posicionamento relativo

Os operadores: *LEFT*, *RIGHT*, *TOP*, *BOTTOM*, *FRONT*, *BACK* (correspondendo a posicionar: a esquerda, a direita, em cima, em baixo, a frente e atrás, respectivamente), permitem efectuar o posicionamento relativo de objectos. Estes operadores foram inspirados nos operadores de posicionamento relativo do PLaSM (secção 4.1.1.11).

5.1.6 To2Face

Este operador foi inspirado no operador *Poly2DoubleFace* da linguagem GML (secção 4.2.1.1) e permite converter um polígono numa figura geométrica de duas faces, ou seja, faz o preenchimento do espaço entre as arestas de um polígono.

5.1.7 ConnectFaces

Este operador permite, dadas duas faces de um sólido ou dois sólidos, criar faces que unem as arestas das faces do(s) sólido(s) correspondentes. Este operador foi baseado no operador *Bridgerings* da linguagem GML (secção 4.2.1.2).

5.1.8 Extrude

O operador *extrude* é o operador que dado um polígono (ou figura geométrica) de perfil e uma curva guia, efectua uma "varredura" ao longo da curva utilizando o polígono (ou figura geométrica), resultando essa "varredura" num novo polígono (ou figura geométrica). Este operador foi baseado no operador *Extrude* da linguagem GML (secção 4.2.1.3).

5.1.9 CurveByPoints

A operação CurveByPoints permite a criação de curvas a partir da especificação de um conjunto de pontos por onde a curva deve passar. A operação foi inspirada pelo operador *NurbsCurve.ByPoints* da linguagem Dynamo (secção 4.8.1.3).

5.1.10 SurfaceByLoft

A operação SurfaceByLoft permite a criação de superfícies a partir da especificação de um conjunto de curvas (ou linhas) que servem de base para a superfície. A operação foi inspirada pelo operador *Surface.Loft* da linguagem Dynamo (secção 4.8.1.4).

5.1.11 SurfaceByRevolve

A operação SurfaceByRevolve permite a criação de superfícies através da varredura com uma curva em torno do eixo central, resultando naquilo a que se chama uma *superfície de revolução*. Este operador foi influenciado pelo operador *Surface.ByRevolve* do Dynamo (secção 4.8.1.4).

5.1.12 Outros operadores e mecanismos de programação

Além dos operadores de alto nível descritos nesta secção, os ambientes de desenvolvimento / linguagens de programação para Desenho Generativo devem suportar as operações básicas do domínio, como é o caso das transformações geométricas, operadores de mudança de coordenadas, operações booleanas, construção de formas geométricas, etc.

Linguagens que guardam estado (Processing, por exemplo), apresentam formas sintácticas mais compactas, permitido por exemplo, especificar um conjunto de pontos e em seguida invocar uma operação que irá "consumir' esses pontos. Este mecanismo facilita a descrição de programas.

5.2 Linguagem de programação a definir/incluir

Definir uma linguagem expressiva, de fácil aprendizagem e utilização é um dos objectivos deste trabalho. A linguagem será baseada numa linguagem existente e incluirá os operadores definidos na secção 5.1. Novas formas sintácticas ou variações serão decididas recorrendo ao processo de prototipagem (Smith, 1991), ou seja, para definir/estender a sintaxe da linguagem apresentar-se-á várias alternativas a pessoas que recorrem ao Desenho Generativo para gerar modelos geométricos, bem como para obter dados de suporte a decisões.

5.3 Arquitectura da solução proposta

Na figura 5.3.1 apresenta-se a arquitectura da solução proposta[24]. À arquitectura original do Rosetta acrescentou-se dois módulos (sombreados): *RLang* (de *Rosetta Language*) e *RLib* (de *Rosetta Library*), correspondendo a unidades de implementação dos novos operadores de alto nível e da nova linguagem, respectivamente.

O módulo *RLib* corresponde a uma biblioteca e disponibilizará o conjunto de operadores geométricos definidos em 5.1. Este módulo recorrerá ao módulo *core* e aos backends de cada CAD para implementar os novos operadores geométricos, operadores esses que estarão disponíveis para serem utilizados nos programas para os novos e actuais frontends (linguagens) do Rosetta.

O módulo *RLang* irá ser implementado à semelhança dos actuais frontends do Rosetta, ou seja, recorrendo ao módulo *core*. Tal como os actuais e futuros frontends do Rosetta, nos programas para o *RLang* poder-se-á utilizar os operadores definidos no módulo *RLib*.

6 Metodologia de Avaliação do Trabalho

A avaliação dos novos operadores geométricos (secção 5.1) e da nova linguagem de programação (secção 5.2) efectuar-se-á através de um estudo comparativo entre:

1. O Rosetta original e o Rosetta estendido;

[24] Alguns módulos do Rosetta não foram representados, como é o caso do módulo de suporte ao Python e o módulo de suporte ao Maya.

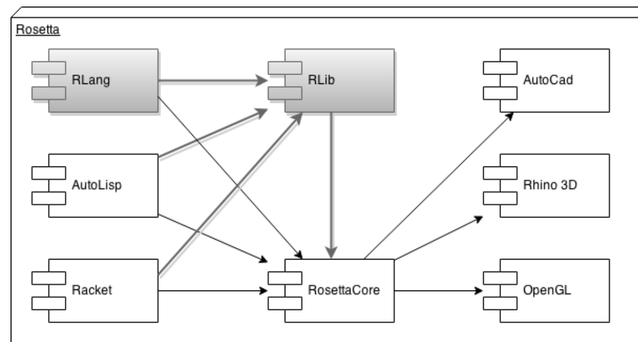


Figura 5.3.1: Arquitectura da solução proposta.

2. O Rosetta estendido e outras ferramentas / linguagens para Desenho Generativo.

As métricas a utilizar serão:

1. Complexidade final da descrição de um modelo geométrico (programa), ou seja, esforço necessário para perceber a descrição (código) de um modelo geométrico;
2. Tempo necessário para perceber e descrever (programar) um modelo geométrico;
3. Número de linhas de código necessários para representar um modelo geométrico;
4. Alterações necessárias no código para adaptar o modelo geométrico a novos requisitos.

Sempre que for possível e se considerar necessário, envolver-se-ão designers e arquitectos, tanto na avaliação deste trabalho como na recolha de informações para tomada de decisões, uma vez que são eles o nosso público-alvo.

7 Conclusões

Neste trabalho apresentou-se um estudo das principais linguagens de programação e ambientes de desenvolvimento para Desenho Generativo. Linguagens específicas do domínio, como é o caso do PLaSM e do GML e linguagens de propósito geral utilizados em Desenho Generativo, como é o caso do AutoLisp e do RhinoScript. Para cada linguagem fez-se uma análise dos operadores geométricos definidos e dos aspectos positivos e negativos da linguagem. O problema da maioria das linguagens de programação apresentadas prende-se com o facto de serem de difícil aprendizagem e utilização para pessoas com reduzida formação em programação e/ou não apresentarem mecanismos de suporte ao Desenho Generativo (operadores geométricos de alto nível, por exemplo).

Neste trabalho definiu-se também uma lista de operadores geométricos e uma linguagem de programação que os ambientes de programação para Desenho Generativo devem implementar por forma a facilitar a aprendizagem e a utilização, bem como para que possa ser atractivo e ter a expressividade desejada. Para validar essa proposta estender-se-á a ferramenta Rosetta (secção 4.9) para incluir os operadores e a nova linguagem de programação, sendo o resultado final comparando com a versão original do Rosetta e com as outras linguagens de programação apresentadas na secção 4.

Este relatório documenta a primeira parte de uma tese de mestrado cujo objectivo é apresentar soluções para os problemas que enfrentam os designers e arquitectos que recorrem a métodos computacionais para gerar modelos geométricos. Na segunda parte deste trabalho estender-se-á o Rosetta para incluir a *Proposta de Tese* apresentada na secção 5 e proceder-se-á à validação dessa proposta segundo a metodologia descrita na secção 6.

Referências

Adobe (1999). *PostScript Language Reference*. Number 3.

Arie Deursen, P. K. (2002). Domain-specific language design. requires feature descriptions. *Journal of Computing and Information Technology - CIT 10*, 1(1–17).

Autodesk (2012). *AutoLISP Developer's Guide*. Autodesk.

Autodesk (xxxx). *Dynamo Language Manual*. Autodesk.

Cabecinhas, F. A. (2010). A high-level pedagogical 3d modeling language and framework.

Casey Reas, B. F. (2014). *Processing: A Programming Handbook for Visual Designers and Artists*. MIT Press.

Charles Eastman, K. W. (1979). Geometric modeling using the euler operators.

College, D. (1964). *A manual for BASIC, the elementary algebraic language designed for use with the Dartmouth Time Sharing System*. Dartmouth College.

Graphisoft (2004). *GDL Reference Guide*. Graphisoft.

Havemann, S. (2003). Introduction to the generative modeling language.

James Foley, A. v. D. (1982). Fundamentals of interactive computer graphics.

John Backus, John Williams, E. W. (1989). Fl language manual, parts 1 and 2. *IBM Research Report*.

John Backus, John Williams, E. W. (1990). An introduction to the programming language fl. *Research topics in functional programming*.

Krause, J. (2003). Reflections: The creative process of generative design in architecture.

Leitão, A. M. (2014). Improving generative design by combining abstract geometry and higher-order programming. *Rethinking Comprehensive Design*.

Lopes, J. (2012). Modern programming for generative design.

Moses, J. (1970). The function of function in lisp, or why the funarg problem should be called the environment problem.

Nicholson-Cole, D. (2000). *The GDL Cookbook, The source of all that is good in GDL*. Number 3. Marmalade Graphics, Nottingham.

Paoluzzi, A. (2003). *Geometric Programming for Computer-Aided Design Alberto Paoluzzi*. John Wiley.

Rutten, D. (2007). *Rhinoscript 101 for Rhinoceros 4.0*. Robert McNeel and Associates.

Smith, M. (1991). *Software Prototyping: Adoption, Practice and Management*.

A Apêndice: Plano de execução da Tese

Tasks	Sub Tasks	Jul 2015	Aug 2015	Sep 2015	Oct 2015	Nov 2015	Dec 2015	Jan 2016	Feb 2016	Mar 2016	Apr 2016	May 2016
RLib Development												
	Operators Validation											
	Operators Implementation											
	Operators Tests											
RLang Development												
	Sintaxe Definition											
	Implementation											
	Tests											
Documentation												