SIGRADI2018
TECHNOPOLITICAS
xxii congresso da sociedade iberoamericana de gráfica digital
22th conference of the iberoamerican society of digital graphics
07|08|09|novembro|2018
iau usp | são carlos | sp br

Language design for modelling: a cognitive approach

Gustavo Henrique Montesião de Sousa
Centro Universitário Belas Artes | Brazil | gustavo@claroenigma.com.br

Abstract

Programming languages have traditionally being designed or chosen to be used in modelling systems with little care to what should be its central concern: the model, and its relation to the programmer's body, to her self,to her being in the world, causing frustration to the student trying to learn the basics of programmatically modelling. This work presents an alternative approach for the design of a language for modelling, aimed to mitigate some of the cognitive barriers normally found in traditional systems.

Keywords: Computational design; programming languages; cognition.

INTRODUÇÃO

A cognição humana pode ser modelada satisfatoriamente como uma máquina de correlações estatísticas, na qual os estímulos corporais de um indivíduo ganham um significado na medida que são correlacionados com um *corpus* de imagens interiores, construído por ele a partir dos estímulos prévios recebidos durante a vida (Clark, 2013). Ou seja, uma imagem formada a partir de um conjunto de estímulos sensoriais é interpretada pelo seu sistema cognitivo e ganha um significado de acordo com sua correlação estatística com o conjunto de imagens armazenadas ao longo das experiências vivenciadas por ele. Tais imagens são sempre compostas por um conjunto total de estímulos — táteis, olfativos, auditivos… — e não meramente por estímulos visuais (Slater, Lotto, Arnold, & Sanchez-Vives, 2009).

Isto significa que uma nova informação, quando vira conhecimento — ou seja, quando o sistema cognitivo lhe atribui um significado —, está sempre associada a experiências prévias de um indivíduo, experiências que são sempre mediadas pelo corpo. Daí a importância de se transmitir uma informação em um formato que encontre ressonância na vivência do receptor, e de um receptor que existe somente mergulhado no mundo, interagindo com o ambiente em redor, como um corpo entre corpos, e não como uma máquina de manipulação de símbolos.

Contudo, isso tende a não ser levado em consideração quando se escolhe uma linguagem de programação para ser usada dentro de um ambiente de modelagem computacional, o que cria uma grande barreira cognitiva dentro do ensino e aprendizagem de programação, como se verá mais adiante. Tende-se a tratar a escolha da linguagem como meramente técnica ou fruto da preferência pessoal do desenvolvedor do sistema, quando na verdade deveria ser vista pelo que é: uma escolha de políticas educacionais e culturais, pois escolhas descuidadas terminam por afastar a maioria dos estudantes universitários de Arquitetura ou Design da área de Design Computacional, relegando-os à periferia das transformações tecnológicas por que estamos passando, por mais que a modelagem por computador tenha se tornado uma ferramenta onipresente em seu dia-a-dia de trabalho.

É comum argumentar que o programador deve aprender a adequar seus modelos mentais ao modo de trabalho da linguagem ou do *framework* que utiliza. Contudo, esse argumento desconsidera o fato de que tal linguagem só exige um determinado modo de trabalho porque foi concebida desde seu projeto para funcionar assim. Por trás de toda linguagem de programação e do modo como ela é usada, existe uma visão de mundo dada pelo projetista dessa linguagem, e é essa visão que dita os modos de pensar necessários para sua utilização. De fato, ao longo da breve história das linguagens de programação, diversos foram os paradigmas alternativos que surgiram para o projeto das linguagens, como se verá adiante, de maneira que um modo único de pensar não é característica intrínseca e comum de todas as linguagens, nem muito menos da programação em si. E, se há maneiras de se projetar uma ferramenta de modo a melhor adequá-la ao operador, não existe razão por que forçá-lo a aprender formas contraintuitivas de se operar uma ferramenta mal projetada para ele.

A seguir, veremos alguns dos principais paradigmas de programação existentes e como eles se relacionam com nosso sistema cognitivo, sobretudo no que tange os processos de modelagem. Serão elencados alguns dos problemas dos paradigmas tradicionais e se proporá um paradigma alternativo, o das linguagens declarativas, que será ilustrado por uma biblioteca para modelagem computacional no SketchUp chamada *Curvas*, desenvolvida pelo autor.

PARADIGMAS DAS LINGUAGENS DE PROGRAMAÇÃO

O projeto de uma linguagem de programação nunca é arbitrário, mas sempre elaborado a partir de alguma metáfora de mundo que lhe confere coesão e significado.

Os paradigmas que servem de substrato para o projeto das linguagens de programação permitem que as separemos em alguns grandes grupos, como o das linguagens imperativas e o das linguagens funcionais.

As linguagens imperativas são as que estão mais amplamente difundidas no mundo, em particular por causa do seu largo emprego na indústria de *software*. A elas pertencem linguagens como C, Javascript, Python e Processing — que, na realidade, é um conjunto de bibliotecas para a linguagem Java. Tais linguagens espelham, em maior ou menor grau, o modo de funcionamento de um computador, que é baseado na manipulação de octetos em diferentes regiões de memória e de registradores do processador.

Um outro paradigma, que vem ganhando corpo recentemente na indústria, é o das linguagens funcionais, ao qual pertencem linguagens como LISP, Erlang e Haskell. Em vez de funcionarem como abstrações sobre o modo como o computador opera, definindo um problema em termos de sequências de operações em regiões de memória, as linguagens funcionais veem um programa como uma descrição matemática do problema a ser resolvido.

Dentro do ramo das linguagens funcionais, temos as linguagens de programação visuais, baseadas em diagramas de fluxo de dados, como o Puredata e o Grasshopper.

Dado que todo projeto de linguagem é guiado, explícita ou implicitamente, por uma metáfora de mundo, e diante do que vimos na seção anterior sobre a cognição humana, essa metáfora deverá encontrar correlação na vivência do programador caso se queira que o seu uso seja o mais imediato e "à mão" (Zahorik & Jenison, 1998, pp. 83–84) possível. Portanto, uma vez que o nosso desejo é usar a programação como uma ferramenta para a modelagem computacional, é importante que a linguagem se relacione de alguma maneira com o modelo que se quer gerar.

As linguagens imperativas sendo, no geral, projetadas a partir das linguagens de máquina, possuem uma metáfora de mundo que apresenta baixa correlação com a vivência do estudante e, sobretudo, com o modelo tridimensional que se quer gerar.

As linguagens funcionais, embora baseadas num modelo mais familiar para o estudante, a matemática, tampouco espelham o modo como ele naturalmente concebe um modelo, sua espacialidade e volumetria.

Nessa mesma linha, um caso particular é o das linguagens baseadas em diagramas. Das linguagens mencionadas até aqui, estas talvez sejam as que possuam geralmente a metáfora mais clara para se depreender o encadeamento lógico de um programa. Contudo, é preciso ter claro que o objetivo último de um sistema de modelagem não é obter uma boa compreensão da organização lógica de um código, mas sim construir um modelo tridimensional de maneira programática. E, para tal fim, tampouco um fluxograma é uma metáfora adequada, uma vez que a forma mais imediata de se visualizar um sólido complexo é pensá-lo como uma composição de geometrias mais simples organizadas espacialmente, e não como uma sequência de operações lógicas.

Dito de outra forma: é mais imediato analisar um sólido complexo pelo que ele é — "este sólido é um conjunto desta, dessa e daquela geometria dispostas desta e daquela maneira" — do que como resultado de uma cadeia de procedimentos — "para se obter este sólido, faça isto, então aplique a seguinte transformação e, por fim, aplique esta operação". Ainda assim, os sistemas de modelagem computacional tendem a utilizar linguagens que nos forçam a pensar desta última forma em vez da primeira, comportamento que não encontra amparo na vivência do modelador e dificulta o estabelecimento de significados por seu sistema cognitivo.

Como se vê, precisamos de uma ferramenta que permita *descrever* um sólido mais do que *comandar* as etapas da sua modelagem. Este será o assunto da próxima seção.

LINGUAGENS DECLARATIVAS

Linguagens que permitem programar um sistema descrevendo-o em vez de emitindo comandos sequenciais existem há um bom tempo, embora seu uso não seja popular entre os programadores em geral. As próprias linguagens funcionais já são, por si mesmas, uma guinada rumo ao espectro das linguagens declarativas, mas há aquelas que foram projetadas especificamente com essa finalidade.

Um dos casos de maior sucesso é o do Prolog (Clocksin & Mellish, 2012), criado em 1972, que permite programar um sistema descrevendo-o a partir de restrições em forma de cláusulas lógicas. É o mesmo princípio encontrado em programas de modelagem via resolução de restrições, como o SolveSpace e o FreeCAD.

Um outro exemplo, criado em 2009 pela Nokia, é o QML (Rischpater & Zucker, 2010, Chapter 6; Ryannel & Thelin, 2015), que fornece uma API declarativa sobre as classes do *framework* Qt, escrito em C++. Com ele, é possível escrever em C++ a parte de seu sistema que demande maior desempenho, e expor esse código como elementos em QML para serem compostos de maneira descritiva e simples.

Sobre esse paradigma, foi desenvolvida a biblioteca *Curvas*, que será apresentada a seguir para ilustrar as propostas trazidas por ela para lidar com os problemas cognitivos descritos na seção anterior.

A BIBLIOTECA *CURVAS*

Embora as extensões do SketchUp sejam programadas usando a linguagem Ruby, que pertence à família das linguagens imperativas, essa linguagem possui algumas ferramentas que permitem o desenvolvimento de pequenas sublinguagens especializadas, o que é tecnicamente referido como DSL — do inglês *domain specific language*. Utilizando essa funcionalidade, a biblioteca *Curvas* (Sousa, 2018) foi escrita como uma camada declarativa sobre a API imperativa do SketchUp.

Em sua forma mais simples, ela é basicamente um conjunto de funções que encapsulam detalhes da API

original, tornando seu uso mais simples e direto. Por exemplo, a criação de um cubo é feita da seguinte forma:

```
edge = 1.m
box width: edge, length: edge, height: edge
```

Em contraste, isso teria que ser escrito da seguinte forma na API original:

```
edge = 1.m
x = 0.5*edge
y = 0.5*edge
z = 0.5*edge
a = [x, -y, z]
b = [-x, -y, z]
c = [-x, y, z]
d = [x, y, z]
square = Sketchup.active_model.entities.add_face(a,b,c,d)
square.pushpull edge
```

A diferença de se usar uma linguagem declarativa começa a aparecer quando se pretende aplicar transformações aos sólidos:

```
edge = 1.m
box width: edge, length: edge, height: edge,
    position: [0,0,3.m], rotation: [30.degrees, 0, 0]
```

Em contraste, de maneira imperativa ficaria:

```
edge = 1.m
x = 0.5*edge
y = 0.5*edge
z = 0.5*edge
a = [x, -y, z]
b = [-x, -y, z]
c = [-x, y, z]
d = [x, y, z]

entities = Sketchup.active_model.entities
square = entities.add_face(a,b,c,d)
cube = square.pushpull edge
group = entities.add_group(cube)
translation =
    Geom::Transformation.translation [0,0,3.m]
rotation = Geom::Transformation.rotation [0,0,0],
    [1,0,0], 30.degrees
group.transform! rotation*translation
```

Note a diferença fundamental na maneira como se atua em cada caso — para além da diferença no tamanho do código: no primeiro caso, simplesmente descreve-se o que o sólido *é* — um cubo com 1 m de lado na posição (0, 0, 3 m) e rotacionado 30° ao redor do eixo das abcissas; já no segundo caso, é preciso dizer *como construir* o cubo através de uma sequência de operações ordenadas no tempo — execute um add_face, seguido de um pushpull, um add_group e um transform!, nesta ordem.

Quando a finalidade da programação é gerar um modelo 3D, qualquer característica do código que force o programador a desviar o foco da estrutura do modelo para pensar em processos tende a ampliar o esforço cognitivo necessário à execução da modelagem, uma vez que reduz a correlação entre os símbolos manipulados no código e o que, no mundo físico, corresponderia ao objeto a ser modelado e, portanto, distancia esse código mais e mais das experiências vivenciadas pelo modelador como um ser inserido no mundo.

ABSTRAÇÕES E FUNÇÕES DE ORDEM SUPERIOR

Como *Curvas* é uma DSL construída sobre o Ruby, todas as funcionalidades dessa linguagem podem ser usadas. Composições de objetos e abstrações nascem imediatamente da própria linguagem:

```
def cube edge
  box width: edge, height: edge, length: edge
end

def plate rails, position
  sphere rails: rails, position: position
  disk rails: 1.7*rails, position: position
end

plate rails: 5.m, position: [5.m, 3.m, 2.0.m]
plate rails: 3.0.m, position: [-2.m, 4.m, 1.5.m]
cube edge: 1.m
```

Uma característica interessante do Ruby é o uso que ele faz de clausuras, que são funções anônimas que capturam o contexto no qual elas foram criadas. Essa funcionalidade foi trazida das linguagens funcionais via uma linguagem chamada SmallTalk, que serviu de inspiração para o Ruby e, assim como nas linguagens funcionais, seu uso extensivo no Ruby traz grande expressividade à linguagem, permitindo escrever conceitos complexos de forma simples. Por exemplo, eis como se podem gerar 10 cubos:

```
edge = 1.m

10.times do |n|
  box width: edge, length: edge, height: edge,
      position: [2*n*edge, 0, 0]
end
```

No exemplo acima, o código contido entre o *do* e o *end* é uma função anônima de um argumento — o argumento *n*. Na nomenclatura do Ruby, tais funções anônimas são chamadas *blocos*. Esse bloco é executado 10 vezes e a cada vez o valor do argumento *n* é alterado, começando em zero e indo até 9. O bloco é capaz de enxergar as variáveis em voga no momento de sua criação — neste exemplo, a variável *edge* — e, assim, capturar esse contexto para quando o bloco for executado. Contraste isso com a forma tradicional de interação em linguagens imperativas — o laço, como um *for* ou um *while* — e verá como se alcança uma concisão e uma clareza muito maiores desta maneira.

Não só a concisão e a clareza aumentam, como também se expande a capacidade de se expressarem conceitos abstratos de forma direta e simples. Uma curva matemática pode ser expressa de maneira precisa sem

dificuldades em uma única linha de código. Por exemplo, para se gerar uma parábola — que é definida pela curva matemática f(x) = x² —, basta fazer:

```
curve length: 2m, height: 1m do |x| x**2 end
```

Nesse exemplo, o bloco é chamado uma certa quantidade de vezes — 100 vezes por padrão — e, a cada chamada, o argumento *x* do bloco recebe um valor que varia de maneira crescente entre -1 e 1. Ou seja, a curva gerada será aquela correspondente a f(x) = x², -1 ≤ x ≤ 1. O resultado devolvido pelo bloco é, então, multiplicado pelo valor informado no argumento *height*, que, neste exemplo, é 1 metro.

Uma espiral é, igualmente, um conceito que pode ser expresso em uma linha:

```
rotate_curve length: 5m, rails: 1m do |t| t end
```

Usando um misto de funções declarativas e blocos, é possível expressar qualquer relação de modelagem algorítmica, embora a implementação atual não contenha por enquanto uma API para algoritmos complexos, como os de sistemas generativos. Ainda assim, a experiência de outras linguagens declarativas, como o QML, mostra que algoritmos arbitrariamente complexos podem ser expostos segundo uma API declarativa para construir sistemas tão diversos quanto interfaces gráficas — o nicho original do QML —, modelagem 3D (Qt, 2018) e jogos eletrônicos (Google, 2014).

4 PRESERVAÇÃO DE ESTRUTURAS

Expressar relações de pertença é possivelmente a área na qual uma API declarativa leva mais ampla vantagem sobre outras. Considere que se queira criar um grupo contendo um quadrado e um círculo, todo ele rotacionado a 45° ao redor do eixo das ordenadas. Usando a biblioteca *Curvas*, tem-se:

```
group rotation: [0, 45.degrees, 0] do
  rectangle width: 2m, length: 2m
  disk rails: 40cm, position: [0, 2m, 0]
end
```

Em contraste, usando a API imperativa do SketchUp,

```
group = Sketchup.active_model.entities.add_group
entities = group.entities
entities.add_face [-1.m, -1.m, 0], [1.m, -1.m, 0],
   [1.m, 1.m, 0], [-1.m, 1.m, 0]
circle = entities.add_circle [0, 2.m, 0], [0,0,1], 40.cm
entities.add_face circle
group.transform! Geom::Transformation.rotation [0,0,0],
   [0,1,0], 45.degrees
```

Note como, no segundo caso, é necessário um esforço de interpretação da lógica do código para perceber que o círculo e o quadrado pertencem ao grupo. Já no primeiro caso, a relação de pertença salta aos olhos, refletida pela estrutura do próprio código, sem necessidade de um esforço cognitivo tão grande. Isso fica ainda mais evidente quando há mais níveis de pertença, como nos exemplos abaixo:

```
group do
  polygon sides: 8, radius: 50cm

  group position: [15m, 0, 0] do
    rectangle width: 2m, length: 2m
    disk radius: 40cm, position: [0, 2m, 0]
  end
end
```

E, comparativamente,

```
external_group =
  Sketchup.active_model.entities.add_group
external_entities = external_group.entities
external_entities.add_ngon [0,0,0], [0,0,1], 50cm, 8
internal_group = external_entities.add_group

internal_entities = internal_group.entities
internal_entities.add_face [-1m,-1m,0],
  [1m,-1m,0], [1m,1m,0], [-1m,1m,0]
circle = internal_entities.add_circle [0,2m,0],
  [0,0,1], 40cm
internal_entities.add_face circle
internal_group.transform!
  Geom::Transformation.translation [15m,0,0]
```

Veja, agora, um exemplo considerando uma combinação mais complexa de elementos:

```
group do
  grid amount: [5, 5, 1], gaps: [1.m, 1.m, 0] do
    mirror do
      rectangle width: 20.cm, length: 20.cm,
        position: [sqrt(2)*10.cm, 0, 0],
        rotation: [0, 0, 45.degrees]
    end
  end

  surface do
    curve length: 5.m, height: 20.cm,
      position: [0, 0, 3.m] do |x| sin(2*PI*x) end
    curve length: 5.m, height: 20.cm,
      position: [0, 5.m, 3.m] do |x| sin(2*PI*x + PI/2) end
  end
end
```

Note como o código preserva a *estrutura* do modelo gerado. Ao olhar o modelo tridimensional, poderíamos bem descrevê-lo como "um grupo, dentro do qual há um *grid* 5x5 e uma superfície; dentro do *grid*, temos o espelhamento de um quadrado rotacionado a 45°; já a superfície é composta por duas curvas senoidais fora de fase". Ou seja, há uma transcrição direta da estrutura do modelo para a estrutura do código. Uma tal característica não é observada em outros paradigmas, nem mesmo nos baseados em diagramas, como o Grasshopper, os quais nos forçam a pensar no modelo como uma sucessão de operações lógicas, impondo uma barreira cognitiva a mais, uma etapa extra indireta de transcrição do modelo para o código.

A dificuldade em acompanhar a relação de pertença em uma API imperativa aumenta ainda mais quando, por alguma razão, o fluxo do código precisa ser interrompido

no meio das operações de inserção. Para ilustrar essa afirmação, suponha que queiramos criar um *grid* bidimensional de círculos cujos raios aumentam na mesma proporção de sua distância ao começo do *grid*. Vamos considerar, por simplicidade, que nossa API imperativa tenha uma função chamada *new_grid*, que nos gera um *grid* ao qual adicionamos objetos através do método *add*. Então, o código ficaria semelhante a este:

```
grid = new_grid[5,5], [1cm,1cm,0]
entities = Sketchup.active_model.entities

for i in 0..5
  for j in 0..5
    grid.add(i, j, entities.add_circle [0,0,0], [0,0,1],
      20.cm + sqrt(i*i + j*j))
  end
end
```

A função *new_grid* criará um objeto *grid* que será o responsável por reposicionar os círculos adicionados a ele. Os círculos são adicionados ao *grid* informando em que casa do *grid* eles deverão ficar — a posição *i,j*.

Já utilizando a biblioteca *Curvas*, isso fica assim:

```
grid.anon! [5,5], gaps: [1cm,1cm,0] do |i,j,k|
  circle radius: 20.cm + 2*sqrt(i*i + j*j)
end
```

INDEPENDÊNCIA TEMPORAL

A discussão anterior aponta ainda para uma última característica fundamental das linguagens declarativas: a independência temporal. Por ser uma descrição de um sistema, o código em uma linguagem declarativa tende a não apresentar uma dependência do tempo: não existe uma ordem temporal fixa na qual cada linha ou comando precise ser executado para que o código produza o resultado correto. Em vez disso, cada linha se relaciona com todas as outras ao mesmo tempo para compor uma descrição única, que é válida em sua totalidade e não linha-a-linha.

Essa característica é fundamental para que o processo de modelagem via programação aconteça a partir da percepção do que o modelo é, e não a partir do que ele se tornará após aplicar uma dada cadeia de operações lógicas.

CONCLUSÕES

A escolha de uma linguagem para se usar em um sistema de modelagem computacional não pode ser feita sem reflexão. Tampouco deve ser guiada pela popularidade de uma linguagem nesse nicho — como por exemplo, o uso de Python ou de linguagens baseadas em fluxogramas. Tal escolha precisa ser feita sempre no âmbito de uma política educacional, pois impacta diretamente no grau de dificuldade que um estudante terá para aprender a usar esse sistema. E não somente na aprendizagem, como também no esforço cognitivo despendido por profissionais, por mais habituados que estejam com a linguagem, pois o sistema cognitivo é o mesmo nos dois casos: seja o de um estudante, seja o de um programador profissional.

É necessário, portanto, dar mais atenção à linguagem escolhida, porque o fim último de se usar uma linguagem de programação nesses sistemas é gerar um modelo: é o modelo que importa, é ele o objetivo, não o algoritmo, não o processo, que são meios para se atingir esse objetivo.

Como nosso sistema cognitivo atribui significado às coisas a partir da correlação de imagens, das quais o corpo todo participa, e não através de manipulações simbólicas, a associação entre o modelo e o código deve ser a mais direta possível, de modo a se criarem menos barreiras para a aprendizagem.

Para tanto, as linguagens declarativas fornecem uma ferramenta mais adequada do as alternativas de que dispomos comercialmente, pelas seguintes características: focam e nos fazem pensar no que o sistema é, e não no processo; preservam estruturas, que funcionam como pontes entre o mundo real e o mundo simbólico da linguagem; e possuem independência temporal.

A implantação da biblioteca Curvas, apresentada aqui, visou a explorar e ilustrar esses conceitos dentro de um ambiente de modelagem muito popular, o SketchUp.

A escolha do SketchUp para criar essa biblioteca se deveu a dois fatores: popularidade do uso dessa ferramenta entre arquitetos e, particularmente, estudantes de arquitetura; e o uso que ela faz do Ruby como linguagem de *script*. O uso do Ruby se mostrou particularmente valioso, devido a duas características fundamentais suas:

- a possibilidade de se criarem DSL, o que permitiu desenvolver uma API declarativa de alto nível sobre a API imperativa do SketchUp;
- a possibilidade de se programar usando funções de ordem superior, que foi crítico para se criarem funções como a *curve* e o *grid*; sem essa característica, tais funções seriam muito mais complexas e menos intuitivas.

Vê-se que, por essas características, seria muito mais difícil criar uma biblioteca como a *Curvas* para programas que usam linguagens de *script* como o Python, tais como o Blender e o FreeCAD. Contudo, não há nada nesta proposta que esteja univocamente atrelada ao uso do SketchUp, tendo sido este programa escolhido, como dito acima, sobretudo pela conveniência de se poder explorar rapidamente os conceitos apresentados neste artigo. APIs declarativas podem ser, em princípio, desenvolvidas para quaisquer programas de modelagem, em particular os de maior sucesso comercial atualmente.

Desta maneira, vislumbra-se como trabalho futuro estudar a possibilidade de se implantar uma solução declarativa para programas de modelagem como o Blender, e por uma série de razões:

- o Blender é um programa de modelagem muito mais robusto e avançado que o SketchUp;

- ele é um *software* livre, diferentemente do SketchUp, que é proprietário e possui uma licença dispendiosa;

- a API Ruby do SketchUp é mal elaborada, e diversas funcionalidades são desnecessariamente difíceis de serem implementadas.

Uma possibilidade talvez seja criar uma interface de programação declarativa dentro do Blender que não envolva o uso do Python, mas para isso faz-se necessário um maior estudo do problema.

REFERÊNCIAS

Clark, A. (2013). Whatever next? Predictive brains, situated agents, and the future of cognitive science. Behavioral and brain sciences, 36(3), 181–204.

Clocksin, W. F., & Mellish, C. S. (2012). Programming in Prolog: Using the ISO standard. Springer Science & Business Media.

Google. (2014). The VoltAir Project. Recuperado de http://google.github.io/VoltAir/doc/main/html/index.html

Qt. (2018). Qt 3D Overview. Recuperado de https://doc.qt.io/qt-5/qt3d-overview.html

Rischpater, R., & Zucker, D. (2010). Introducing Qt Quick. In Beginning Nokia Apps Development (p. 139–158). Springer.

Ryannel, J., & Thelin, J. (2015). Qt5 Cadaques.

Slater, M., Lotto, B., Arnold, M. M., & Sanchez-Vives, M. V. (2009). How we experience immersive virtual environments: the concept of presence and its measurement. Anuario de psicología, 40(2).

Sousa, G. H. M. (2018). Curvas. Recuperado de https://gitlab.com/gustavo-hms/curvas

Zahorik, P., & Jenison, R. L. (1998). Presence as being-in-the-world. Presence, 7(1), 78–89.