UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS DE CRATEÚS

CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FRANCISCO ALEX SOUSA ANCHIÊTA

MODELAGEM PROCEDURAL DE ELEMENTOS DE FACHADA PARA ARQUITETURAS COMPLEXAS UTILIZANDO *SELECTION EXPRESSION* (*SELEX*)

CRATEÚS

2023

FRANCISCO ALEX SOUSA ANCHIÊTA

MODELAGEM PROCEDURAL DE ELEMENTOS DE FACHADA PARA ARQUITETURAS COMPLEXAS UTILIZANDO *SELECTION EXPRESSION* (*SELEX*)

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação do Campus de Crateús da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da Computação.

Orientador: Prof. Me. Arnaldo Barreto Vila Nova

CRATEÚS

2023

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

A554m Anchieta, Francisco Alex Sousa.
Modelagem procedural de elementos de fachada para arquiteturas complexas utilizando Selection Expression (SELEX) / Francisco Alex Sousa Anchieta. – 2023.
110 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Crateús, Curso de Ciência da Computação, Crateús, 2023.
Orientação: Prof. Me. Arnaldo Barreto Vila Nova.

1. Modelagem procedural. 2. Modelagem de fachadas. 3. Selection Expression. 4. Ambientes urbanos. I. Título.

CDD 004

FRANCISCO ALEX SOUSA ANCHIÊTA

MODELAGEM PROCEDURAL DE ELEMENTOS DE FACHADA PARA ARQUITETURAS COMPLEXAS UTILIZANDO *SELECTION EXPRESSION* (*SELEX*)

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação do Campus de Crateús da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da Computação.

Aprovada em:

BANCA EXAMINADORA

Prof. Me. Arnaldo Barreto Vila Nova (Orientador)
Universidade Federal do Ceará (UFC)
Campus de Crateús

Prof. Dr. Markos Oliveira Freitas
Universidade Federal do Ceará (UFC)
Campus de Russas

Prof. Dr. Roberto Cesar Cavalcante Vieira
Universidade Federal do Ceará (UFC)
Departamento de Arquitetura e Urbanismo e Design

À minha família, por acreditar em mim em todos os momentos desta caminhada.

AGRADECIMENTOS

À minha mãe, Lucimar Sousa Anchieta, e ao meu pai, Raimundo Anchieta Vieira, pelo apoio incondicional em todas as minhas escolhas e pelo estímulo em investir na educação, desde cedo, como forma de crescimento moral e pessoal. Aos meus irmãos, pelo companheirismo e o auxílio nos momentos difíceis.

Aos amigos, que compartilharam comigo todos os momentos desta jornada, nos trabalhos em grupo, nos momentos descontraídos, estando presente nos momentos de alegria e de tristeza. À eles, sou muito grato.

Ao meu orientador, Prof. Arnaldo Barreto Vila Nova, pela dedicação e auxílio na elaboração deste trabalho, sendo sempre paciente e compreensivo nos momentos de dúvidas.

Aos demais professores, por repartir seu conhecimento com todos, com empenho e alegria, de modo que nos inspira a buscar sempre mais conhecimento.

Ao Doutor em Engenharia Elétrica, Ednardo Moreira Rodrigues, e seu assistente, Alan Batista de Oliveira, graduado em Engenharia Elétrica, pela adequação do *template* utilizado neste trabalho para que o mesmo ficasse de acordo com as normas da biblioteca da Universidade Federal do Ceará (UFC).

E à todos que, de alguma forma, me ajudaram no decorrer do meu percurso. Que todos possam ser abençoados de todas as formas.

"O conhecimento emerge apenas através da invenção e da reinvenção, através da inquietante, impaciente, contínua e esperançosa investigação que os seres humanos buscam no mundo, com o mundo e uns com os outros."

(Paulo Freire)

RESUMO

A modelagem procedural de ambientes virtuais tornou-se de grande relevância nos últimos anos, principalmente devido às suas vantagens em relação ao processo manual de modelagem. Em vez de um *designer* construir um objeto manualmente, o que pode ser muito custoso e demorado, é possível definir alguns parâmetros que caracterizam o objeto, de modo que seja gerado automaticamente. Nesse sentido, a geração de construções é um dos tópicos mais avançados no campo da modelagem procedural. Isso ocorre devido a importância de representar os ambientes urbanos, que reproduzem o contexto vivenciado pela sociedade e caracteriza a cultura e identidade de um povo. Existem diversas técnicas que buscam gerar esses elementos da maneira mais fidedigna possível, mas a maioria delas limita-se em estruturas retas e retangulares. Porém, muitas construções possuem estruturas mais complexas, com deformações, como as encontradas em construções com arquitetura orgânica. Em relação às fachadas, apresenta ainda mais adversidades, uma vez que elas devem se adequar corretamente a construções multiforme e também podem possuir mais detalhes distintos. Com isso, o presente trabalho busca tornar possível a geração procedural de elementos de fachadas em construções com geometria arredondada, utilizando *selection expressions* (*SELEX*). Para isso, foram implementadas operações descritas na linguagem *SELEX* que auxiliam no manipulação das formas de construção. Além disso, foram desenvolvidas novas operações que permitem adicionar elementos de fachadas, inclusive com curvatura.

Palavras-chave: Modelagem procedural. Modelagem de fachadas. *Selection expression*. Ambientes urbanos.

ABSTRACT

Procedural modeling of virtual environments has become highly relevant in recent years, mainly due to its advantages over the manual modeling process. Instead of a designer building an object manually, which can be costly and time-consuming, it is possible to define certain parameters that characterize the object so that it can be generated automatically. In this sense, the generation of buildings is one of the most advanced topics in the field of procedural modeling. This is due to the importance of representing urban environments, which reproduce the context experienced by society and characterize the culture and identity of a people. There are several techniques that aim to generate these elements as faithfully as possible, but most of them are limited to straight and rectangular structures. However, many buildings have more complex structures with deformations, such as those found in constructions with organic architecture. Regarding facades, there are even more challenges, as they must properly adapt to multifarious structures and may also have distinct details. Therefore, this study aims to enable the procedural generation of facade elements in buildings with rounded geometry using selection expressions (SELEX). To achieve this, operations described in the SELEX language that assist in manipulating the building shapes have been implemented. Additionally, new operations have been developed to add facade elements, including curved ones.

Keywords: Procedural modeling. Modeling of facades. selection expression. Urban environments.

LISTA DE FIGURAS

LISTA DE CÓDIGOS-FONTE

LISTA DE ABREVIATURAS E SIGLAS

L-Systems	*Lindenmayer Systems*
3D	Três dimensões
CGA	*Computer Generated Architecture*
SELEX	*Selection Expressions*
fpm	*Floor Plan Module*
pp-connection	*point-point-connection*
ee-connection	*edge-edge-connection*
UI	*User Interface*
2D	Duas dimensões
API	*Application Programming Interface*
bpy	*Blender Python*

SUMÁRIO

1 INTRODUÇÃO

1.1 Contextualização

Nas últimas duas décadas, os mundos virtuais 3D avançaram de modelos rudimentares para ambientes complexos de alta imersão. No entanto, o processo de modelagem, ferramentas e técnicas utilizadas para gerá-los não avançaram muito: são trabalhosos e repetitivos em uso e requerem habilidades especializadas em modelagem 3D. A modelagem procedural tem sido um tópico de pesquisas ativo por pelo menos trinta anos (SMELIK *et al.*, 2009).

As técnicas procedurais têm sido utilizadas ao longo da história da computação gráfica. Muitas das primeiras técnicas de modelagem e texturização incluíram descrições procedurais de geometria e coloração de superfície. Desde então, elas explodiram em um paradigma importante e poderoso de modelagem, texturização e animação (EBERT *et al.*, 2002).

A filosofia da modelagem procedural é, em vez de projetar ambientes manualmente, criar um procedimento que gere o conteúdo automaticamente. Essa abordagem foi aplicada com sucesso para gerar, por exemplo, texturas, modelos geométricos, animações e até clipes de sons. Um tópico importante na modelagem procedural é a geração automática de modelos de terreno, que começou com fenômenos naturais, como elevação de terreno e crescimento de plantas nas décadas de 1980 e 1990, e estendeu seu foco para ambientes urbanos no início do novo milênio (SMELIK *et al.*, 2009).

Com o uso de métodos de geração procedural, pode ser criado conteúdo ilimitado para jogos, cenas completas e complexas para filmes, reconstruir e visualizar as partes que faltam em sítios arqueológicos, fornecer a visualização de várias possibilidades em simulações e fornecer a designers sugestões iniciais de projetos arquitetônicos (COGO *et al.*, 2019).

Para o desenvolvimento de jogos, um dos principais componentes é o design de conteúdo. É por meio desse elemento que são gerados os objetos e ambientes que irão interagir com o jogador. A qualidade da geração do conteúdo impacta diretamente no custo do projeto e a sua produção muitas vezes requer uma equipe composta por diversos *designers* 3D. A geração do ambiente, por exemplo, exige a criação de uma área principal, onde o *player* passará a maior parte do tempo, e das áreas periféricas, que são locais que não podem ser visitados pelo *player*. Se o arredores não forem tão importante, o processo de criação desse conteúdo pode levar a perda de tempo para a equipe. Dessa forma, o tempo investido nessas áreas poderia ser melhor aplicado na área principal. Nesse sentido, o uso de técnicas procedurais é considerado como uma

potencial solução para a criação de conteúdo. Além disso, mesmo que a geração procedural crie um ambiente completo que pode ser usado sem modificação, também pode ser usado como um ponto de partida para a equipe de *design* (CARLI *et al.*, 2011).

A geração de cidades virtuais, com o crescimento da disponibilidade de dados sobre áreas urbanas, tem impulsionado pesquisas e o desenvolvimento de softwares que realizam a simulação desses ambientes. Simular os ambientes urbanos é uma tarefa importante em um grande número de aplicações, por exemplo: no planejamento urbano, para coordenar mudanças nos espaços urbanos ou visualizar o impacto de novas construções; na geração detalhada de ambientes virtuais, como cidades inteiras de maneira rápida e realística; e na logística, em situações onde a representação fidedigna de um espaço urbano é importante, como em treinamento de socorro de evacuação, mapeamento de tráfego, entre outras (RODRIGUES *et al.*, 2015).

Dessa forma, observa-se que a aplicação de técnicas procedurais para a modelagem de elementos gráficos é amplamente difundida, uma vez que as suas vantagens são de grande importância para as mais diversas áreas da sociedade, tanto para fins comerciais como acadêmicos.

1.2 Justificativa

A criação de modelos arquitetônicos de forma procedural pode diminuir significativamente os custos de modelagem, uma vez que permite gerar uma variedade de formas semelhantes a partir de uma descrição do que será modelado (EDELSBRUNNER *et al.*, 2017). Isso também vale para a geração de fachadas, onde os seus componentes podem ser posicionados por meio das regras que a especificam.

Porém, algumas das técnicas existentes lidam apenas com estruturas retas, de forma que construções com geometria mais complexa, com formas arredondadas por exemplo, são adicionadas apenas por meio de importações de modelos externos (BRITO *et al.*, 2021).

Dessa forma, o presente trabalho busca complementar a pesquisa de Brito *et al.* (2021) e sua implementação, que possibilita a criação de modelos arquiteturais com geometria arredondada, através de *selection-expressions*. À vista disso, a contribuição é na adição da possibilidade de gerar as fachadas dos modelos gerados, incluindo construções com arredondamento.

1.3 Objetivos

1.3.1 Objetivo Geral

Apresentar elementos de fachada de construções em modelos de geometria arredondada utilizando *selection-expression*.

1.3.2 Objetivos específicos

- Estabelecer linguagem para geração de componentes para fachadas, com a aplicação de atributos e restrições;
- Promover a vinculação da linguagem estabelecida com uma ferramenta de modelagem 3D;
- Avaliar a solução elaborada;
- Analisar os resultados obtidos, em comparação com outros trabalhos.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, é desenvolvido o respaldo teórico para o entendimento do que é apresentado pelo presente trabalho, partindo de um conceito amplo e especificando-o até alcançar os relacionados à modelagem procedural de fachadas. Inicialmente, são descritos os fundamentos para a modelagem procedural. Em seguida, são apresentadas técnicas para geração de ambiente urbano, edifícios e de fachadas prediais. Por fim, é apresentada uma introdução ao *SELEX*, técnica abordada por esta pesquisa.

2.1 Modelagem Procedural

A modelagem procedural é o termo geral para definir um conjunto de técnicas que usam regras para representar a geometria, textura e comportamento dos objetos gráficos. O uso da palavra *procedural* advém do fato de que as regras são definidas de forma semelhante a um *script* ou programa (KLAVDIANOS *et al.*, 2013).

Duarte *et al* (2006) descrevem a modelagem procedural como o uso de algoritmos para gerar modelos gráficos, de forma que são criados dinamicamente, diferentemente de modelos gerados manualmente por um *designer*. Dessa forma, é possível gerar modelos gráficos a partir de parametrizações, que determinam características do objeto gráfico para formá-los automaticamente.

Rodrigues *et al.* (2015) apontam que a modelagem procedural pode ser aplicada satisfatoriamente ao representar características como repetições, autossimilaridade e aleatoriedade. Além disso, apresentam uma vantagem em relação à modelagem manual: o seu potencial de escalabilidade. Diversos modelos podem ser gerados a partir de variações de parâmetros e pequenos conjuntos de regras.

Dessa forma, de acordo com Ebert *et al.* (2002), uma das características mais importantes das técnicas procedurais é a abstração. Com uma abordagem procedural, as informações necessárias para a geração de uma cena gráfica estão abstraídas em um algoritmo ou função (ou seja, um procedimento), ao invés de especificá-los e armazená-los explicitamente. Com isso, há uma economia de armazenamento, uma vez que os detalhes não são mais armazenados, sendo gerados implicitamente pela função/algoritmo. Além disso, a responsabilidade de especificar as informações do modelo ou da cena é transferida do programador para a máquina, economizando tempo de modelagem.

Outra característica é o controle paramétrico, que permite atribuir a um parâmetro um conceito significativo, como por exemplo um número que torna as montanhas mais ásperas ou suaves. Este controle liberta o *designer* do controle de baixo nível e da especificação de detalhes. Os modelos procedurais também oferecem flexibilidade, de forma que o design pode capturar a essência do objeto, fenômeno ou movimento sem ser restringido pelos limites do mundo real. Dessa forma, pode ser produzida uma ampla variedade de efeitos, desde a simulação precisa das leis naturais, até efeitos puramente artísticos (EBERT *et al.*, 2002).

Segundo Schwarz e Müller (2015), as técnicas de modelagem procedural são utilizadas em diversas áreas, como planejamento urbano, games e filmes, para criar várias instâncias semelhantes. No caso da geração de edifícios, são normalmente baseadas em gramáticas, onde um conjunto de regras descrevem como uma forma pode ser refinada para um novo conjunto de formas. A partir de uma forma inicial, as regras são aplicadas iterativamente, evoluindo a estrutura do modelo hierarquicamente e, de forma incremental, adicionando detalhes.

Thaller *et al.* (2013) definem uma gramática como um conjunto de regras, que possuem um lado esquerdo e um lado direito. O lado esquerdo especifica a quais formatos a regra se aplica, fornecendo um único rótulo não terminal e, opcionalmente, uma condição. O lado direito define as formas que substituem a forma selecionada; essas formas de substituição são especificadas em termos de operações aplicadas sobre a forma selecionada.

A seguir, serão apresentados os principais tipos de grámaticas utilizadas, sendo elas as *L-Systems*, *shape grammar* e *split grammar*.

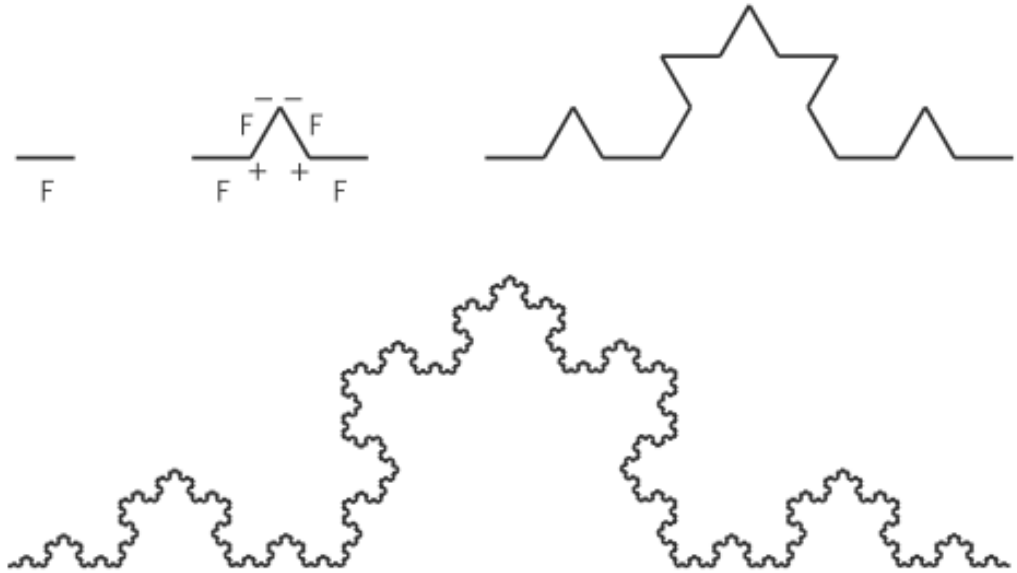
2.1.1 L-Systems

A *L-Systems* é uma notação de gramática de reescrita paralela para a modelagem de desenvolvimento de sistemas biológicos. As gramáticas são similares aos definidos na teoria de linguagem formal definida por Chomsky (1956), exceto que as produções são aplicadas simultaneamente, e não há distinção entre terminais e não-terminais. Todas as strings geradas a partir de um axioma são consideradas palavras em uma linguagem da gramática (SMITH, 1984).

Uma maneira intuitiva de representar as *L-Systems* é utilizando *Turtle Graphics*, onde as *strings* são interpretadas como movimentos e orientações de uma tartaruga em um plano. O símbolo “F” move a tartaruga para frente, o símbolo “+” gira a tartaruga para o sentido anti-horário e o símbolo “-” gira para o sentido horário, por um ângulo definido (EBERT *et al.*, 2002). A Figura 1 apresenta um exemplo da aplicação da regra: $F \rightarrow F+F-F-F+F$,

considerando um ângulo de inclinação de 60º.

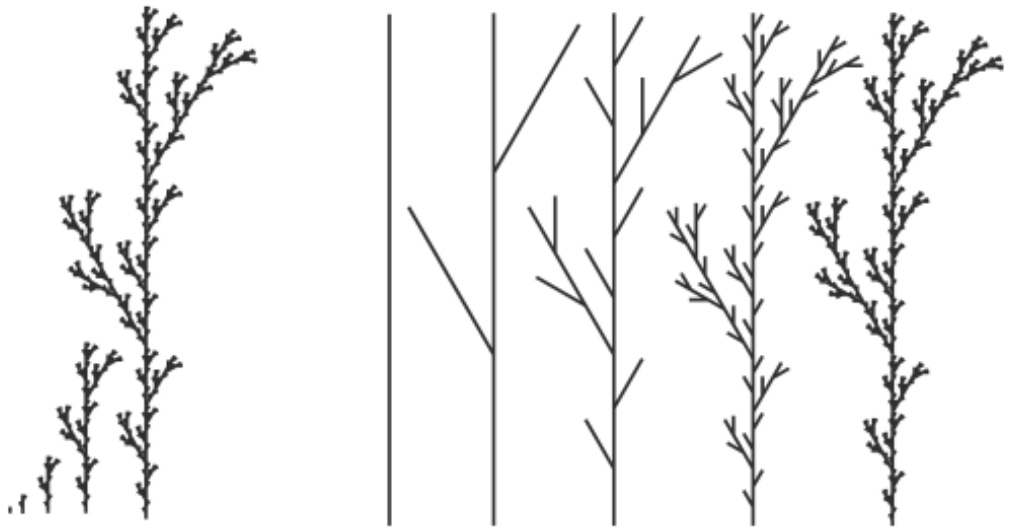
Figura 1 – Representação da curva de Koch



Fonte: Ebert *et al.* (2002).

Conforme demonstrado por Ebert *et al.* (2002), existem extensões que visam adicionar funcionalidades a essa representação. Por exemplo, o símbolo "[" é utilizado para armazenar o estado (posição e orientação) da tartagura, enquanto que o símbolo "]" restaura para um estado previamente armazenado. A Figura 2 apresenta um exemplo de aplicação desses símbolos para a produção: $F \rightarrow F[+F]F[-F]F$.

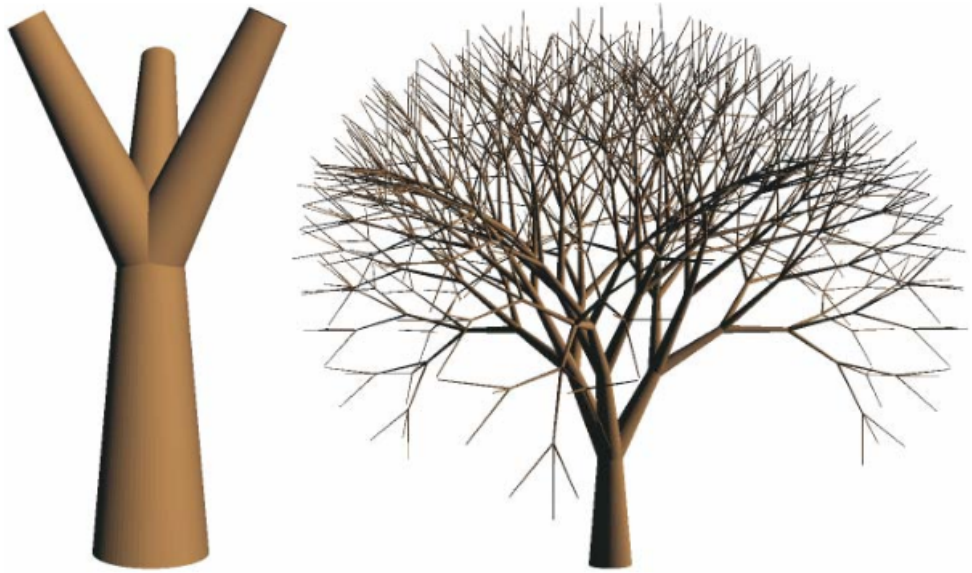
Figura 2 – Aplicação das extensões para armazenamento do estado, para os símbolos "["e "]"



Fonte: Ebert *et al.* (2002).

Ademais, para estender suas funcionalidades em modelos 3D, são adicionados os símbolos "∧" que movimenta para cima e "&" para baixo, além dos símbolos "\" e "/" para definir a rotação. Adicionamente, é possível parametrizar os símbolos, de forma que seja possível cada um apresentar uma distância ou ângulo distinto (EBERT *et al.*, 2002). A Figura 3 demonstra um exemplo de modelos 3D utilizando os simbolos adicionais e o uso de parametrização.

Figura 3 – *L-Systems* aplicados a modelos 3D e o uso de parametrizações.



Fonte: Ebert *et al.* (2002).

2.1.2 Shape grammar

Stiny e Gips (1971) apresentam um formalismo para representar as formas em termos de uma gramática, de modo que o *designer* utiliza um conjunto de regras para gerar os modelos gráficos, semelhante à linguagem formal definida por Chomsky (1956).

Knight (2000) descreve que a *shape grammar* é um conjunto de regras de formas que se aplicam passo a passo para gerar um conjunto, ou linguagem, de *designs*. As *shape grammars* são descritivas e generativas. As regras de uma *shape grammar* geram ou calculam *designs*, e as próprias regras são descrições das formas dos modelos gerados.

Stiny (1980) apresenta as *shape grammars* por quatro componentes:

- Conjunto finito de formas S;
- Conjunto finito de símbolos (rótulos) L;
- Conjunto finito de regras R, da forma $\alpha \rightarrow \beta$, onde α e β são formas rotuladas;
- Uma forma rotulada inicial I.

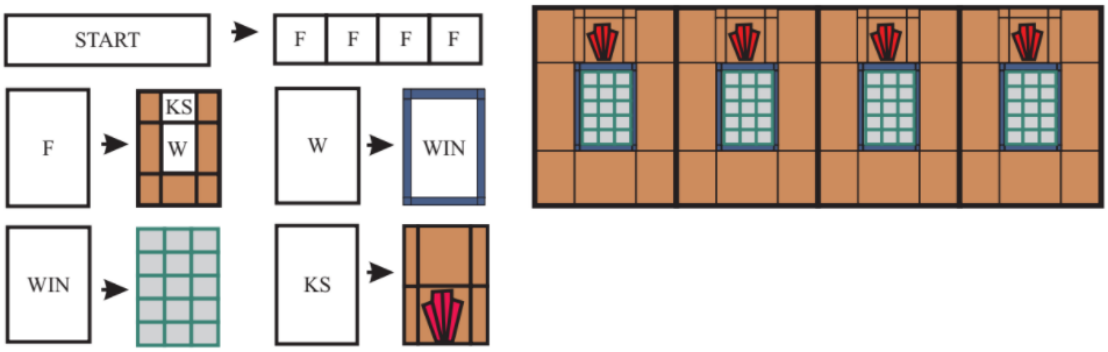
Em uma *shape grammar*, as formas no conjunto S e os símbolos no conjunto L fornecem os blocos de construção para a definição das regras de forma no conjunto R e a forma inicial I. Formas rotuladas geradas usando *shape grammar* também são construídas em termos desses elementos primitivos. As formas nas regras e na forma inicial são rotuladas para ajudar a orientar o processo de geração de formas (STINY, 1980).

As *shape grammars* têm propriedades atribuídas a torná-las especialmente adequadas para projetos, sem sacrificar o rigor formal. Primeiro, os componentes das regras são formas: pontos, linhas, planos ou volumes. As regras de forma geram modelos usando as operações de adição e subtração de forma, além de transformações espaciais familiares aos *designers*, como deslocamento, espelhamento e rotação. Em síntese, as gramáticas de formas são algoritmos espaciais, em vez de textuais ou simbólicos. Em segundo lugar, as gramáticas de forma tratam as formas como entidades não atômicas, ou seja, podem ser decompostas e recompostas livremente. Terceiro, as *shape grammars* não são determinísticas - o usuário de uma gramática de formas pode ter muitas opções de regras e maneiras de aplicá-las em cada etapa (KNIGHT, 2000).

2.1.3 Split grammar

As *split grammars* são gramáticas usadas para a modelagem procedural de construções, concebida por Wonka *et al.* (2003). A partir de uma forma inicial, a gramática gera fachadas da construção que, por sua vez, dividem-se em seus elementos estruturais, até o nível de elementos de design básicos, como janelas, cornijas, portas, etc (WONKA *et al.*, 2003). A Figura 4 ilustra a aplicação das sucessivas divisões.

Figura 4 – Ilustração da derivação de uma fachada gerada com split grammar



Fonte: Wonka *et al.* (2003).

Wonka *et al.* (2003) definem uma *split grammar* como um conjunto de formas básicas B, composta por: uma regra de divisão $a \rightarrow b$, onde a é um subconjunto conexo de B, e b

contém os mesmos elementos de a, exceto por um elemento, por qual será aplicada a divisão; e uma regra de conversão $a \rightarrow b$, onde a é um subconjunto conexo de B que contém uma única forma básica, e b contém os mesmos elementos de a, exceto que a forma básica foi substituída por outra, com a restrição de que a forma básica em b deve estar contida no volume da forma básica em a.

Em outras palavras, a regra de divisão particiona a forma em um conjunto de formas menores, que respeitem o limite da forma original, e a regra de conversão transforma uma forma básica em outra.

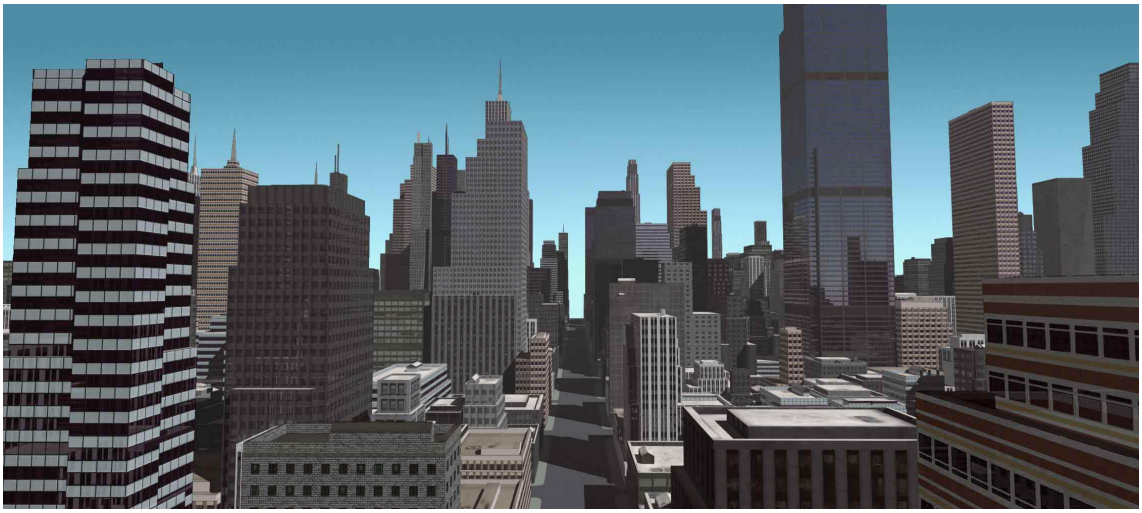
2.2 Modelagem em ambientes urbanos

A geração de modelos de cidades virtuais auxilia na representação das características dos seres humanos. Por meio delas, é observado como um conjunto de pessoas se congregam em um espaço em comum e como se relacionam entre si. A seguir, são apresentados dois modelos para a geração procedural de cidades virtuais.

2.2.1 Procedural modeling of cities

Parish e Müller (2001) apresentam um sistema para a geração procedural de cidades, denominado de *CityEngine*, que é capaz de modelar uma cidade virtual (Figura 5) a partir de um conjunto de informações estatísticas e geográficas. Com o CityEngine, é possível gerar ambientes urbanos do zero, utilizando um modelo hierárquico de regras que podem ser estendidas pelo usuário, de acordo com a necessidade.

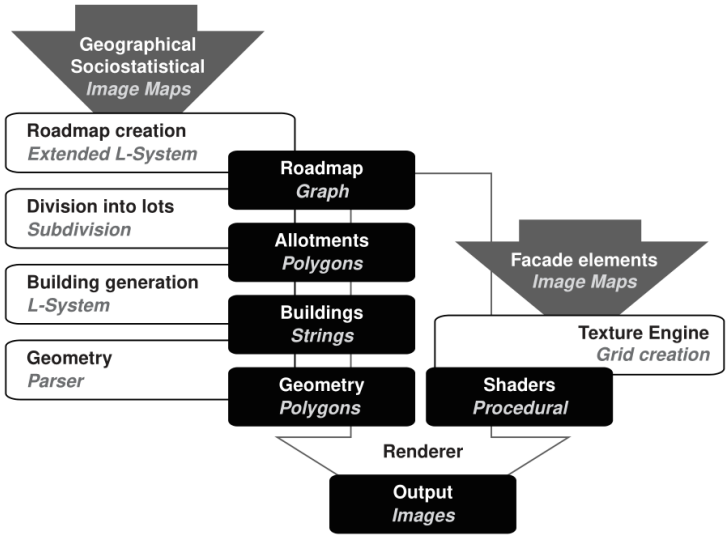
Figura 5 – Ambiente urbano gerado pela *CityEngine*



Fonte: Parish e Müller (2001)

A *CityEngine* é composta por ferramentas que formam os passos necessários para gerar o modelo urbano. No primeiro passo, os dados de entrada são consumidos pelo sistema de geração de ruas, utilizando uma extensão das *L-Systems*, onde as áreas entre as ruas caracterizam as quadras da cidade. Depois, as quadras são subdivididas em lotes, em que serão construídos os edifícios. No próximo passo, as construções são geradas por um outro *L-Systems*, como uma representação em string de operações booleanas em formas básicas. Por fim, os resultados podem ser gerados e visualizados (PARISH; MÜLLER, 2001). A Figura 6 descreve o *pipeline* do *CityEngine*

Figura 6 – *Pipeline* do modelo proposto para geração de cidades virtuais



Fonte: Parish e Müller (2001).

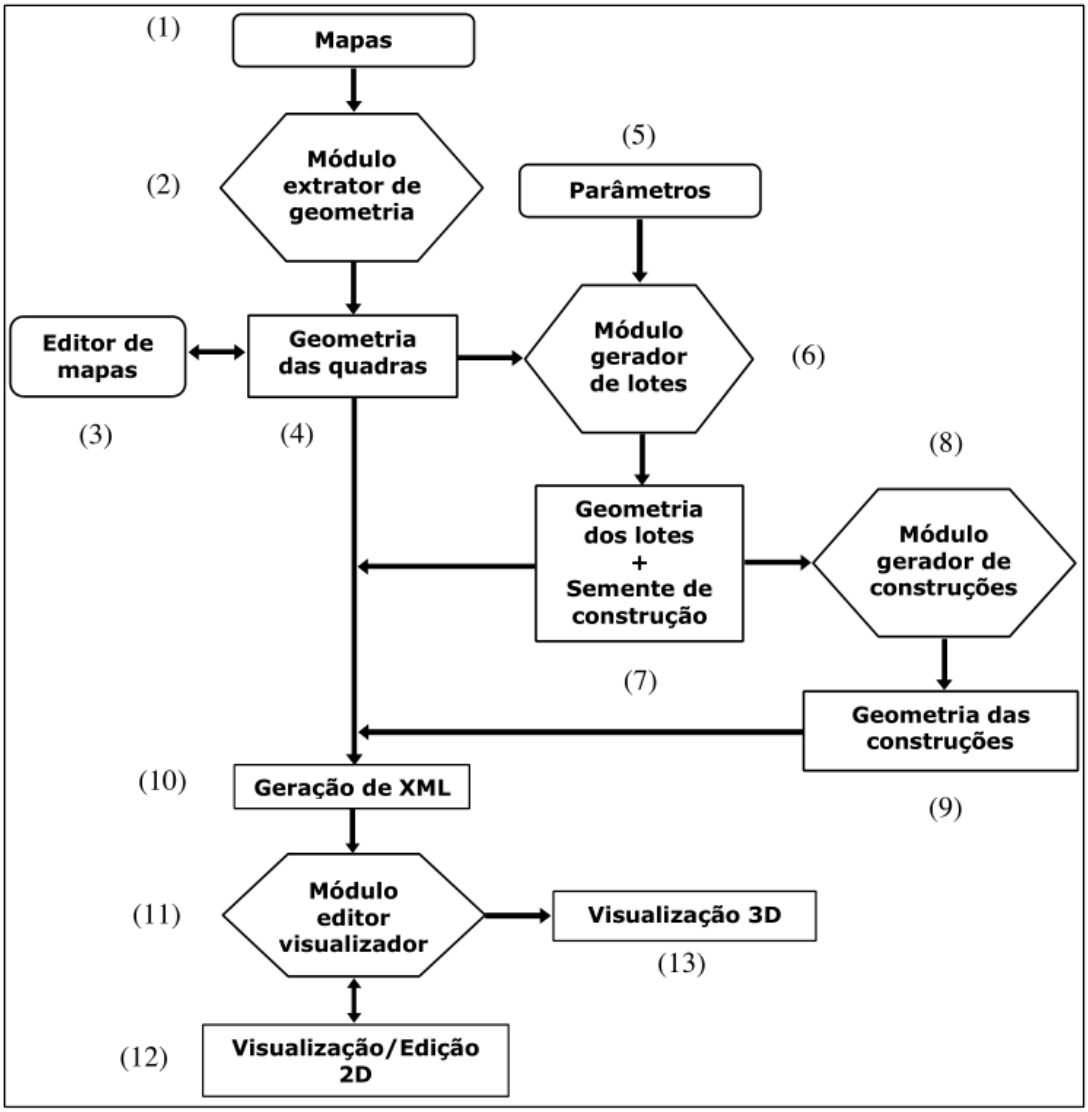
2.2.2 Modelagem procedural de cidades virtuais

Marson *et al.* (2003) descrevem um modelo parametrizado para geração de ambientes urbanos a partir de um mapa 2D como entrada. Para tanto, o sistema é construído utilizando módulos que realizam ações no processo para a geração da cidade virtual.

Inicialmente, o usuário deve fornecer um mapa de uma cidade criada digitalmente, ou mesmo de uma cidade real, de forma que o sistema realize a segmentação do modelo em quadras, obtendo as informações geométricas sobre a topologia da cidade a ser gerada. Tais informações são extraídas a partir do módulo extrator de geometria, utilizando técnicas de processamento de imagem. As especificações da geometria geradas por este módulo podem ser editadas a partir do editor de mapas. Definida a geometria das quadras, é realizada a sua divisão em lotes, por meio do módulo gerador de lotes. Para isso, é necessário que o usuário

defina parâmetros que caracterizam o ambiente com dados estatísticos, como tipo de zona e construção, densidade habitacional, entre outros. Depois, será gerada uma semente, que contém as informações que serão utilizadas pelo módulo gerador de construções. Por fim, é gerado um arquivo com as informações semânticas do modelo, que é utilizado pelo módulo editor/visualizador para visualizar ou editar a posição da geometria 2D no formato planta baixa e visualizar a cidade em 3D (MARSON *et al.*, 2003). A Figura 7 apresenta o modelo descrito.

Figura 7 – Arquitetura do modelo proposto para gerar ambiente virtuais.



Fonte: Marson *et al.* (2003)

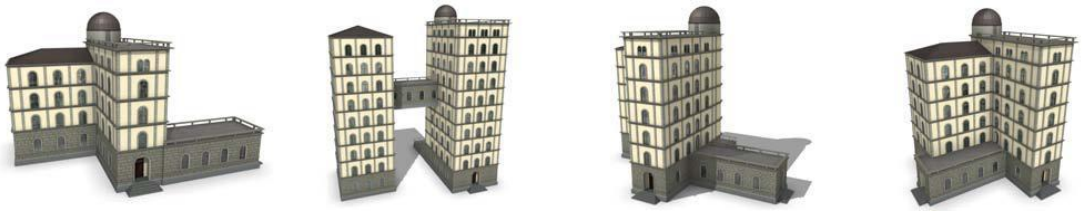
2.3 Modelagem procedural de construções

Segundo Smelik *et al.* (2014), a geração procedural de construções é uma das áreas da modelagem procedural mais desenvolvidas e, como dito outrora, a maioria dos métodos nesta

categoria utilizam gramáticas.

Müller *et al.* (2006) descrevem uma nova gramática de forma denominada CGA *shape*, focada para a geração de modelos de construções detalhadas (Figura 8), com regras de produção que evoluem iterativamente. Essas regras inicialmente criam um modelo básico, denominado modelo de massa, e em seguida adicionam as fachadas, aplicando detalhes como portas, janelas e ornamentos. A principal vantagem dessa abordagem é que a estrutura hierárquica e a anotação do modelo são especificados no processo de modelagem.

Figura 8 – Exemplo de modelos gerados a partir da gramática CGA *Shape*.



Fonte: Müller *et al.* (2006)

A notação da gramática e as regras gerais para adicionar, escalar, transladar e rotacionar formas são inspiradas pelas *L-Systems*, mas são estendidas para o contexto de construções. Enquanto que gramáticas paralelas, como as *L-Systems*, são criadas por um processo de crescimento, uma aplicação linear de regras permite a caracterização da estrutura. Com isso, CGA *Shape* é uma gramática sequencial, semelhante às definidas por Chomsky (MüLLER *et al.*, 2006).

Schwarz e Müller (2015) apresentam uma evolução da CGA *shape*, denominada CGA++. Ela apresenta melhorias em relação ao CGA *shape*, permitindo que as formas fiquem expostas na gramática, de modo que formas individuais podem ser identificadas com exclusividade, bem como transmitidas e armazenadas como valores. Particulamente, as operações podem assumir formas como argumentos, permitindo operações booleanas em várias formas. Além disso, possibilita acessar, percorrer e consultar a árvore de formas, construída no processo de derivação.

Jiang *et al.* (2018) introduziram as *Selection Expressions* (*SELEX*), como uma alternativa para substituir as abordagens baseadas em gramáticas apresentada anteriormente. Com *Selection Expressions*, as formas não são selecionadas através de simples correspondência de strings, mas utilizando expressões que especificam como selecionar um subconjunto de formas de uma árvore de formas. Dessa maneira, as regras passariam da forma *label* $\rightarrow$ *actions* para *selection-expressions* $\rightarrow$ *actions*, ou seja, o *SELEX* propõe uma melhoria no lado esquerdo da

regra. Na Seção 2.5, serão apresentados os conceitos relacionados a essa abordagem.

2.4 Geração procedural de fachadas

A geração de fachadas é um grande desafio para modelagem procedural de modelos arquiteturais. Isso ocorre devido a grande quantidade de elementos que as constituem, como janelas, portas, cornijas, platibandas, ornamentos, entre outros elementos. Além disso, é necessário organizar cada componente de modo que simule construções reais, considerando o alinhamento, simetria e a proporção, por exemplo.

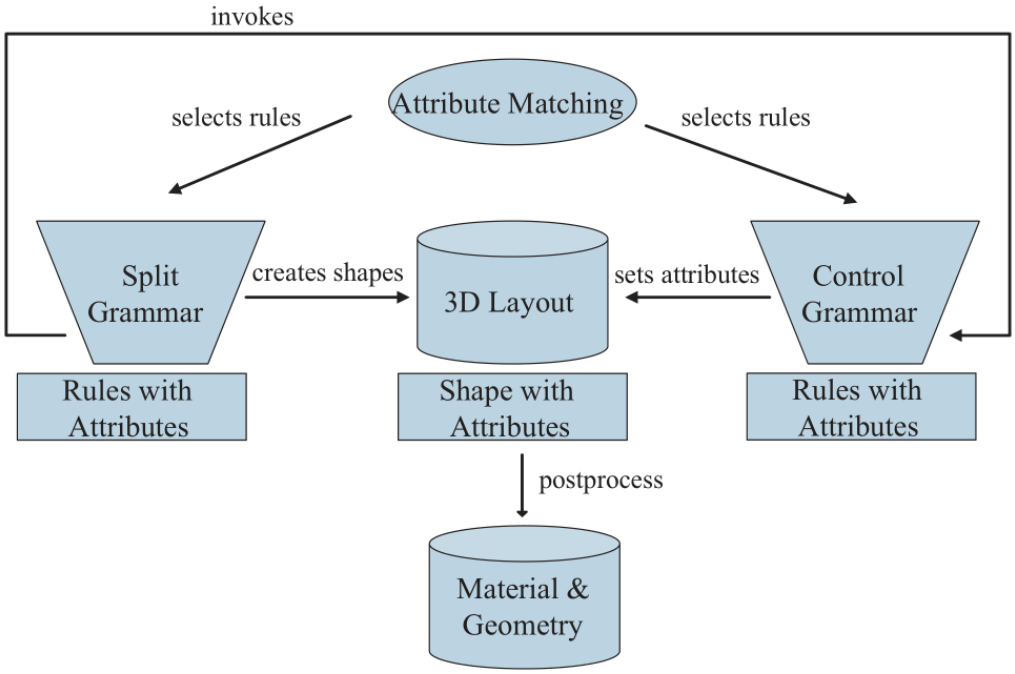
Wonka *et al.* (2003) introduziram um sistema para modelagem de construções (Figura 9), em especial para gerar fachadas com alto nível de detalhes. Esse método constitui-se de três elementos principais. Primeiro, pelas gramáticas denominadas *split grammar*, responsáveis pela geração das fachadas de construções, partindo de formas básicas, sendo divididas em elementos estruturais, como portas e janelas. Segundo, pelas gramáticas de controle, usadas para calcular e distribuir os atributos das formas geradas pela *split grammar*. Por fim, por um sistema de correspondência de atributos, que seleciona uma regra entre várias correspondentes, tanto das gramáticas de controle como das *split grammar*. Sempre que várias regras correspondentes são encontradas em alguma etapa da derivação, os atributos especificados nelas são comparados com os atributos associados ao símbolo gramatical atual e a regra com a melhor correspondência é selecionada. Os atributos associados às regras são especificados durante o projeto de gramática.

Figura 9 – Exemplo de modelo gerado a partir do uso de *split grammar*

Fonte: Wonka *et al.* (2003)

Conforme apresentado por Wonka *et al.* (2003), os atributos são usados para codificar e propagar informações de baixo nível, como um valor de cor específico para paredes, ou de alto nível, como um estilo de construção. Ademais, são utilizados para orientar o processo de derivação, selecionando uma regra específica entre um conjunto de regras de correspondência, por meio de um processo de correspondência de atributos. A Figura 10 demostra a interação cada componente especificado pelo sistema.

Figura 10 – Interação entre as *split grammar*, a gramática de controle e o sistema de correspondência de atributos para gerar um modelo gráfico.



Fonte: Wonka *et al.* (2003)

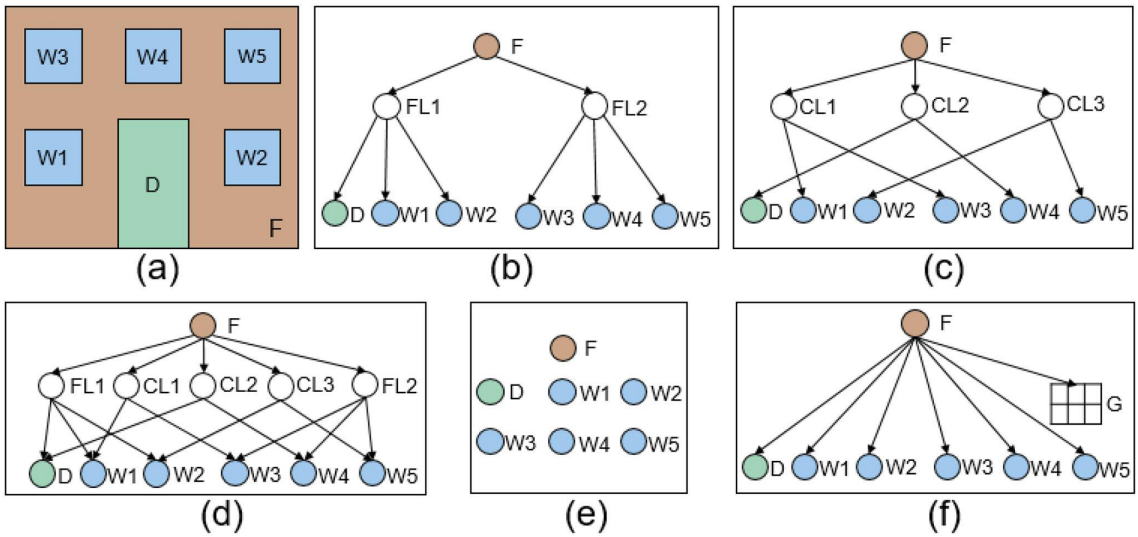
Por fim, Rodrigues *et al.* (2015) introduziram uma técnica que busca otimizar *split grammar* utilizando algoritmos genéticos, de modo que realiza uma busca no conjunto de gramáticas dado como entrada para gerar gramáticas, que por sua vez geram modelos com certas características predefinidas. No Capítulo 3, serão apresentadas outras abordagens para a geração de fachadas.

2.5 Selection Expression

Elaborado por Jiang *et al.* (2018), o *SELEX* é uma linguagem que aplica o conceito de seleções baseadas em expressões, que realizam uma busca na árvore de formas. A seguir, serão apresentados, resumidamente, os conceitos fundamentais das *Selection Expressions*.

Conforme Jiang *et al.* (2018), uma importante decisão de *design* é em como organizar uma coleção de formas desenvolvidas pela linguagem. Uma escolha seria estruturar o conjunto de formas sem nenhuma hierarquia, conforme apresentado na Figura 11(e), porém, é bastante complexo formalizar certas seleções. Outra alternativa é organizar as formas em uma árvore hierárquica, representado nas Figuras 11(b) e 11(c), sendo uma abordagem clássica na modelagem procedural baseada em gramáticas, contudo limita a uma hierarquia particular. Especialmente para fachadas, podem haver múltiplas hierarquias e as operações de modelagem são normalmente expressas em hierarquias distintas. Dessa forma, uma outra solução seria criar e manter múltiplas hierarquias explicitamente, em paralelo, conforme retratado na Figura 11(d). No entanto, isso deve levar a inconsistências, devido à criação de formas intermediárias para agrupar outras formas.

Figura 11 – Diferente escolhas de *design* para representar uma fachada



Fonte: Jiang *et al.* (2018)

Para organizar a coleção de formas produzidas pela linguagem, foram utilizados formas virtuais, de maneira que elas podem gerar quaisquer sub-região como forma auxiliar em tempo real, sem gerar e gerenciar explicitamente a sub-região (a Figura 11(f) apresenta esta abordagem). Pisos e colunas em fachadas são exemplos de sub-regiões que podem ser gerados. Com isso, a forma é definida por dois tipos: formas virtuais, apresentadas acima, e formas de construção, sendo todas as outras formas similares às definidas pela CGA e CGA++, por exemplo (Jiang *et al.*, 2018).

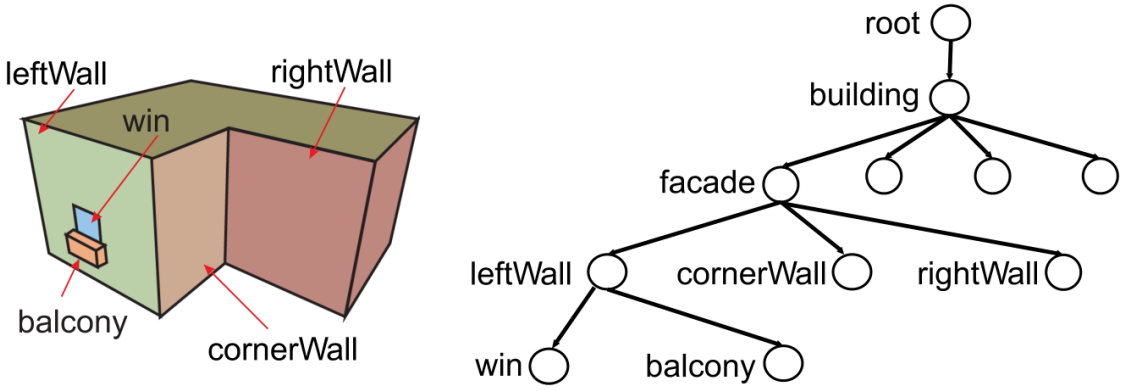
As formas virtuais podem ser posicionadas em formas de construção 2D, que geralmente consistem em várias células, com linhas e colunas. Essas *grids* orientam a disposição

de outras formas de construção, porém não são usadas para dividi-las. São utilizadas em três cenários: para localizar uma posição, facilitar na seleção de formas de construção que estão incluídas dentro deles e auxiliar na maneira que o redimensionamento é realizado (Jiang *et al.*, 2018).

Segundo Jiang *et al.* (2018), as formas possuem um conjunto de atributos que a caracterizam: *label* é o nome da forma (por exemplo, "window"); *type* indica se a forma é virtual ("virtual"), de construção ("construction") ou uma célula ("cell") de uma forma virtual; *dim* especifica se as formas são 2D ou 3D; e o escopo descreve a coordenada, posicionamento e o tamanho da forma, delimitando-a. Além disso, dispõe de informações topológicas em relação à árvore de formas, como a conexão para o pai, uma lista de filhos e uma lista de vizinhos.

As formas são estruturadas em uma hierarquia de formas e armazenadas em uma árvore, como apresentado na Figura 12. Formas são adicionadas na árvore por uma função da linguagem. Cada forma possui apenas um pai e apenas formas de construção podem ter filhos. As formas não podem ser excluídas, mas podem se tornar invisíveis. A raiz da árvore é uma forma com o *label* "*root*" (Jiang *et al.*, 2018).

Figura 12 – Representação de um modelo de uma construção (à esquerda) e a árvore que a representa (à direita).



Fonte: Jiang *et al.* (2018)

De acordo com Jiang *et al.* (2018), a linguagem *SELEX* executa apenas um comando por vez. Esses comandos podem ser regras ou atribuições de variável. Uma regra é da forma *selection-expressions* $\rightarrow$ *actions*, onde *selection-expresssion* é responsável por selecionar uma lista de formas da árvore de formas atual, e *action* são as operações realizadas sobre cada forma da lista. Por sua vez, uma atribuição é da forma *identifier* = *expression*; onde *identifier* indica uma variável e *expression* atribui um valor. A Figura 13 demonstra algumas regras (C2-C11) e

variáveis (C1).

Figura 13 – Exemplos de regras e atributos do *SELEX*

```
##C1:
facW = 17.6; facH = 12.8;
##C2:
{<> -> addShape("facade", ...);}
##C3: split facade into cells and set it as the working
node
{<[label == "facade"]> -> createGrid("main", ...);}
{<[label == "facade"]/[label == main]> -> setHeadNode();}
##C4: add a door touching the ground;
{<cell()[colLabel == "mid"][rowLabel == "gnd"]> ->
addShape("door", ...);}
##C5: add the glass windows above the door
{<cell()[colLabel == "mid"][rowIdx > 1][::groupCols()]> ->
addShape("glass", ...);}
##C6: add ledges at the left side of the top floor.
{<cell()[colLabel == "left"][rowLabel == "top"]
[::groupPairs()]> ->
addShape("ledge", ...);}
##C7: add the two-cell window at the right side the top
floor.
{<cell()[colLabel == "right"][rowLabel == "top"]
[::groupRows()]}> ->
addShape("win4", ...);}
##C8: add windows on the ground floor.
{<cell()[colLabel in ("left", "right")][rowIdx == 1]> ->
addShape("win1", ...);}
##C9: add windows on the second and third floor.
{<cell()[colLabel in ("left", "right")][rowIdx in
rowRange(2, -2)]> ->
addShape("win2", ...);}
##C10: add windows in the left side of the top floor
{<cell()[colLabel == "left"][rowLabel == "top"]> ->
addShape("win3", ...);}
##C11: generate walls to fill the space.
{<root()/[label == "facade"]> -> coverShape();}
```

Fonte: Jiang *et al.* (2018)

As *selection expressions* selecionam uma lista de formas da árvore de forma usando seletores intercalados com o operador "/", que aplica os seletores para cada forma, sequencialmente. Cada seletor recebe uma lista de formas e retorna uma nova lista. Seletores podem ser tipificados de três formas (Jiang *et al.*, 2018):

- **Seletores de topologia** produzem uma lista de formas com uma relação topológica especificada para a forma de entrada. Eles possuem a forma *[topology-*

functions()]. Pode ser utilizados as seguintes funções: "*child()*", "*descendant()*", "*parent()*", "*root()*", "*neighbor()*", e "*contained()*";

- **Seletores de atributos** retornam uma lista de formas cujos atributos satisfazem as condições definidas. Em sua forma básica, são estruturados como *[atribute-name comparison value]*. Por exemplo, a instrução [label = "window"] deve retornar a forma que possui o label nomeado "window";
- **Seletores de grupos** aplicam operações de agrupamento para retornar uma lista de formas combinadas. Elas operam apenas em formas virtuais e reagrupam sub-regiões. Estes seletores são implementados usando funções de agrupamento, da forma *[::grouping-functions()]*. Alguns exemplos de funcões são *groupRows* e *groupColumns*, que agrupa formas virtuais adjacentes, previamente selecionadas.

Conforme especificado por Jiang *et al.* (2018), a *selection expression* é definido como: < *[topoS] [attrS | groupS]* / [topoS] [attrS | groupS]* / ...* >. Ou seja, em cada seletor há zero ou um seletor de topologia (*topoS*), zero ou vários seletores de atributos (*attrS*) ou seletores de grupos (*groupS*). O seletor de topologia deve ser o primeiro, mas a ordem dos seletores de atributos e de grupos podem ser intercalados. Se uma expressão de seleção estiver vazia, será retornada a entrada. Caso uma forma que não existe seja especificada, a seleção retorna uma forma vazia e a regra não será executada.

As *actions* são sequências de funções que são executadas na lista de formas gerada pela aplicação das *selection-expressions*. A *action* mais importante são as funções utilizadas para criar novas formas, como "*addShape*", "*attachShape*", "*coverShape*" e "*connectShape*", por exemplo. Essas funções criam novas formas e as adicionam na árvore. O pai de uma nova forma pode ser especificado explícita ou implicitamente usando valores padrões. Normalmente, o pai é a forma de construção de entrada ou, no caso de uma forma virtual, o primeiro ancestral que é uma forma de construção (Jiang *et al.*, 2018).

Para melhorar localmente o layout, Jiang *et al.* (2018) ainda fornecem ações que podem especificar restrições. Uma lista de restrições é definida pela formato: *constrain(constraint1, constraint2, ...)*. As restrições podem ser compatíveis ou não. Para lidar com conflitos potenciais, é feita uma verificação de compatibilidade. Se nenhum conflito for detectado, será adicionada a restrição no conjunto de restrições. As restrições incompatíveis são removidas. As restrições suportadas pelo *SELEX* são de alinhamento, simetria, distância até os limites e prevenção de interseção. A descrição de todas as funcionalidades oferecidas pela linguagem são disponibilizadas

em material suplementar[1].

2.6 Geração de construções com geometria arredondada utilizando *SELEX*

Uma das limitações do *SELEX*, apontadas por Jiang *et al* (2018), é a impossibilidade da modelagem de formas curvas diretamente, aceitando apenas a sua importação como *assets*. Dessa forma, não é possível modelar a maioria das fachadas curvas.

Brito *et al.* (2021) apresentam uma solução para a limitação na geração de modelos de massa com geometria arredondada, com a criação da *action* denominada *roundShape*, que gera as estruturas arredondadas de um modelo. Essa operação é definida da seguinte forma:

```
roundShape(type, direction, roundingDegree, segments, sideReference, axis,
  insideDegree),
```

onde:

1. **type:** Define o tipo de arredondamento, sendo classificado como “front”, “left”, “right”, “top” ou “bottom”;
2. **direction:** Define a direção do arredondamento, sendo classificado como “outside” ou “inside”;
3. **roundingDegree:** Representa o grau de arredondamento;
4. **segments:** Representa a quantidade de faces que serão criadas no processo de deformação;
5. **sideReference:** Define a direção em que o vetor normal da região está voltado, sendo classificado como “main_front”, “main_back”, “main_left” ou “main_right”;
6. **axis:** Definido para operações de arredondamento, cujo type assume o valor “front”, sendo classificado como “vertical” ou “horizontal”;
7. **insideRounding:** Define o grau de arredondamento interno.

Com a aplicação desta nova *action* é possível gerar modelos com deformações aplicadas as *selection expressions*. Como exemplo, Brito *et al.* (2021) apresentam a geração de um modelo simples. Inicialmente, são definidas as dimensões do modelo, utilizando a criação de variáveis. Depois, é gerado o modelo de massa com o uso da *action* “*createShape*” (Veja Figura 14(a)). Para selecionar uma sub-região da parte frontal do modelo, é adicionada uma forma virtual com a *action* “*createGrid*” (Veja Figura 14(b)). Com a forma virtual adicionada, pode ser realizada a seleção das células na operação de agrupamento, com o seletor “*groupRegions*”. Para realizar a operação de extrusão da área selecionada, é utilizada a *action* “*addVolume*” (Veja

[1] Disponível em http://peterwonka.net/Publications/pdfs/2018.TVCG.Haiyong.SELEX.additional.pdf.

Figura 14(c)). Por fim, para efetuar o arredondamento na área que foi adicionada o volume, é utilizada a *action* "*roundShape*" (Veja Figura 14(d)). A seguir, é apresentada a descrição do modelo gerado conforme a definição da linguagem.

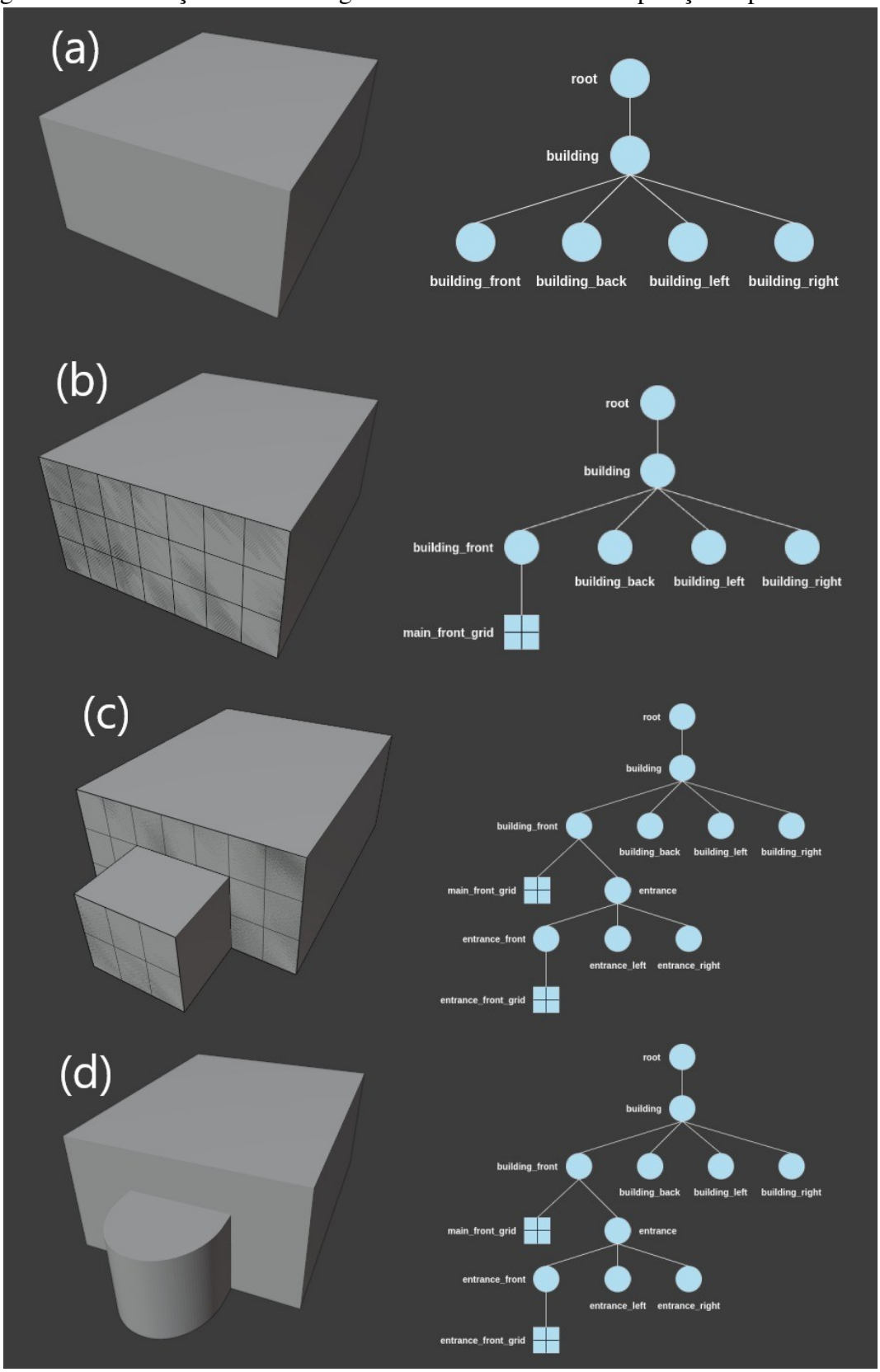
Código-fonte 1 – Descrição de modelo de massa de construção

```
#C1: Initial settings
label = "building"; width = 9; depth = 11; height = 5;

#C2: Generating mass model
{<>    > createShape(label, width, depth, height)};

#C3: Adding virtual shape
{<descendant() [label=="building"] / [label=="building_front"]>    > createGrid("
    main_front_grid", 3, 7)};

#C4: Selecting regions and performing extrusion
{<descendant() [label=="building"] / [label=="building_front"] / [label=="
    main_front_grid"] / [type=="cell"] [rowIdx in (2, 3)] [colIdx in (3, 4, 5)] [::
    groupRegions()]>    > addVolume("entrance", "building_front", 3, ["entrance_front"
    , "entrance_left", "entrance_right"])};

#C5: Applying roundShape deformation
{<descendant() [label=="building"] / [label=="building_front"] / [label=="entrance"] /
     [label=="entrance_front"]>    > roundShape("front", "outside", 0.42, 30, "
    main_front", "vertical")};
```

2.7 Considerações finais

Neste capítulo, foram apresentados os conceitos fundamentais da modelagem procedural e as técnicas precursoras para a geração de modelos utilizando gramáticas. Além disso, foram apresentadas trabalhos que descreve regras para a geração de elementos urbanos. Por fim, foi abordada a linguagem *SELEX* para a geração de construções, e o trabalho de Brito *et al.* (2021), base para o presente trabalho. No próximo capítulo, serão retratadas algumas metodologias para a geração de elementos de fachadas.

Figura 14 – Ilustração do modelo gerado com o decorrer das operações aplicadas.



Fonte: Adaptado de Brito *et al.* (2021)

3 TRABALHOS RELACIONADOS

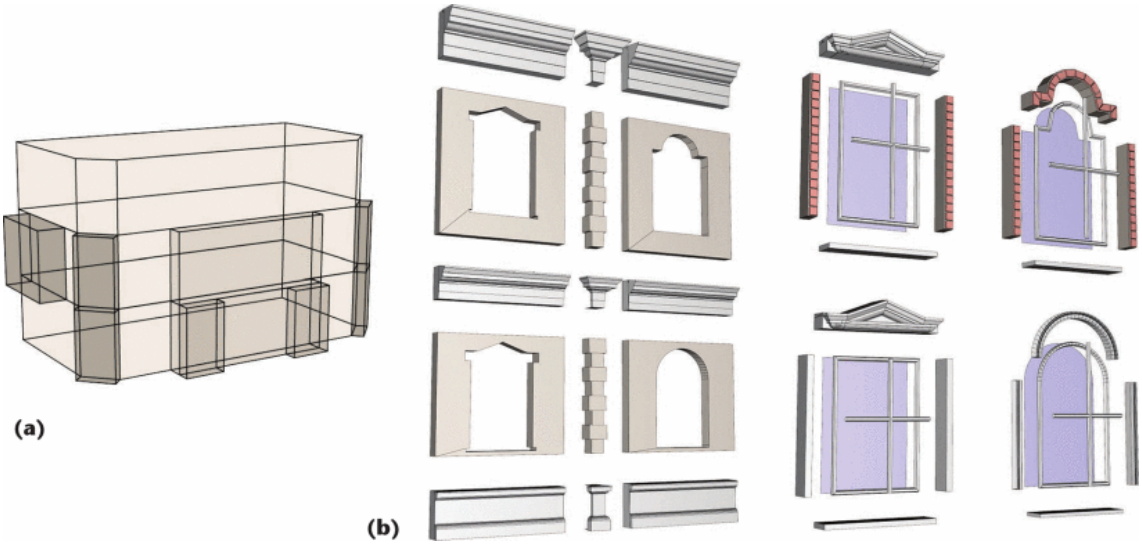
Neste capítulo, são apresentadas, em ordem cronológica, diferentes técnicas para gerar proceduralmente elementos arquiteturais de fachada, expondo suas principais características.

3.1 Detailed Building Facades

Finkenzeller (2008) apresenta uma abordagem para geração de fachadas de construções, de modo que o *designer* fornece informações abstratas, como o esboço da construção, propósito e estilo, e então o computador extrai as informações espaciais para os elementos de fachada no estilo definido e automaticamente identifica e adapta as estruturas adjacentes. O computador gera uma representação semântica da fachada, armazenando-a em uma estrutura hierárquica (ou seja, gera uma parametrização de toda a fachada). Como efeito, os projetistas podem alterar os parâmetros da fachada em alto nível e produzir estruturas mais complexas em menos tempo.

Para modelar um edifício, é traçado um contorno simples, incluindo elementos constituintes da construção, como alas laterais e varandas, e é definida a aparência. Essas informações são armazenadas em um grafo, que inclui as informações geométricas necessárias. A principal vantagem disso é que os usuários podem modificar as informações no grafo e alterar a aparência do edifício em um nível abstrato. Eles possuem a capacidade de atribuir estilos a todo o edifício, a cada piso ou mesmo a cada janela. (FINKENZELLER, 2008).

Figura 15 – Representação das duas partes da modelagem: (a) o esboço do edifício fornecido pelo usuário e (b) os elementos gerados pelo computador.

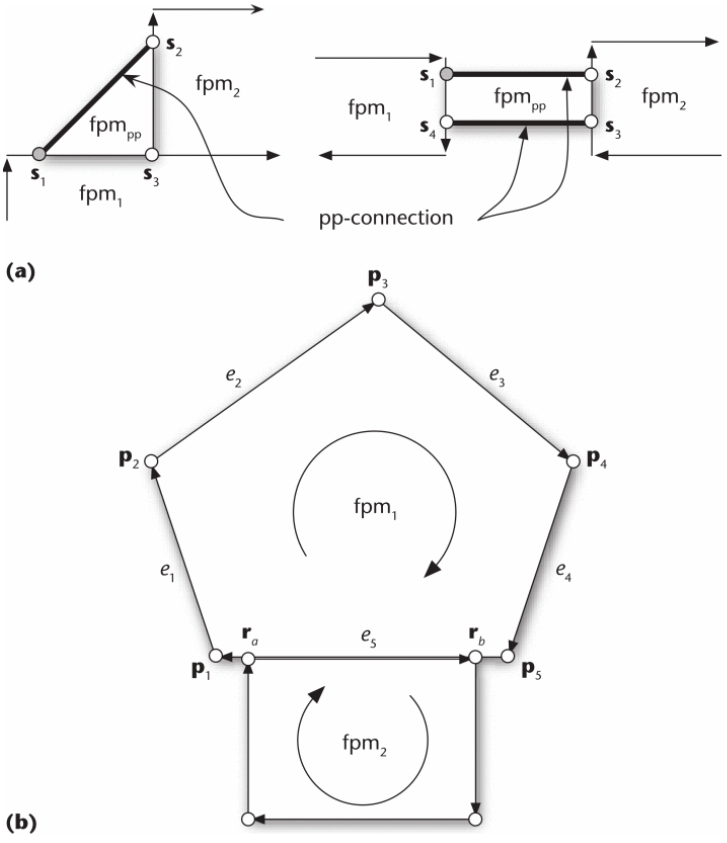


Fonte: Finkenzeller (2008)

Finkenzeller (2008) explica que, ao delimitar o modelo da construção, são adicionadas as informações arquitetônicas básicas na planta baixa, sendo composto de polígonos 2D convexos, chamados de módulo de planta baixa (*fpm*). Cada *fpm* representa uma estrutura de fachada, como alas laterais, oriel ou varandas, com as informações necessárias como tipo, material ou geometria. Os *fpms* podem ser conectados de duas formas:

- Por uma conexão ponto-a-ponto (*pp-connection*), onde os *fpms* são interligados por meio da adição de arestas. Neste caso, é importante que a aresta criada não ultrapasse nenhum *fpm* ou quaisquer outras arestas criado por uma *pp-connection*, para evitar paredes que se cruzam. A Figura 16(a) mostra que novas arestas podem definir uma nova *fpm*;
- Por uma conexão aresta-a-aresta (*ee-connection*), que conecta diretamente dois *fpms* por meio de suas arestas, onde uma aresta é conectada a um intervalo da outra aresta. O *fpm* conectado deve ser modificado para que ambas as arestas correspondam. A Figura 16(b) apresenta um exemplo de *ee-connection*. Também é possível conectar um *fpm* à aresta de uma *pp-connection*.

Figura 16 – Os tipos de conexão de *fpms*, sendo (a) *pp-connection* e (b) *ee-connection*.



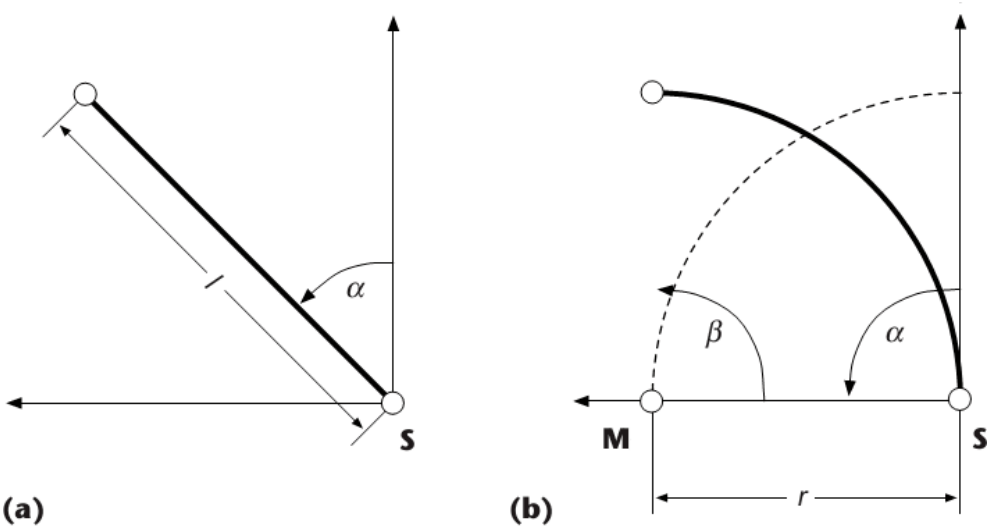
Fonte: Finkenzeller (2008)

Com a representação da planta baixa, podem ser definidos as paredes e os seus cantos. Cada aresta de cada *fpm* forma uma parede básica e todos os vértices internos de um *fpm* formam cantos básicos. O estilo fornece informações para a subdivisão vertical das paredes, espaços entre as paredes, cantos em partições e, opcionalmente, contém informações do modelo de uma janela ou porta (FINKENZELLER, 2008).

Conforme Finkenzeller (2008), para gerar cornijas, é utilizada uma linguagem descritiva, com a definição do contorno em 2D, em dois comandos:

- *line*, que desenha uma linha reta e possui três parâmetros: (*rel* | *abs*), α, e *l*. *rel* indica que a direção da linha é relativa à posição atual de *S*, como mostrado na Figura 17. Por sua vez, *abs* aponta que a direção da linha é relativa à origem. α representa o ângulo da direção e *l* o tamanho da linha (veja a figura 17(a));
- *arc*, que desenha um arco e possui quatro parâmetros: (*rel* | *abs*), α, β e *r*. O parâmetro (*rel* | *abs*) é análoga ao comando de linha. α determina o ponto médio do arco *M*, β define o comprimento do arco e *r* o raio do círculo que delimita o arco. A Figura 17(b) apresenta um exemplo desse comando.

Figura 17 – Os tipos de comandos para descrever um perfil de cornija: (a) comando *line* e (b) comando *arc*



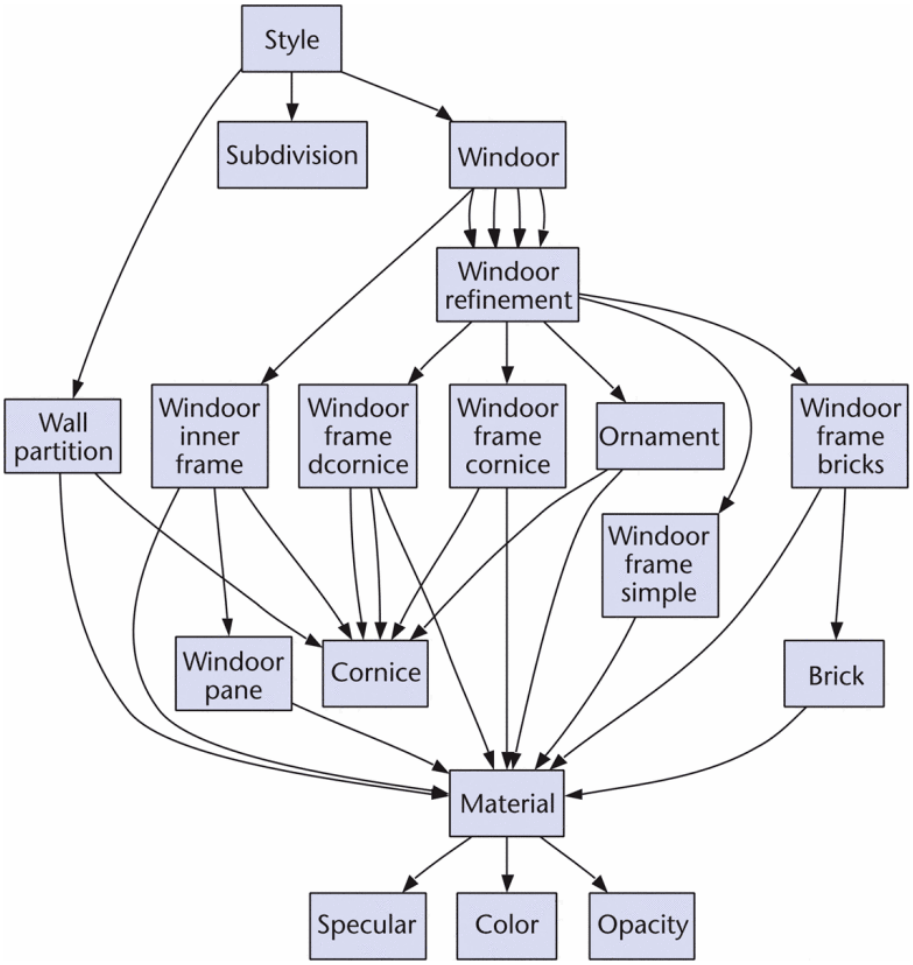
Fonte: Finkenzeller (2008)

Para gerar portas e janelas, o usuário define um espaço retangular na parede, e depois é realizado o refinamento de cada uma das quatro arestas por meio da mesma descrição usada para as cornijas. Isso permite formas arbitrárias de janelas e portas (FINKENZELLER, 2008).

Para representar os componentes da construção, que definem o estilo, é utilizada uma árvore semântica hierárquica (veja Figura 18), partindo de um estado mais amplo até especificar

os detalhes. Quando um estilo é aplicado ao contorno de um edifício, as informações geométricas são suficientes para criar automaticamente a fachada detalhada da construção (FINKENZELLER, 2008).

Figura 18 – Hierarquia do estilo de construções



Fonte: Finkenzeller (2008)

Depois de definir a geometria da construção e o estilo de sua aparência, no próximo passo, o sistema combina o contorno e o estilo, cria uma descrição hierárquica para a construção inteira e a armazena em um grafo. Isso inclui a subdivisão de paredes básicas em paredes individuais e a atribuição de estilos de parede e janela em cada parede. O grafo ainda possui representação simbólica com informações geométricas mínimas. O *designer* pode interagir com os dados armazenados e, assim, modificar a aparência da construção facilmente (FINKENZELLER, 2008).

Uma limitação apontada por Finkenzeller (2008) do sistema proposto é que ele não lida diretamente com estruturas de edifícios mais complexas, como por exemplo, paredes

inclinadas. Além disso, não adiciona detalhe nos telhados, como chaminés.

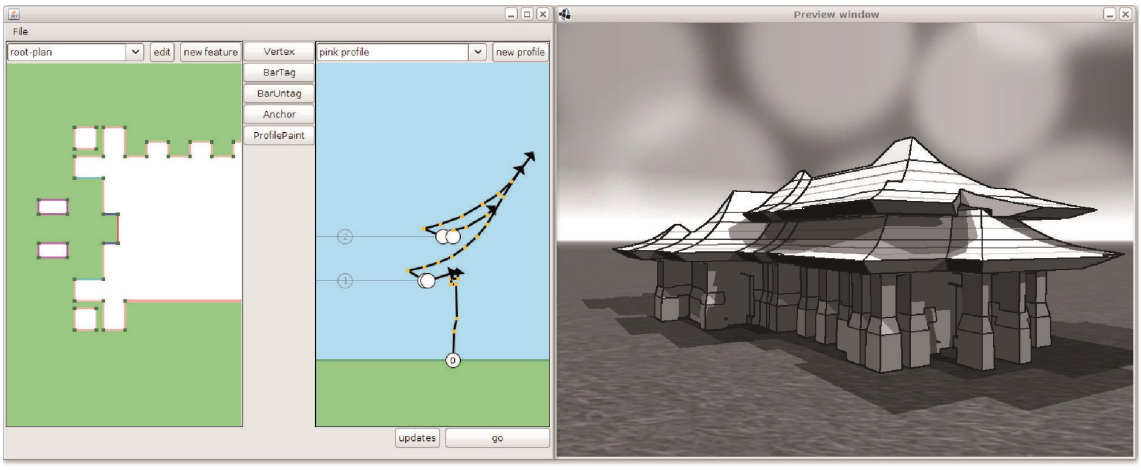
3.2 Interactive Architectural Modeling with Procedural Extrusions

Kelly e Wonka (2011) especificam um sistema de modelagem interativa e procedural de exteriores de construções, baseado em extrusões procedurais a partir de uma planta baixa. Esta abordagem busca desenvolver uma ferramenta de modelagem interativa e procedural para gerar elementos arquitetônicos complexos, como telhados curvos, telhados pendentes, lucarnas (aberturas existente no telhado), pilares, chaminés, janelas salientes, pilastras entre outros.

A primeira parte da abordagem de Kelly e Wonka (2011) é identificar as edições mais importantes e projetar uma interface de usuário (UI) para especificar as extrusões. A segunda parte é uma coleção de algoritmos para calcular extrusões especificadas pelo usuário.

A interface de usuário controla uma sequência de extrusões que são particularmente adequadas para criar a massa de modelos arquiteturais. A UI consiste em um painel mostrando o plano, outro para apresentar o perfil e uma janela de visualização 3D, conforme ilustrado na Figura 19. O plano representa um piso da construção, com um conjunto de arestas e de vértices. Para cada aresta do plano, existe uma coleção de segmentos de polilinhas, denominada de perfil, que define a direção de uma seção transversal pela construção naquela aresta do plano. Conforme o usuário edita o plano ou os perfis, é apresentado o resultado na janela de visualização 3D. A UI apresenta operações para inserir, excluir e mover vértices nas polilinhas do perfil e vértices no plano (KELLY; WONKA, 2011).

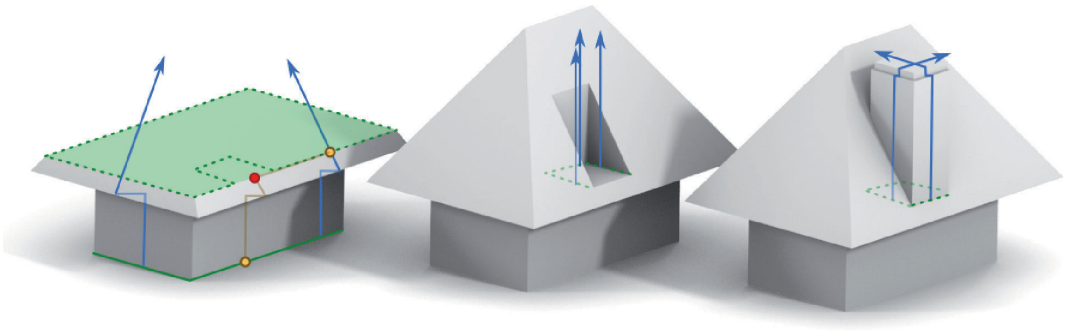
Figura 19 – Exemplo da UI na modelagem de um templo.



Fonte: Kelly e Wonka (2011)

Ademais, Kelly e Wonka (2011) apresentam o conceito de âncoras, que marca os componentes no modelo da construção, como portas, janelas ou chaminés, para serem localizados após edições subsequentes. Com a especificação de âncoras, podem ser realizadas edições no plano selecionado, para modificar, criar ou excluir arestas. São permitidos dois tipos de adições de arestas: por um conjunto arbitrário de arestas no plano (denominado *forced steps*) ou pela escolha de uma variedade de formas simples que podem ser inseridas (denominado *natured steps*). A Figura 20 ilustra a aplicação desses conceitos: a planta (em verde) e os perfis (linhas azuis) definem a forma da estrutura, enquanto que as âncoras localizam onde ficará a chaminé (ponto vermelho). O *natural step* é inserido no edifício no local ancorado (linhas verdes tracejadas).

Figura 20 – Representação do plano, perfis, âncoras e edição do plano (Esquerda), sendo realizada a extrusão do telhado (Meio) e depois da chaminé (Direita).



Fonte: Kelly e Wonka (2011)

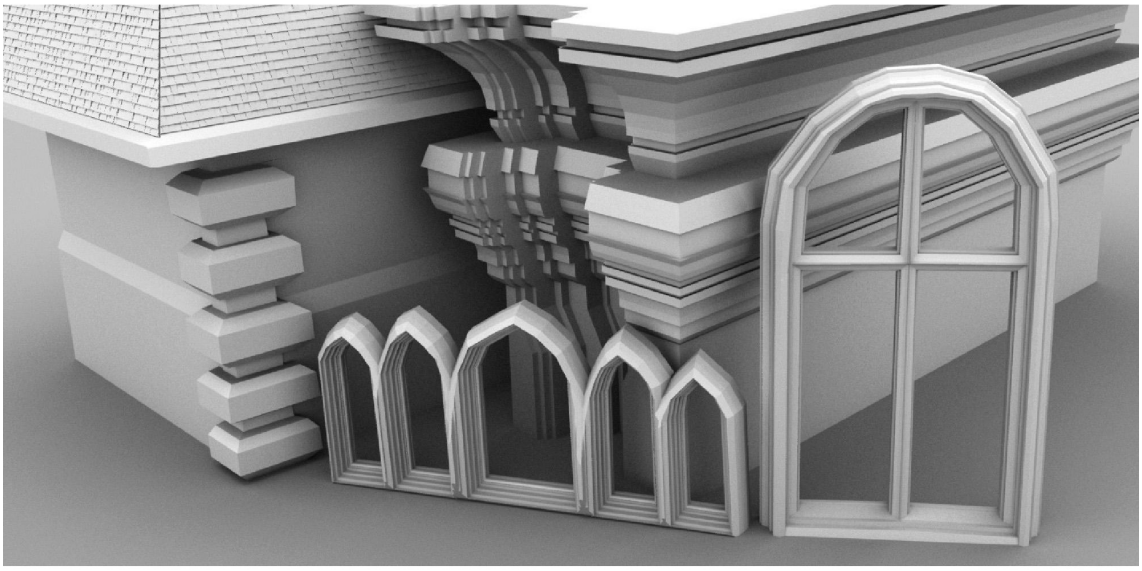
No que se refere aos algoritmos, o sistema recebe como entrada a planta baixa, os perfis associados às arestas da planta, os eventos de deslocamento de perfil e os eventos de âncora. O evento de deslocamento de perfil indica o início das saliências no modelo e o evento de âncora especifica a localização das edições do plano ou uma instância de malha. A saída principal do algoritmo é o modelo de massa da construção. Depois, ele pode ser pós-processado para aplicar texturas, adicionar geometrias e anexar elementos em pontos de ancoragem (KELLY; WONKA, 2011).

Como resultado, Kelly e Wonka (2011) apresentam exemplos de recursos que podem ser gerados por sua abordagem. A Figura 21 ilustra alguns elementos arquiteturais.

Kelly e Wonka (2011) destacam as limitações do sistema proposto, sendo elas: edições menores na planta baixa podem resultar em mudanças maiores na superfície do telhado; modelar arcos circulares são difíceis, pois qualquer ajuste na largura do arco deve ser feito um reescalonamento dos perfis; não é interessante modelar um telhado que é sustentado apenas por um grande número de pilares, porque não é fácil modelar a transição dos pilares para o telhado;

por fim, não existe uma garantia formal de que a implementação de um algoritmo funcione corretamente para todas as entradas.

Figura 21 – Representação de diferentes recursos arquitetônicos que podem ser gerados.



Fonte: Kelly e Wonka (2011)

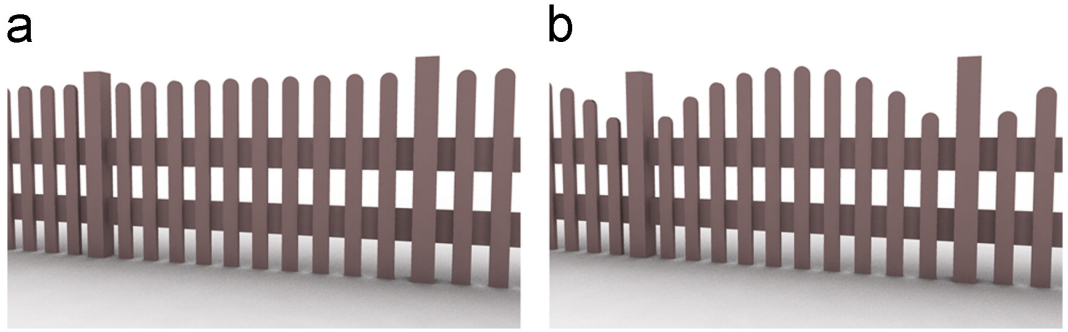
3.3 Shape grammars on convex polyhedra

Os sistemas de formas normalmente utilizam caixas (cuboides) como delimitadores de volume, com isso, ela pode ser dividida em caixas menores ao longo de qualquer um de seus três eixos. Thaller *et al.* (2013) propõem utilizar um poliedro convexo como delimitador de volume em vez de utilizar caixas. Isso aumentaria as possibilidades de divisão, pois não estão mais limitadas aos eixos principais, mas podem utilizar planos arbitrários. Essas divisões permitem uma decomposição volumétrica em elementos convexos e, como os poliedros convexos podem representar muitas formas com mais fidelidade do que as caixas, as regras gramaticais das formas podem se adaptar a uma gama mais ampla de contextos distintos.

Como exemplo motivador, Thaller *et al.* (2013) apresentam uma cerca conforme a Figura 22(a). Ela pode ser facilmente modelada em sistemas de gramática de forma que suportam caixas e operações de divisão. Por sua vez, a cerca na Figura 22(b) apresenta um formato diferente, onde cada segmento de cerca tem um limite curvo, as estacas individuais têm alturas diferentes e apresentam uma forma arredondada. Um sistema de gramática de caixa suficientemente poderoso pode modelar isso usando primeiro uma caixa delimitadora maior e,

em seguida, calcularia a caixa curva delimitadora da cerca. Isso é grosseiro, uma vez que o delimitador perde a maior parte de seu significado e já não representa o espaço disponível para ser preenchido de uma forma.

Figura 22 – Em relação a gramática de formas que utiliza uma caixa como delimitador de volume, um exemplo de cercas que podem ser facilmente modeladas (a) e que representam um desafio (b).



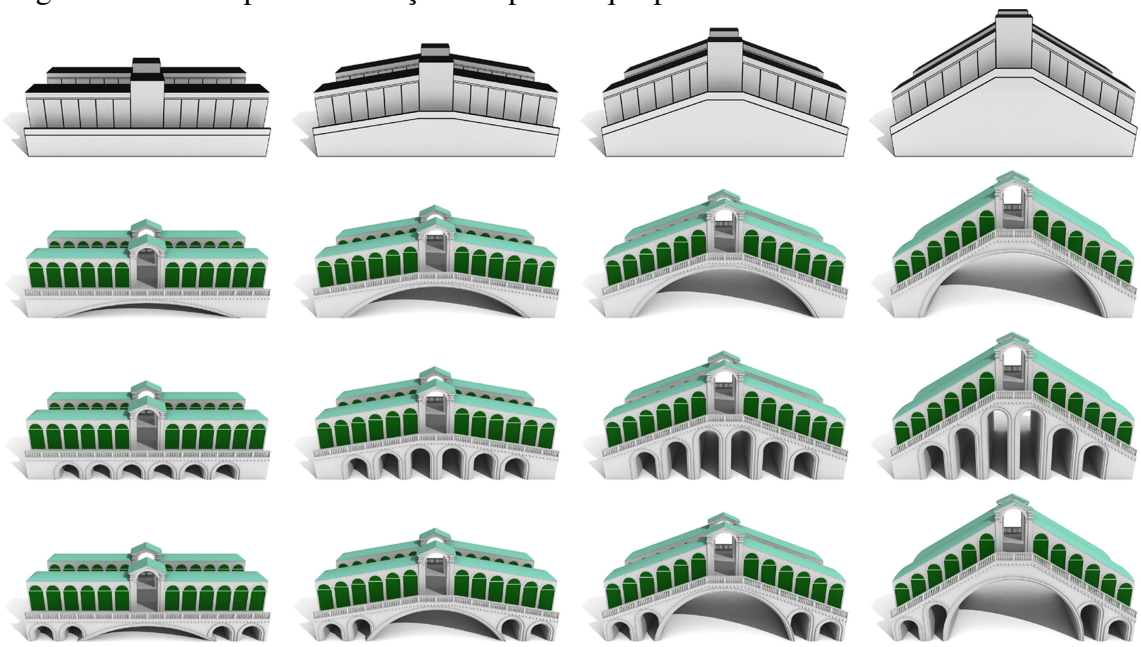
Fonte: Thaller *et al.* (2013)

Além de propor poliedros convexos como uma generalização de caixas ao usar planos divididos arbitrariamente, Thaller *et al.* (2013) definem um conjunto de operações que permitem adaptar subformas à forma pai, ou mais especificamente, ao poliedro convexo que a delimita.

Thaller *et al.* (2013) apresentam um exemplo de aplicação de seu modelo. A Figura 23 apresenta variações de pontes que demonstram as vantagens do uso dos poliedros convexos, das seguintes maneiras: a inclinação da ponte define os escopos básicos (poliedros convexos) que formam a ponte, de modo que são divididos sem envolver o ângulo de declive em nenhum cálculo; e todos os arcos usados se adaptam ao seu escopo. A primeira linha da Figura 23 apresenta uma etapa intermediária da gramática da ponte demonstrando a estrutura básica.

O uso de poliedros convexos permite que as regras se adaptem a uma gama muito maior de formas. E não há perda de flexibilidade, já que o que pode ser feito com caixas também pode ser feito com poliedros convexos. Ao mesmo tempo, um modelo de gramática de caixa pode ser traduzido em poliedros convexos de maneira direta, sem complexidade adicional. Porém, as operações padrões de subdivisão e repetição claramente não são satisfatórias para descrever layouts mais complexos. Ademais, a flexibilidade adicional de poliedros convexos tem um custo na complexidade de implementação, comparada a uma gramática de forma cujas formas não terminais têm um escopo em forma de caixa e nenhuma geometria adicional. Por fim, ainda existem formas que não são adequadamente descritas por um escopo convexo (THALLER *et al.*, 2013).

Figura 23 – Exemplos de variações de pontes que podem ser modeladas.



Fonte: Thaller *et al.* (2013)

Edelsbrunner *et al.* (2017) argumentam que, com o método apresentado, elementos arquitetônicos como arcos, escadas em espiral ou escadas inclinadas e corrimãos são possíveis, porém, esses elementos com forma arredondada podem ser mais refinados. Como não há sistema de referência (ou sistema de coordenadas) para a geometria arredondada, as divisões devem ser feitas ao longo de uma curva e cálculos manuais complicados e desnecessários para a configuração da curva devem ser feitos. Além disso, as divisões são sempre realizadas por um plano, o que deixa uma face plana no local da divisão, não permitindo a geometria arredondada nas divisões.

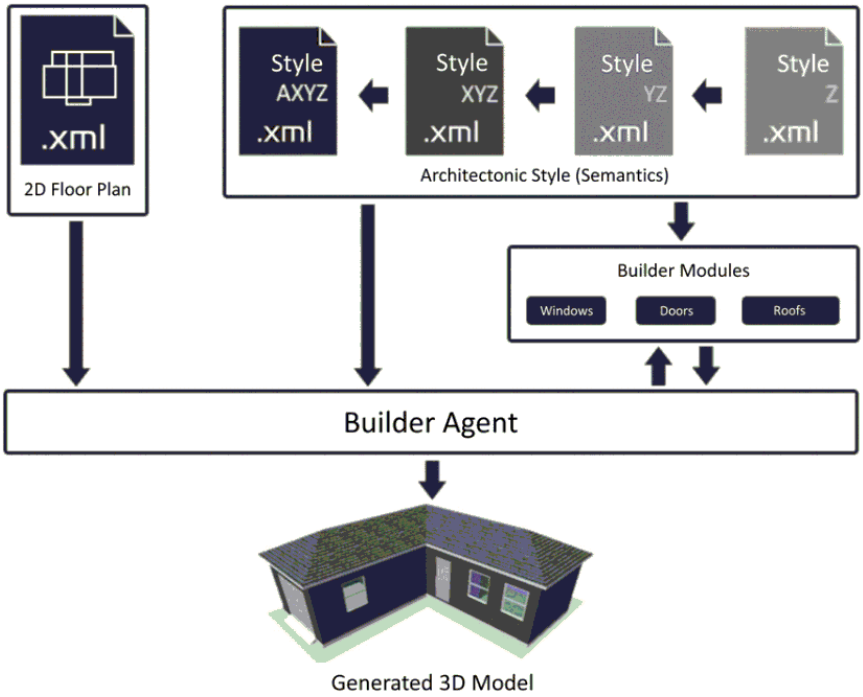
3.4 Real-time Procedural Generation of Personalized Facade and Interior Appearances Based on Semantic

Silveira *et al.* (2015) apresentam um modelo para geração de fachadas customizadas e estilos de interiores de construções que utilizam dois tipos de informações de entrada: geométricas e semânticas. As informações geométricas estão relacionadas à planta bidimensional de um edifício, com suas partes e dimensões, bem como posições de portas e janelas. A informação semântica possibilita a criação de estilos arquitetônicos que possibilita variações de materiais e texturas para fachadas e partes internas, bem como para as formas e dimensões de portas e janelas. Alterar um ou mais parâmetros de entrada modifica a aparência final do resultado.

Para realizar a modelagem, o modelo possui um agente construtor (*builder agent*),

que interpreta as informações de entrada (uma planta 2D e estilo arquitetônico) e solicita ao módulo construtor especializado (*builder module*) a criação de elementos tridimensionais a serem inseridos no edifício (portas, janelas e telhado). Esses elementos são customizados de acordo com o estilo arquitetônico, na sua forma e no material utilizado. O *builder agent* é responsável por consolidar a integração dos elementos e das paredes da construção (SILVEIRA *et al.*, 2015). A Figura 24 ilustra a arquitetura do modelo.

Figura 24 – *Pipeline* do modelo proposto por Silveira *et al.* (2015)



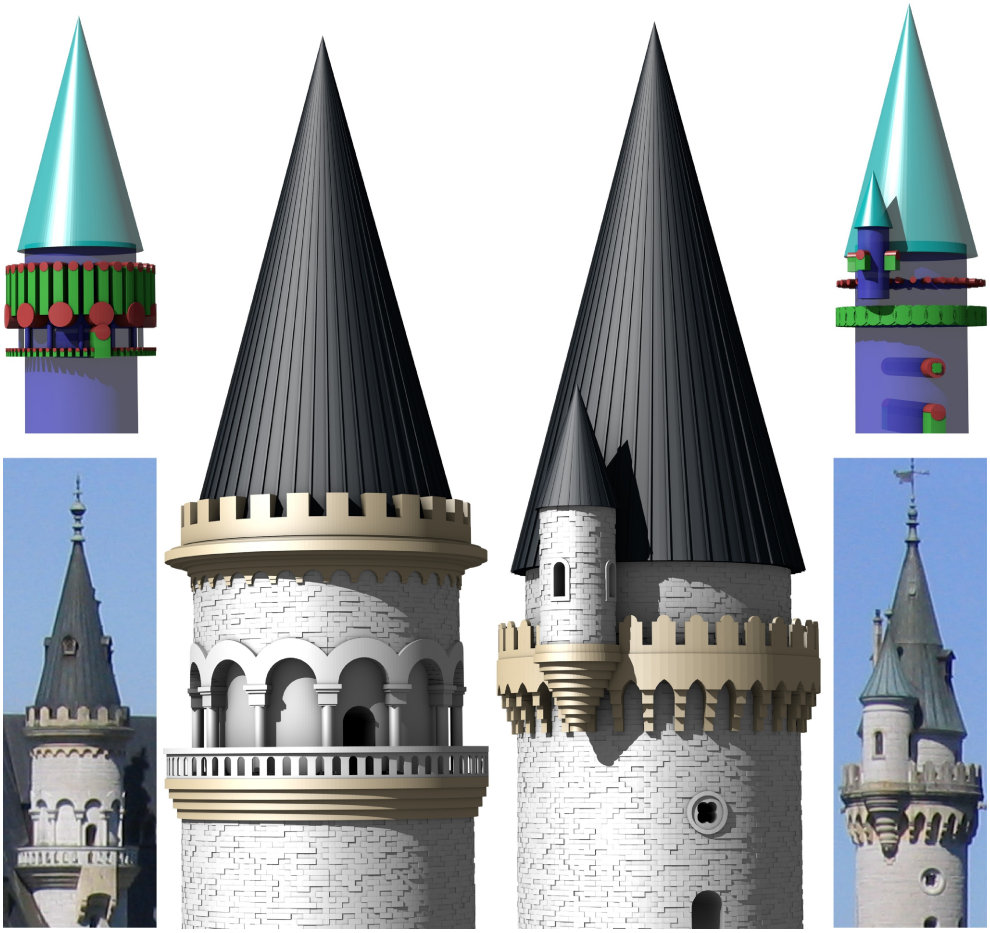
Fonte: Silveira *et al.* (2015)

Como vantagens, o modelo apresentado permite gerar diversas variações de um mesmo edifício em tempo real, expandir o modelo usando diversos módulos construtores especializados e incluir um formato semântico para especificar estilos e recursos das construções. Entretanto, Silveira *et al.* (2015) apontam melhorias para aumentar o nível de detalhes dos modelos gerados, sendo elas a possibilidade de gerar curvas e ornamentos característicos de muitos estilos arquitetônicos e a criação de edifícios com vários pisos, e isso requer a criação de um módulo construtor para gerar escadas e elevadores (SILVEIRA *et al.*, 2015).

3.5 Procedural modeling of architecture with round geometry

Edelsbrunner *et al.* (2017) apresentam uma abordagem que permite integrar sistemas de coordenadas em formatos além do cartesiano, de modo que torna-se possível gerar modelos de construções com estrutura mais complexas, como edifícios com geometria arredondada.

Figura 25 – Modelos 3D (centro) gerados das torres do castelo de Neuschwanstein, na Alemanha (fotos nas laterais, parte inferior). Nas laterais, parte superior, a representação da geometria.



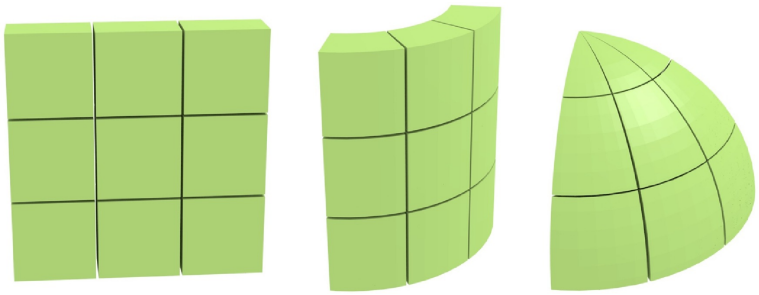
Fonte: Edelsbrunner *et al.* (2017)

Com a possibilidade de utilizar sistemas de coordenadas personalizados, o usuário pode especificar qual deseja utilizar, dependendo da necessidade. Modelos mais simples são gerados por um sistema de coordenadas cartesianas. Para produzir estruturas como torres ou pilares, pode ser utilizado o sistema de coordenadas cilíndricas. Para gerar cúpulas, pode ser usado um sistema de coordenadas esféricas. Além disso, outros sistemas de coordenadas são viáveis, por exemplo, para gerar telhados em forma de cone (EDELSBRUNNER *et al.*, 2017).

Edelsbrunner *et al.* (2017) indicam que, com a possibilidade de escolha de um sistema de coordenadas, torna-se possível realizar mais tipos de divisões na geometria. Onde

uma divisão de uma parede em *split grammar* produz partes retangulares, agora também é suportado dividir paredes cilíndricas ou esféricas em subpartes (veja Figura 26). O objetivo de estabelecer sistemas de coordenadas para o processo de divisão é manter o idioma de divisão existente inalterado, e modificar apenas o espaço no qual está operando. Isso permite que o usuário trabalhe com métodos conhecidos e crie rapidamente modelos procedurais com geometria arredondada.

Figura 26 – Parede dividida em diferente coordernadas.



Fonte: Edelsbrunner *et al.* (2017)

O sistema definido por Edelsbrunner *et al.* (2017) apresenta diversos componentes, sendo os principais:

- O **Sistema de coordenadas** é o principal componente onde a geometria se orienta. Por meio dela, o usuário configura um sistema de coordenadas especificando o tipo (cartesiano, cilíndrico, esférico, entre outros) e os parâmetros necessários (origem, vetores de alinhamento de eixos, etc);
- O **Escopo** descreve a aparência "abstrata" da forma. É a base para outras operações de divisão (as próprias divisões criam novas subformas com escopos modificados);
- A **Forma** representa uma entidade que possui um escopo e vários atributos. A geometria final da forma pode ser a geometria de seu escopo, uma modificação do escopo, *assets* importados pré-modelados ou simplesmente uma geometria vazia;
- A **Árvore de divisão** resulta da aplicação de operações nas formas, uma vez que produzem novas subformas que são inseridas na árvore como filhos. No entanto, a árvore pode ser criada ou alterada manualmente;
- Os **Operadores** permitem modificar os diferentes componentes. Os mais importantes são os aplicados sobre a forma, que são semelhantes aos operadores comuns nas *split grammars*. Um operador de forma gera novas subformas (nós

filhos na árvore) com dimensões específicas no sistema de coordenadas definido. Em seguida, ele copia todos os atributos da forma atual para as subformas e chama as funções de regra especificadas para cada subforma;

- As **Funções de regra** são sub-rotinas que são chamadas pelos operadores. O operador cria uma nova forma e a função de regra funciona localmente nessa forma. Por meio desse mecanismo, as funções de regra podem ser implementadas como funções comuns de linguagem de programação e, dessa forma, o usuário pode tirar proveito de todos os recursos de programação existentes, ao mesmo tempo em que usa expressões idiomáticas de divisão;
- O **Elemento de alinhamento** fornece uma maneira de alinhar os elementos.

Como limitação, Edelsbrunner *et al.* (2017) argumentam que formas que não seguem uma geometria cônica podem ser difíceis ou mesmo impossíveis de reproduzir e precisam de uma aproximação elaborada. Isso ocorre pois a estrutura geométrica de dados escolhida foi a que representa sólidos, uma vez que facilita a execução de operações booleanas (união, interseção, diferença), sendo necessárias em estruturas mais complexas. Uma outra alternativa apontada, seria o uso de uma estrutura de dados em malhas, por ser mais adequada para a representação de formas arbitrárias, porém, é frequentemente sujeita a erros nas operações booleanas. Outra limitação apontada é que essas operações booleanas podem resultar em sérios custos em termos de desempenho.

3.6 Considerações finais

Neste capítulo, foram apresentadas diferentes estratégias para a geração de elementos de fachada, com a demonstração de suas particularidades, além de suas limitações. No próximo capítulo, será abordada a técnica proposta pelo presente trabalho para a geração de elementos de fachadas, com a adição de novas operações complementares na linguagem *SELEX*.

4 METODOLOGIA

Neste capítulo, é apresentada a proposta deste trabalho para a geração procedural de elementos de fachadas com estruturas arquitetônicas complexas, e a abordagem utilizada com o objetivo de solucionar esse desafio.

4.1 Problema

Conforme apresentado no Capítulo 3, a geração de elementos de fachada de edifícios é bastante utilizada no campo da computação gráfica para a representação de ambientes virtuais. Contudo, a maioria dos trabalhos apresentados possuem dificuldade em modelar construções mais heterogêneas, com estruturas arredondadas, ou mesmo não a permitem. Além disso, componentes arquiteturais mais complexos, como janelas e portas com curvas, são muitas vezes adicionados apenas com o uso de importação.

Segundo Jiang *et al.* (2018), a linguagem *SELEX* não modela formas com curvas diretamente, apenas sendo possíveis por importações, de modo que não é permitido modelar a maioria das fachadas curvas, conforme apresentado no Capítulo 2.6. A Figura 27 expõe um exemplo de arquitetura que não pode ser gerada diretamente pelo *SELEX*.

Figura 27 – Exemplo de edifício que está além da capacidade do *SELEX*.

Fonte: Jiang *et al.* (2018)

4.2 Abordagem

Com o objetivo de gerar estruturas arredondadas em modelos arquitetônicos, Brito *et al.* (2021) complementam a linguagem *SELEX* com a criação de uma nova *action* responsável por realizar o arredondamento na forma selecionada, de acordo com parâmetros definidos pelo usuário, conforme apresentado e especificado no Capítulo 2.6. Este trabalho torna possível gerar o modelo de massa de edifícios com arquiteturas complexas.

Brito *et al.* (2021) realizam a integração com o *SELEX* utilizando o *Blender* como ambiente de desenvolvimento e de geração dos modelos. O *Blender* é uma plataforma de criação 3D *open source*, que suporta todo o *pipeline* 3D, mantido pela Blender Foundation (2021a). Dentre diversas ferramentas, é oferecida uma API, denominada *Blender Python* (*bpy*), também mantida pela Blender Foundation (2021b), que permite criar, remover, editar e realizar diversas operações em modelos gráficos com o uso da linguagem de programação *Python*, mantida pela Python Software Foundation (2021).

Desse modo, o presente trabalho acrescenta ao que foi implementado por Brito *et al.* (2021), com a adição de novas operações que podem ser aplicadas nos modelos gerados. As funcionalidades desenvolvidas permitem adicionar janelas ou portas conforme especificação do usuário - que podem ser gerados com arredondamento, atualizar a quantidade de células das formas virtuais existentes e apresenta um novo modo de selecionar as células nas *selection expressions*.

A seguir, são apresentadas as funcionalidades desenvolvidas por Brito *et al.* (2021) que foram atualizadas e, depois, as novas operações desenvolvidas.

4.2.1 Módulos atualizados

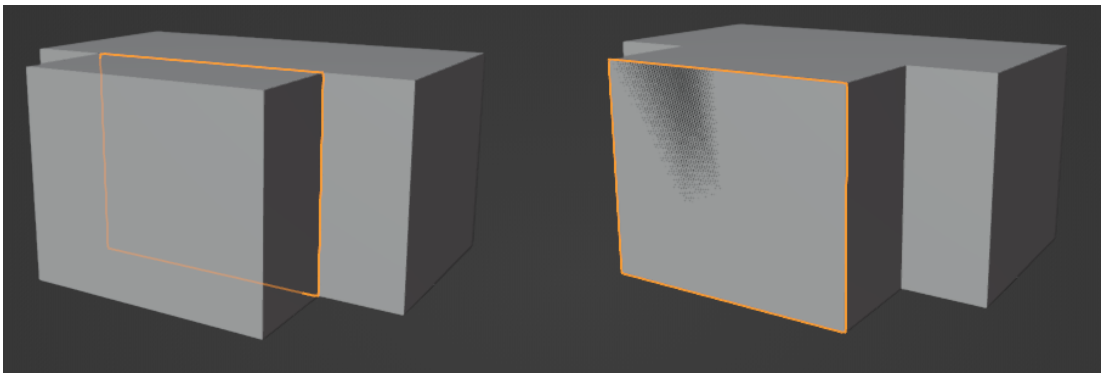
No desenvolvimento, foram utilizadas as mesmas tecnologias apresentadas anteriormente. Porém, a versão do *Blender* utilizada é a 3.5, em contrapartida da versão utilizada por Brito *et al.* (2021), que era a versão 2.83.8. Devido a essa alteração, foi necessário atualizar os parâmetros da operação `bevel`, do *bpy*, onde a função é utilizada na implementação da *action roundShape*.

Para proporcionar mais opções na geração de fachadas, tornou-se opcional o uso da operação de agrupamento *groupRegions*, que é utilizada para agrupar um conjunto de células selecionadas, que são aplicadas nas *actions* subsequentes. Dessa forma, torna-se mais simples

criar diversos elementos adjacentes, como janelas, em uma mesma seleção.

Também foi realizada a atualização do método que implementa a *action addVolume*, com o mesmo nome, para que as laterais do volume criado após a extrusão sejam salvas corretamente. Além disso, foi corrigido o posicionamento da *grid* gerada na aplicação desta *action*, uma vez que a forma virtual permanecia na posição original da forma de origem. Isso será importante, pois as operações para adicionar os elementos de fachada dependem da posição correta das formas virtuais. A Figura 28 apresenta o resultado da alteração no posicionamento da forma virtual.

Figura 28 – Comparação da forma virtual gerada na aplicação da *action addVolume* antes da alteração (esquerda) e depois (direita).



Fonte: Próprio autor

Código-fonte 2 – Descrição do modelo gerado da Figura 28

```
label = "building"; width = 8; depth = 4; height = 4;

{<> -> createShape(label, width, depth, height)};

{< descendant() [label=="building"] / [label=="building_front"] > -> createGrid("
    main_front_grid", 1, 5)};

{< descendant() [label=="building"] / [label=="building_front"] / [label=="
    main_front_grid"] / [type=="cell"] [rowIdx in (1)] [colIdx in (2, 3, 4)] [::
    groupRegions()] > -> addVolume("entrance", "building_front", 2, ["entrance_front",
     "entrance_left", "entrance_right"])};
```

4.2.2 Módulos adicionados

Com a descrição da *action roundShape*, é possível gerar modelos de massa com geometria arredondada que permitem a criação de construções complexas. No entanto, não

possibilita a adição de elementos de fachada. Dessa forma, foram introduzidas novas operações que auxiliam na construção de janelas ou portas em modelos de construção, gerados a partir da estrutura desenvolvida por Brito *et al.* (2021).

As operações foram desenvolvidas visando tornar o processo de adicionar esses elementos mais intuitivo, de modo que os parâmetros sejam coerente com a nomenclatura. Por exemplo, as operações evitam que o usuário necessite especificar parâmetros de arredondamento quando deseja gerar uma porta reta.

A seguir, serão apresentadas as funcionalidades desenvolvidas. Inicialmente, será demonstrada uma função que auxilia na seleção das células nas formas virtuais. Depois, são apresentadas quatro *actions* que adicionam elementos de fachadas nas formas de construção. Por fim, são apontadas as principais funções utilitárias implementadas para auxiliar o desenvolvimento.

4.2.2.1 Pattern

A operação *pattern* é uma função de seleção especificada por Jiang *et al.* (2018), que seleciona células de uma forma virtual, de acordo com uma expressão regular simples. Ela é descrita pela forma:

```
pattern(regex, pat),
```

onde *regex* é uma expressão regular e *pat*, um caractere. A expressão regular *regex* será utilizada para gerar um conjunto de caracteres com o tamanho máximo da quantidade de células da forma virtual, definido pelo número de colunas multiplicado pelo número de linhas. Com isso, os caracteres que são iguais ao de *pat* serão as células selecionadas. Por exemplo, se a expressão regular é "*BAAAAB*" e o parâmetro *pat* é "*B*", serão selecionadas a primeira e última célula.

O campo *regex* permite qualquer caractere alfabético e de três operações especiais, sendo elas:

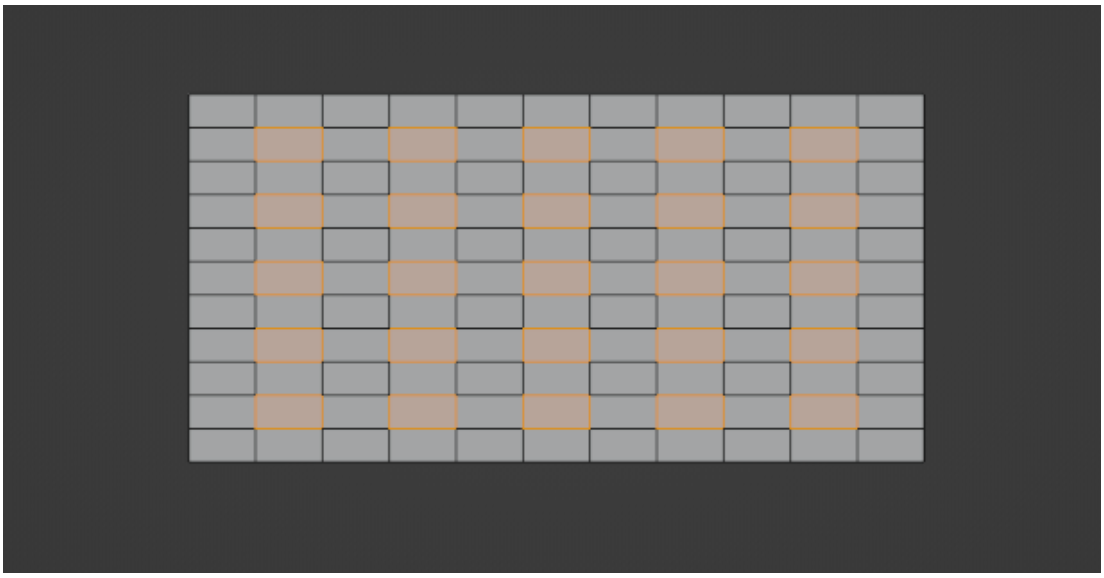
1. Os símbolos de parênteses na forma "(c)", onde c representa um ou mais caracteres que serão memorizados e que podem ser utilizados na aplicação de outras operações;
2. O símbolo de asterisco "$*$", para gerar diversos caracteres repetidamente até o limite de células da forma, sem considerar os caracteres antes e depois dele. Em outras palavras, ele gera $x - y - z$ caracteres, onde x é a quantidade de células da forma selecionada, y é o número de caracteres gerados antes dessa operação e z o número de caracteres gerados depois dela;
3. E os símbolos de chaves na forma "$\{n\}$", onde geram n ocorrências do caractere ou

conjunto de caracteres (quando memorizado).

Como exemplo da aplicação da função *pattern*, considere uma forma virtual com dez células. Suponha que o parâmetro *regex* seja igual a "$(AB)*$" e o parâmetro *pat* é igual a "*A*". A *string* gerada será "*ABABABABAB*", e as células selecionadas serão as de índice par. Observe que, com o uso do asterisco, são gerados caracteres até atingir a quantidade de células, mas se houver um caractere após o símbolo, ele será descontado. Em outro exemplo, considere que o parâmetro *regex* seja "$B*AA$", será gerada a *string* "*BBBBBBBBAA*" e serão selecionadas as duas últimas células. As figuras a seguir ilustram o resultado da aplicação de algumas expressões regulares na seleção de células. Todas as *grids* geradas nos exemplos possuem 11 colunas e linhas, porém, com diferentes dimensionamentos.

A Figura 29 apresenta o exemplo da aplicação da função *pattern*, com a descrição: *[pattern("X{11}((XA){5}XX{11})*", "A")]*. Observe que é empregado o asterisco para repetir o padrão que foi memorizado anteriormente com o uso dos parênteses. Além disso, foram utilizadas as chaves para evitar a repetição de caracteres, simplificando a expressão.

Figura 29 – Representação da *grid* com células selecionadas de forma alternada.

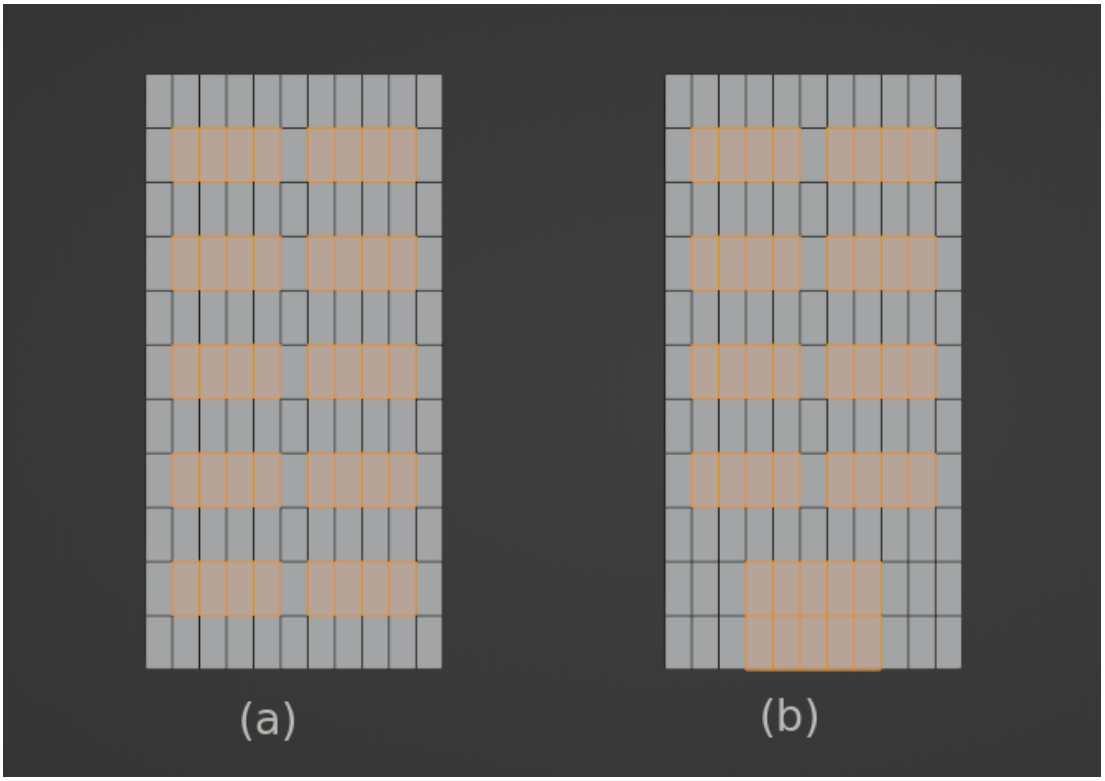


Fonte: Próprio autor

A Figura 30(a) apresenta um modelo em que é gerado elementos que podem representar janelas até o final das células da grid. Essa operação é feita pela forma: *[pattern("X{11}((XA{4}){2}XX{11})*", "A")]*. A Figura 30(b), por sua vez, limita a geração das janelas por quatro vezes, sendo realizada uma seleção que pode representar uma porta, por exemplo. A expressão é descrita pela forma: *[pattern("X{11}((XA{4}){2}XX{11}){4}(XXXA{5}XXX){2}", "A")]*. Esses modelos exemplificam a seleção de células consecutivas quando a função de agrupa-

mento *groupRegions* não é utilizada.

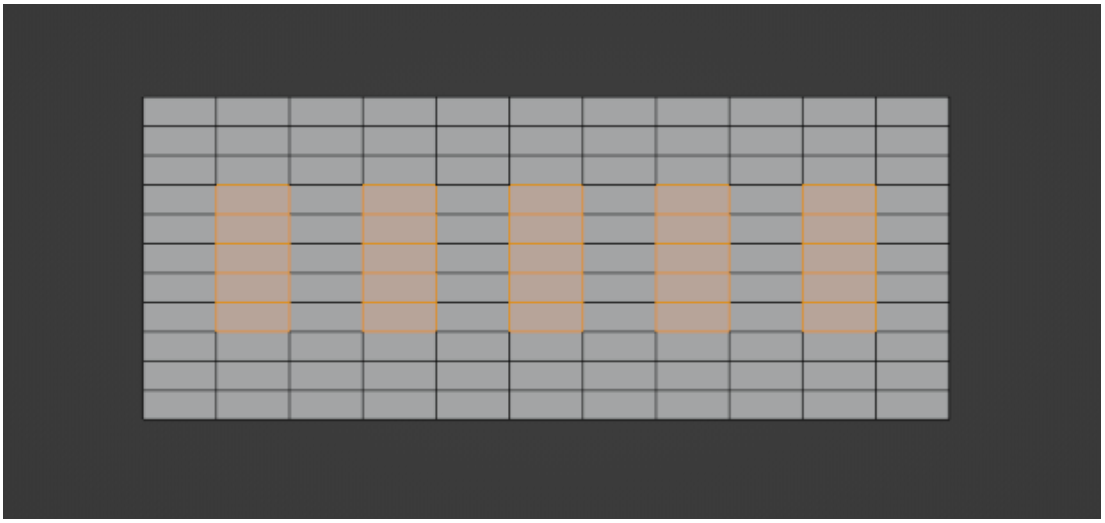
Figura 30 – Modelos de *grid* com as mesmas dimensões, com diferentes células selecionadas, em laranja.



Fonte: Próprio autor

A Figura 31 apresenta um modelo com a operação sendo aplicado na forma: *[pattern("(X{11}){3}((XA){5}X){5}(X{11}){3}", "A")]*. Observe que operações encadeadas também podem ser aplicados nos símbolos de chaves, quando memorizado.

Figura 31 – *Grid* com várias células selecionadas verticalmente, em destaque (laranja).



Fonte: Próprio autor

Vale ressaltar que a escolha de caractere é independente da quantidade utilizada, pois apenas um caractere representa as células selecionadas. Dessa forma, os outros caracteres serão ignorados, podendo ser um ou vários. Além disso, a leitura das células é feita da esquerda para a direita e de cima para baixo. Logo, ao construir a expressão regular, essa ordem deve ser observada.

Por fim, o asterisco não pode ser utilizado mais de uma vez, já que ele repete os caracteres até que o tamanho limite da *string* seja atingido. Caso o símbolo do asterisco não seja utilizado e a *string* não possua o tamanho esperado, as células finais que representam a *substring* não preenchida serão consideradas como não selecionadas. Por exemplo, a operação *pattern* aplicado na Figura 31 pode ser reescrita da forma: *[pattern("(X{11}){3}((XA){5}X){5}", "A")]*.

4.2.2.2 addFacade

A operação *addFacade* é uma *action* que possui a responsabilidade de gerar elementos de fachada, ainda que a construção possua deformidade. Ela é definida da forma:

`addFacade(label)`,

onde *label* é o nome do elemento criado. Note que essa *action* gera apenas modelos retos, logo não permite criar fachadas mais complexa que possuam um determinado grau de arredondamento. O Código-fonte 3 apresenta a sua aplicação.

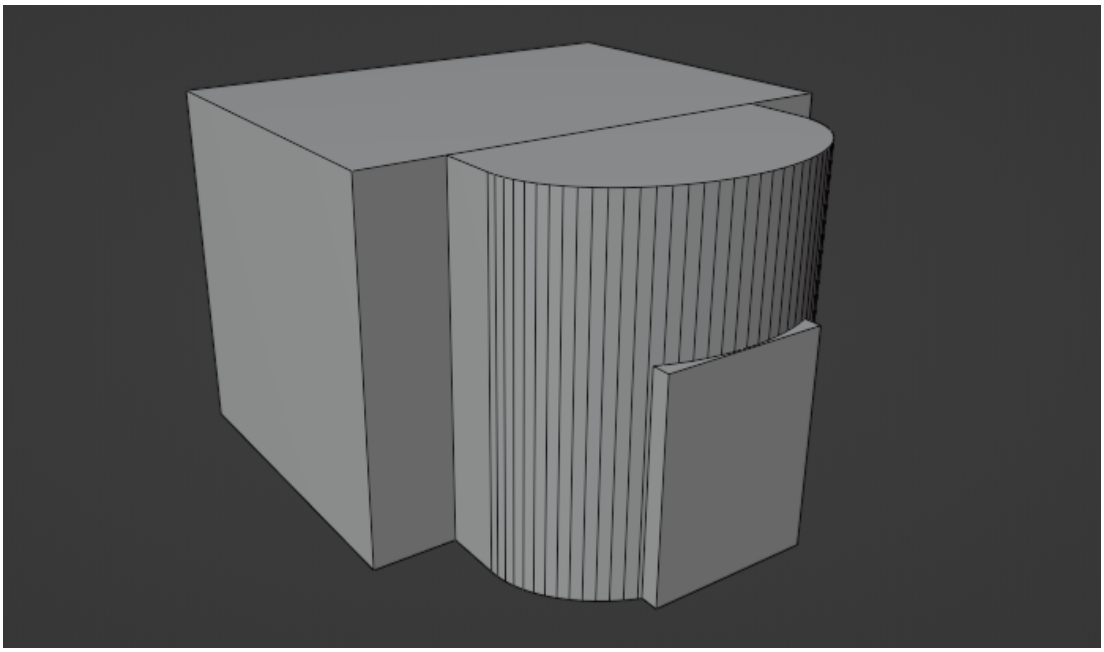
Código-fonte 3 – Geração de construção com estrutura arredondada que possui uma porta

```
#C1 Variables
label = "building"; width = 7; depth = 5; height = 5;

#C2 Create construction shape
{<> -> createShape(label, width, depth, height)};

#C3 Add virtual shape
{< descendant() [label=="building"] / [label=="building_front"] > -> createGrid("
    main_front_grid", 5, 6)};

#C4 Add volume to selected cells
{< descendant() [label=="building"] / [label=="building_front"] / [label=="
    main_front_grid"] / [type=="cell"] [rowIdx in (1,2,3,4,5)] [colIdx in (2,3,4,5]
    [::groupRegions()] > -> addVolume("entrance", "building_front", 2.5, ["
    entrance_front", "entrance_left", "entrance_right"])};

#C5 Add deformation in construction shape
{< descendant() [label=="building"] / [label=="building_front"] / [label=="entrance"]
    / [label=="entrance_front"] > -> roundShape("front", "outside", 0.9, 20, "
```

```
    main_front", "vertical")};

#C6 Add Facade in construction
{<descendant() [label=="building"] / [label=="building_front"] / [label=="entrance"] /
     [label=="entrance_front"] / [label=="entrance_front_grid"] / [type=="cell"] [
    rowIdx in (3,4,5)] [colIdx in (2,3)] [::groupRegions] > -> addFacade("door")}
```

A primeira instrução cria as variáveis que serão utilizadas para gerar o modelo e delimitá-lo (C1). Depois, é criada a forma básica pela *action createShape*, definindo o seu nome e dimensões, sendo a largura, profundidade e altura (C2). Após isso, é utilizada a *action createGrid*, que gera a forma virtual (C3). Observe que serão utilizadas *selection expressions* para selecionar a forma de construção em que será criada a forma virtual. Em seguida, é aplicada a *action addVolume* (C4), que realiza a extrusão no objeto. Após isso, é utilizada a *action roundShape* (C5) com o objetivo de arredondar a forma de construção, selecionando todas as células para aplicar a deformação no volume adicionado. Por fim, foi realizado o uso da *action addFacade*, sendo aplicada no centro da forma de construção. Note que mesmo com a deformação do modelo, a fachada adicionada se encaixa perfeitamente, com a aplicação do volume para que não haja espaços vazios. A Figura 32 apresenta o modelo descrito.

Figura 32 – Modelo de construção com porta, gerado a partir da *action addFacade*.



Fonte: Próprio autor

4.2.2.3 addRoundFacade

Para gerar estruturas de fachada que apresentem um grau maior de complexidade, é definida a *action addRoundFacade* que, ao selecionar um conjunto de células de uma *grid*, gera elementos de fachadas com arredondamento, de acordo com a definição do usuário. Ela é composta pelos seguintes parâmetros:

`addRoundFacade(label, corners, offset, segments, profile),`

onde:

1. ***label*:** Define o nome para a fachada;
2. ***corners*:** Descreve em quais extremidades da fachada adicionada será aplicado o arredondamento, sendo no formato de array que aceita os valores “top_left” (superior esquerdo), “top_right” (superior direito), “bottom_right” (inferior direito), “bottom_left” (inferior esquerdo);
3. ***offset*:** Indica o deslocamento do arredondamento;
4. **segments:** Representa a quantidade de segmentos (faces) que serão criados no processo da deformação.
5. ***profile*:** Define o perfil do elemento. Na prática, indica a direção da deformação.

Com isso, é possível que janelas ou portas sejam adicionadas em modelos gerados pelo *SELEX*, o que complementa o que foi proposto por Brito *et al.* (2021), uma vez que é possível gerar a forma e os elementos de fachada com deformidade. Como exemplo, podemos especificar um modelo da seguinte forma:

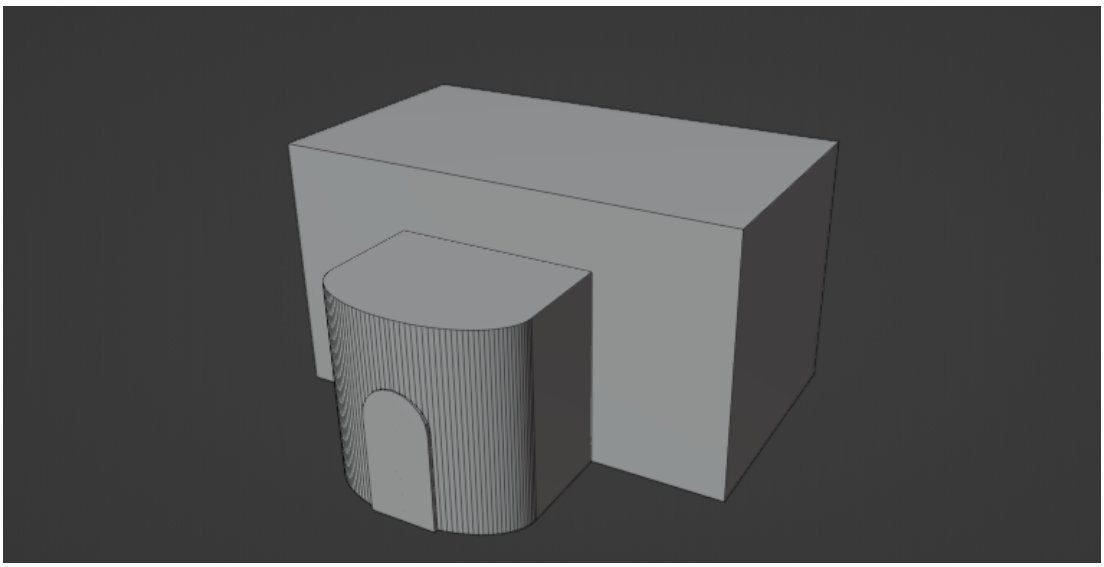
Código-fonte 4 – Modelo de construção com uma porta arredondada

```
#C1 Variables
label = "building"; width = 9; depth = 5; height = 5;

#C2 Create construction shape
{<> -> createShape(label, width, depth, height)};

#C3 Add virtual shape
{< descendant() [label=="building"] / [label=="building_front"] > -> createGrid("
    main_front_grid", 4, 7)};

#C4 Add volume to selected cells
{< descendant() [label=="building"] / [label=="building_front"] / [label=="
    main_front_grid"] / [type=="cell"] [rowIdx in (2, 3, 4)] [colIdx in (3, 4, 5)] [::
    groupRegions()] > -> addVolume("entrance", "building_front", 3, ["entrance_front",
     "entrance_left", "entrance_right"])};
```

```
12
13 #C5 Add deformation in construction shape
14 {< descendant() [label=="building"] / [label=="building_front"] / [label=="entrance"]
       / [label=="entrance_front"] > -> roundShape("front", "outside", 0.42, 30, "
       main_front", "vertical")};
15
16 #C6 Add round facade in construction shape
17 {<descendant() [label=="building"] / [label=="building_front"] / [label=="entrance"] /
        [label=="entrance_front"] / [label=="entrance_front_grid"] / [type=="cell"] [
       rowIdx in (2,3)] [colIdx in (2)] [::groupRegions] > -> addRoundFacade("door", ["
       top_left", "top_right"], 0.9, 5, 0.5)}
```

As três instruções iniciais (C1, C2 e C3) são as mesmas utilizadas no Código-Fonte 3, modificando apenas os valores utilizados na nomenclatura da forma inicial e suas dimensões. Na instrução seguinte, é adicionado o volume, de acordo com a delimitação aplicada nas células da forma virtual selecionada, por meio da *action addVolume* (C4). Esse volume representa a entrada da casa. A próxima operação aplica um arredondamento no volume criado, pela *action roundShape* (C5). Por fim, é adicionada uma porta na entrada da construção, com a aplicação de um arredondamento nas extremidades superiores da forma, com a *action addRoundFacade* (C6). Essa descrição gera o modelo da Figura 33.

Figura 33 – Modelo gerado que aplica a *action addRoundFacade*.



Fonte: Próprio autor

O efeito de arredondamento, aplicado aos elementos de fachada adicionados, é possível com o uso da função `bevel`, do *bpy*, que permite criar deformações nos objetos. Para isso, é aplicada a operação que afeta um grupo de vértices selecionados, diferentemente do

método desenvolvido por Brito *et al.* (2021), que aplica a operação nas arestas.

Com o uso do parâmetro "*corners*", podem ser definidas, em forma de array, as extremidades em que será aplicado o arredondamento. O Código-fonte 5 apresenta exemplos de seu uso com diferentes valores.

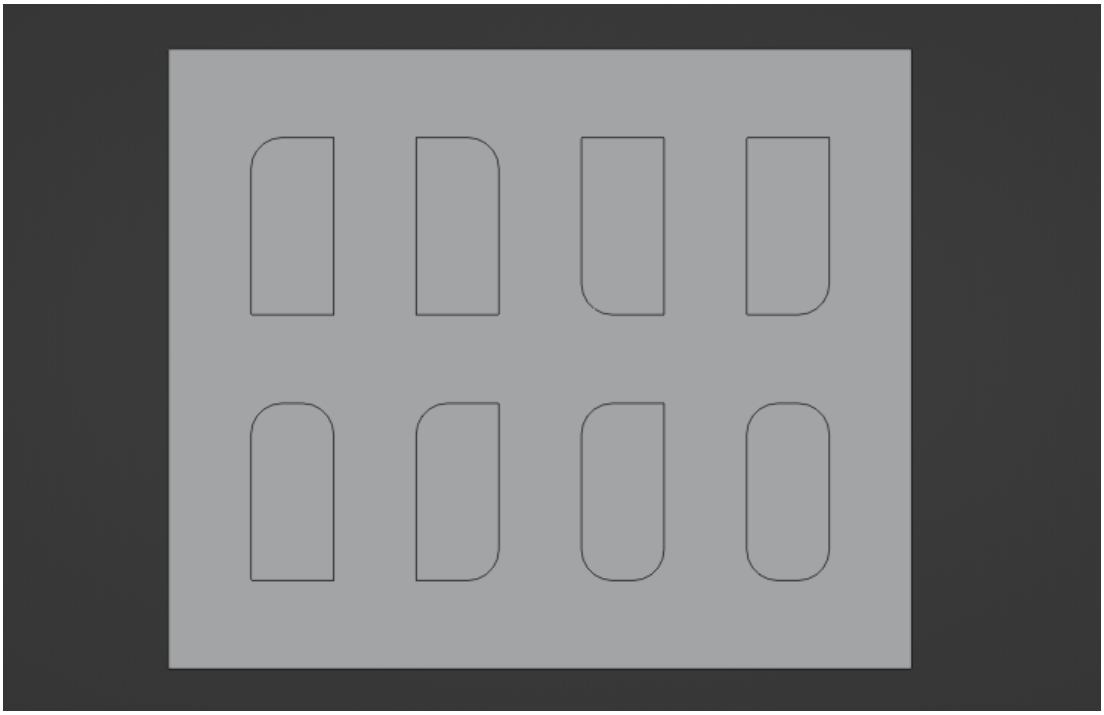
Código-fonte 5 – *Script* para gerar janelas com aplicação do arredondamento em extremidades distintas

```
# Uso do parametro corners em addRoundFacade

# Variables
label = "building"; width = 12; depth = 5; height = 10;

# Create construction shape
{<> -> createShape(label, width, depth, height)};

# Add virtual shape
{< descendant() [label=="building"] / [label=="building_front"] > -> createGrid("
    main_front_grid", 7, 9)};

# top_left
{<descendant() [label=="building"] / [label=="building_front"] / [label=="
    main_front_grid"] / [type=="cell"] [rowIdx in (2,3)] [colIdx in (2)] [::
    groupRegions] > -> addRoundFacade("door", ["top_left"], 0.5, 5, 0.5)}

# top_right
{<descendant() [label=="building"] / [label=="building_front"] / [label=="
    main_front_grid"] / [type=="cell"] [rowIdx in (2,3)] [colIdx in (4)] [::
    groupRegions] > -> addRoundFacade("door", ["top_right"], 0.5, 5, 0.5)}

# bottom_right
{<descendant() [label=="building"] / [label=="building_front"] / [label=="
    main_front_grid"] / [type=="cell"] [rowIdx in (2,3)] [colIdx in (6)] [::
    groupRegions] > -> addRoundFacade("door", ["bottom_right"], 0.5, 5, 0.5)}

# bottom_left
{<descendant() [label=="building"] / [label=="building_front"] / [label=="
    main_front_grid"] / [type=="cell"] [rowIdx in (2,3)] [colIdx in (8)] [::
    groupRegions] > -> addRoundFacade("door", ["bottom_left"], 0.5, 5, 0.5)}

# top_left and top_right
{<descendant() [label=="building"] / [label=="building_front"] / [label=="
    main_front_grid"] / [type=="cell"] [rowIdx in (5,6)] [colIdx in (2)] [::
    groupRegions] > -> addRoundFacade("door", ["top_left", "top_right"], 0.5, 5, 0.5)}

# top_left and bottom_right
{<descendant() [label=="building"] / [label=="building_front"] / [label=="
```

```
    main_front_grid"] / [type=="cell"] [rowIdx in (5,6)] [colIdx in (4)] [::
    groupRegions] > -> addRoundFacade("door", ["bottom_left", "top_left"], 0.5, 5,
    0.5)}

# top_left, bottom_left and bottom_right
{<descendant() [label=="building"] / [label=="building_front"] / [label=="
    main_front_grid"] / [type=="cell"] [rowIdx in (5,6)] [colIdx in (6)] [::
    groupRegions] > -> addRoundFacade("door", ["top_left", "bottom_left", "
    bottom_right"], 0.5, 5, 0.5)}

# all
{<descendant() [label=="building"] / [label=="building_front"] / [label=="
    main_front_grid"] / [type=="cell"] [rowIdx in (5,6)] [colIdx in (8)] [::
    groupRegions] > -> addRoundFacade("door", ["top_left", "bottom_left", "
    bottom_right", "top_right"], 0.5, 5, 0.5)}
```

A Figura 34 ilustra o modelo gerado com as regras especificadas anteriormente.

Figura 34 – Modelo que exemplica o uso das opções do parâmetro “corners”



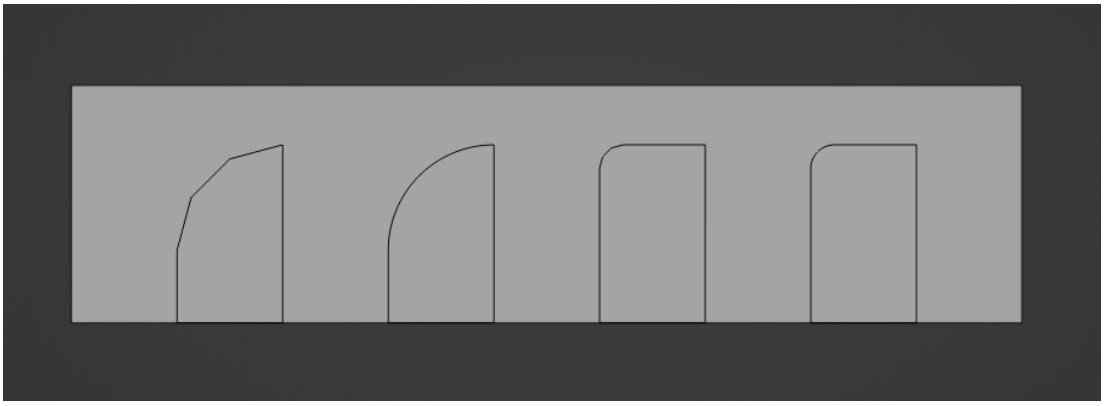
Fonte: Próprio autor

O parâmetro “*offset*” determina o deslocamento do arredondamento, definido pela distância dos vértices criados a partir do vértice original. O parâmetro “*segments*”, por sua vez, permite aumentar o número de segmentos da curva de arredondamento, o que pode torná-la mais suave. O Código-fonte 7 e a Figura 35 apresentam exemplos do uso desses parâmetros.

Código-fonte 6 – Script que gera diferentes elementos com a aplicação dos parâmetros "*offset*" e "*segments*"

```
# Variables
label = "building"; width = 20; depth = 5; height = 5;

# Create construction shape
{<> -> createShape(label, width, depth, height)};

# Add virtual shape
{< descendant() [label=="building"] / [label=="building_front"] > -> createGrid("
    main_front_grid", 4, 9)};

# Offset = 3 and Segments = 3
{<descendant() [label=="building"] / [label=="building_front"] / [label=="
    main_front_grid"] / [type=="cell"] [rowIdx in (2,3,4)] [colIdx in (2)] [::
    groupRegions] > -> addRoundFacade("door", ["top_left"], 3, 3, 0.5)}

# Offset = 3 and Segments = 50
{<descendant() [label=="building"] / [label=="building_front"] / [label=="
    main_front_grid"] / [type=="cell"] [rowIdx in (2,3,4)] [colIdx in (4)] [::
    groupRegions] > -> addRoundFacade("door", ["top_left"], 3, 50, 0.5)}

# Offset = 0.5 and Segments = 3
{<descendant() [label=="building"] / [label=="building_front"] / [label=="
    main_front_grid"] / [type=="cell"] [rowIdx in (2,3,4)] [colIdx in (6)] [::
    groupRegions] > -> addRoundFacade("door", ["top_left"], 0.5, 3, 0.5)}

# Offset = 0.5 and Segments = 50
{<descendant() [label=="building"] / [label=="building_front"] / [label=="
    main_front_grid"] / [type=="cell"] [rowIdx in (2,3,4)] [colIdx in (8)] [::
    groupRegions] > -> addRoundFacade("door", ["top_left"], 0.5, 50, 0.5)}
```

Figura 35 – Modelo que exemplica o uso das opções do parâmetro "offset" e "segments".



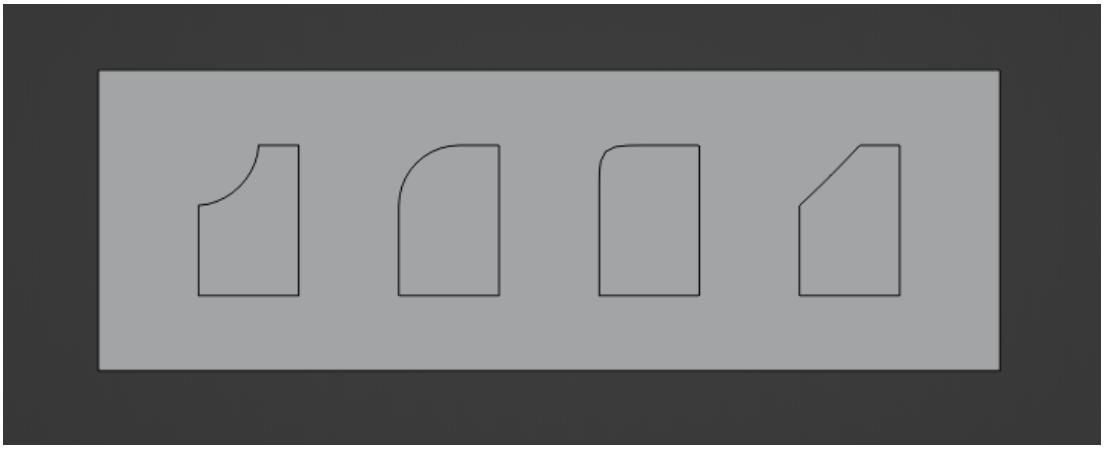
Fonte: Próprio autor

O parâmetro “profile” define a curvatura do arredondamento em relação à forma gerada e permite gerar curvas côncavas ou convexas, assim como uma reta, de acordo com o valor especificado, definido no intervalo de 0 a 1. Vale ressaltar que a especificação de um perfil com um valor maior pode afetar o deslocamento aplicado, uma vez que a curvatura será maior. A seguir, o Código-fonte 7 e a Figura 36 apresentam alguns exemplos de uso desse parâmetro.

Código-fonte 7 – Script gera diferentes aplicações para o parâmetro “offset”.

```
label = "building"; width = 15; depth = 5; height = 5;

{<> -> createShape(label, width, depth, height)};

{< descendant() [label=="building"] / [label=="building_front"] > -> createGrid("
    main_front_grid", 4, 9)};

{<descendant() [label=="building"] / [label=="building_front"] / [label=="
    main_front_grid"] / [type=="cell"] [rowIdx in (2,3)] [colIdx in (2)] [::
    groupRegions] > -> addRoundFacade("door", ["top_left"], 1, 10, 0.1)}

{<descendant() [label=="building"] / [label=="building_front"] / [label=="
    main_front_grid"] / [type=="cell"] [rowIdx in (2,3)] [colIdx in (4)] [::
    groupRegions] > -> addRoundFacade("door", ["top_left"], 1, 10, 0.5)}

{<descendant() [label=="building"] / [label=="building_front"] / [label=="
    main_front_grid"] / [type=="cell"] [rowIdx in (2,3)] [colIdx in (6)] [::
    groupRegions] > -> addRoundFacade("door", ["top_left"], 1, 10, 0.8)}

{<descendant() [label=="building"] / [label=="building_front"] / [label=="
    main_front_grid"] / [type=="cell"] [rowIdx in (2,3)] [colIdx in (8)] [::
    groupRegions] > -> addRoundFacade("door", ["top_left"], 1, 10, 0.24)}
```

Figura 36 – Aplicação do parâmetro que define o perfil do arredondamento.

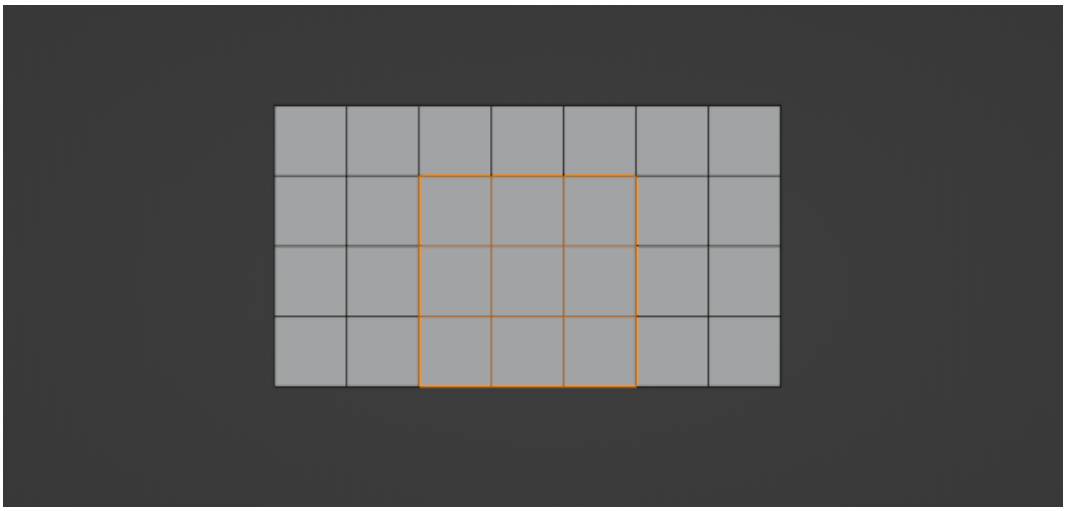


Fonte: Próprio autor

4.2.2.4 UpdateGrid

Ao gerar o modelo da Figura 33, descrito pelo Código-Fonte 4, a grid que representa a parte frontal da entrada da construção, definida por *entrance_front_grid*, é gerada no momento em que a forma *entrance* é criada pela *action addVolume*. Porém, não é especificada a quantidade de colunas e linhas que serão geradas e, consequentemente, a quantidade de células. Na verdade, é criada uma cópia da forma virtual selecionada na adição do volume pela *action addVolume*, e é feito um recorte de acordo com as linhas e colunas selecionadas. Isso limita a quantidade de células que serão criadas nas formas virtuais, uma vez que depende diretamente da forma virtual superior. Além disso, impõe ao *designer* a necessidade de definir previamente a quantidade de células que serão criadas, para que, com as adições de formas virtuais ou de construção, ainda haja células que podem ser selecionadas, o que torna o processo mais complexo. A seguir, é apresentado a representação das formas virtuais *entrance_front_grid* e *main_front_grid*.

Figura 37 – Comparação entre a *main_front_grid* e a *entrance_front_grid*, onde a segunda está sobreposta à primeira e destacada em laranja.



Fonte: Próprio autor

Uma solução seria criar novas formas virtuais, com o uso da *action createGrid*. No entanto, isso tornaria a árvore de elementos mais complexa, uma vez que a forma de construção poderia conter diversos nós filhos apenas para representar diferentes elementos de fachada. Diante disso, é introduzida uma nova *action* denominada *updateGrid*, responsável por atualizar a quantidade linhas e colunas em um forma virtual selecionada. Ela é descrita da forma:

```
updateGrid(rows, columns),
```

onde *rows* indica a quantidade de linhas que devem ser criadas e *columns*, a de colunas. Desse

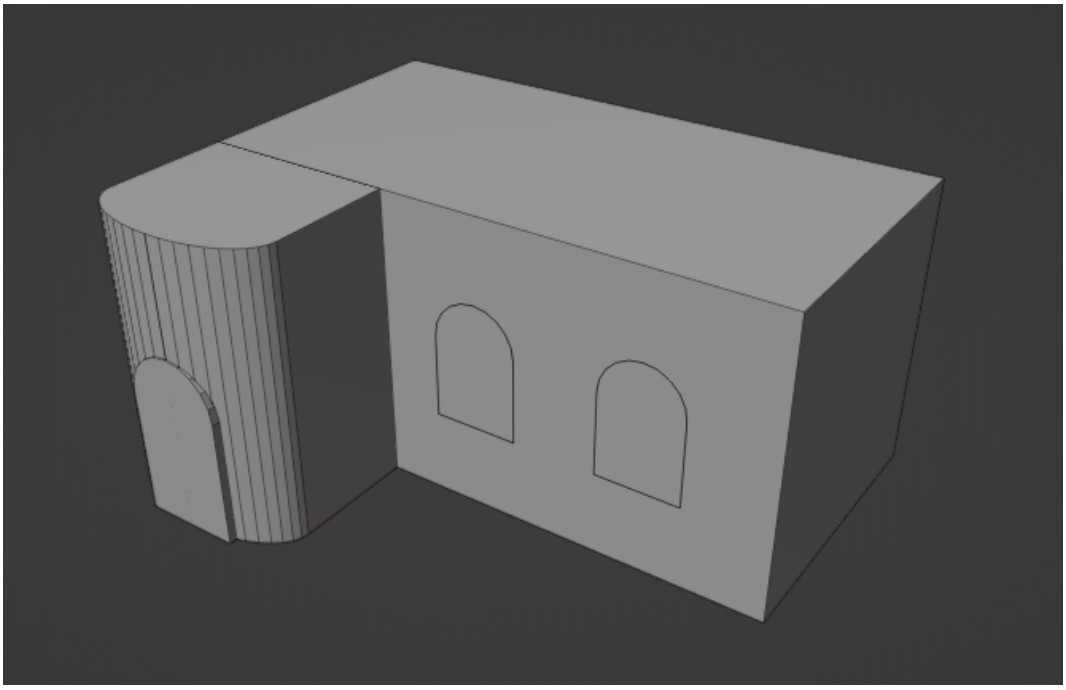
modo, é possível adicionar volume na forma de construção e aproveitar a forma virtual gerada para aplicar novas operações. O Código-fonte 8 apresenta um exemplo da aplicação da *action updateGrid*.

Código-fonte 8 – *Script* para geração de construção que utiliza a *action updateGrid*

```
#C1 Variables
label = "building"; width = 10; depth = 5; height = 5;

#C2 Create construction shape
{<> -> createShape(label, width, depth, height)};

#C3 Add virtual shape
{< descendant() [label=="building"] / [label=="building_front"] > -> createGrid("
    main_front_grid", 1, 3)};

#C4 Add volume to selected cells
{< descendant() [label=="building"] / [label=="building_front"] / [label=="
    main_front_grid"] / [type=="cell"] [rowIdx in (1)] [colIdx in (1)] [::groupRegions
    ()] > -> addVolume("entrance", "building_front", 3, ["entrance_front", "
    entrance_left", "entrance_right"])};

#C5 Add deformation in construction shape
{< descendant() [label=="building"] / [label=="building_front"] / [label=="entrance"]
    / [label=="entrance_front"] > -> roundShape("front", "outside", 0.42, 10, "
    main_front", "vertical")};

#C6 Update grid entrance_front_grid
{< descendant() [label=="building"] / [label=="building_front"] / [label=="entrance"]
    / [label=="entrance_front"] / [label=="entrance_front_grid"] > -> updateGrid(5, 4)
    };

#C7 Add round facade in construction shape
{<descendant() [label=="building"] / [label=="building_front"] / [label=="entrance"] /
     [label=="entrance_front"] / [label=="entrance_front_grid"] / [type=="cell"] [
    rowIdx in (3,4,5)] [colIdx in (2,3)] [::groupRegions] > -> addRoundFacade("door",
    ["top_left", "top_right"], 0.9, 5, 0.5)}

#C8 Update grid main_front_grid
{< descendant() [label=="building"] / [label=="building_front"] / [label=="
    main_front_grid"] > -> updateGrid(7, 7)};

#C9 Add windows in entrance lateral
{<descendant() [label=="building"] / [label=="building_front"] / [label=="
    main_front_grid"] / [type=="cell"] [rowIdx in (3,4,5)] [colIdx in (4,6)] [::
    groupRegions] > -> addRoundFacade("window", ["top_left", "top_right"], 0.9, 5,
    0.5)}
```

A primeira operação que utiliza a *updateGrid* (C6) adiciona mais células à forma virtual (*entrance_grid_front*). Sem o uso dessa *action*, a *grid* teria apenas uma célula, pois na adição do volume (C4) foi selecionada apenas a primeira célula da *main_front_grid*. Essa operação cria a *entrance_grid_front* de acordo com as células selecionadas, tornando mais simples adicionar a porta à forma de construção (C7). Note que as operações não se limitam apenas a formas virtuais geradas pela *action addVolume*, mas também às formas que são criadas diretamente pelo usuário. No segundo exemplo de uso, a quantidade de células da *grid main_front_grid* (C8) é atualizada, e em seguida são adicionados duas janelas à construção (C9). O Código-fonte 8 gera o modelo da Figura 38.

Figura 38 – Modelo gerado que exemplifica o uso da *action updateGrid*.



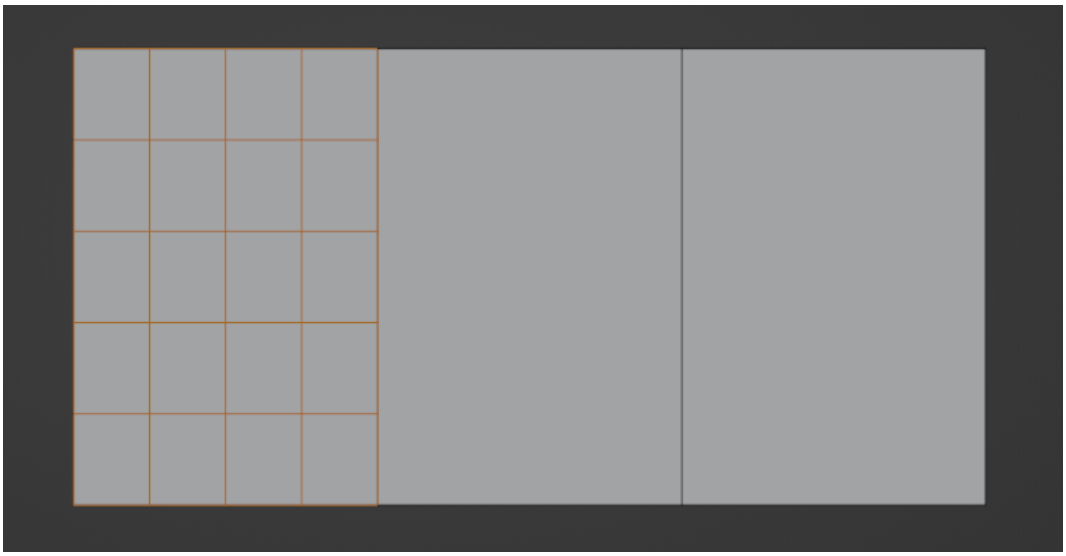
Fonte: Próprio autor

O uso da *action updateGrid* tira a dependência de uma forma virtual gerada a partir de outra com o uso da *action addVolume*, ainda que a nova seja gerada a partir de um recorte da anterior. Vale ressaltar que a aplicação dessa *action* em uma *grid* existente não contradiz a definição das formas virtuais, pois nenhuma modificação é realizada na forma de construção pai, de modo que apenas é atualizada a quantidade de células, que serão usadas exclusivamente para orientar as operações realizadas na forma de construção, em conformidade com a especificação de Jiang *et al.* (2018).

Além disso, a *action updateGrid* produz um valor semântico mais claro, pois anteriormente havia uma correlação entre a *grid* gerada e sua origem, como se houvesse um

parentesco na árvore de formas. No entanto, isso não faz sentido, uma vez que uma forma virtual não deve ter filhos na árvore de formas (Jiang *et al.*, 2018). A *action updateGrid* proporciona maior liberdade ao usuário para atualizar a *grid*, conforme necessário. A Figura 39 apresenta as formas virtuais *entrance_front_grid* e *main_front_grid*, com a aplicação da primeira operação da *updateGrid* (C6), do Código-fonte 8.

Figura 39 – Comparação entre a *main_front_grid* e a *entrance_front_grid* (sobreposta e destacada em laranja), com o uso da *action updateGrid*



Fonte: Próprio autor

4.2.2.5 addFacadeWithFrame

Algumas construções possuem a característica de janelas ou portas serem saltadas, ou seja, apresentam bordas destacadas em relação à construção. Isso pode ser observado em estruturas mais modernas, que possuem um design mais orgânico, como o apresentado na Figura 27. Da mesma forma, existem construções mais antigas que também apresentam essa característica, por exemplo, o apresentado na Figura 25.

Com isso, foi criada a *action addFacadeWithFrame*, que permite adicionar uma borda no elemento de fachada, que pode ser destacado. Ela é definida pela forma:

`addFacadeWithFrame(label, edgeExtrusion, thickness, width),`

onde:

1. **label:** Define o nome especificado para a fachada;
2. **edgeExtrusion:** Descreve o tipo da extrusão que será gerada na criação da borda, podendo ser “all”, em que a extrusão é realizada na fachada adicionada sem considerar a sua borda,

sendo aplicada de forma não perpendicular à construção base; “frame”, onde é adicionada a extrusão apenas na borda; “none”, onde não será aplicada nenhuma extrusão;

3. **thickness:** Indica a espessura da borda criada.
4. **width:** Define a distância da borda adicionada em relação a construção base. Esse parâmetro não terá efeito para o tipo “none”, da propriedade “edgeExtrusion”.

Dessa forma, elementos de fachadas com bordas podem ser adicionadas facilmente. Observe que, assim como a *action addFacade*, ele não gera os elementos com deformação, para que cada operações possua responsabilidade única, tornando assim a linguagem mais coesa. O Código-fonte 9 apresenta um exemplo do uso da *action addFacadeWithFrame*.

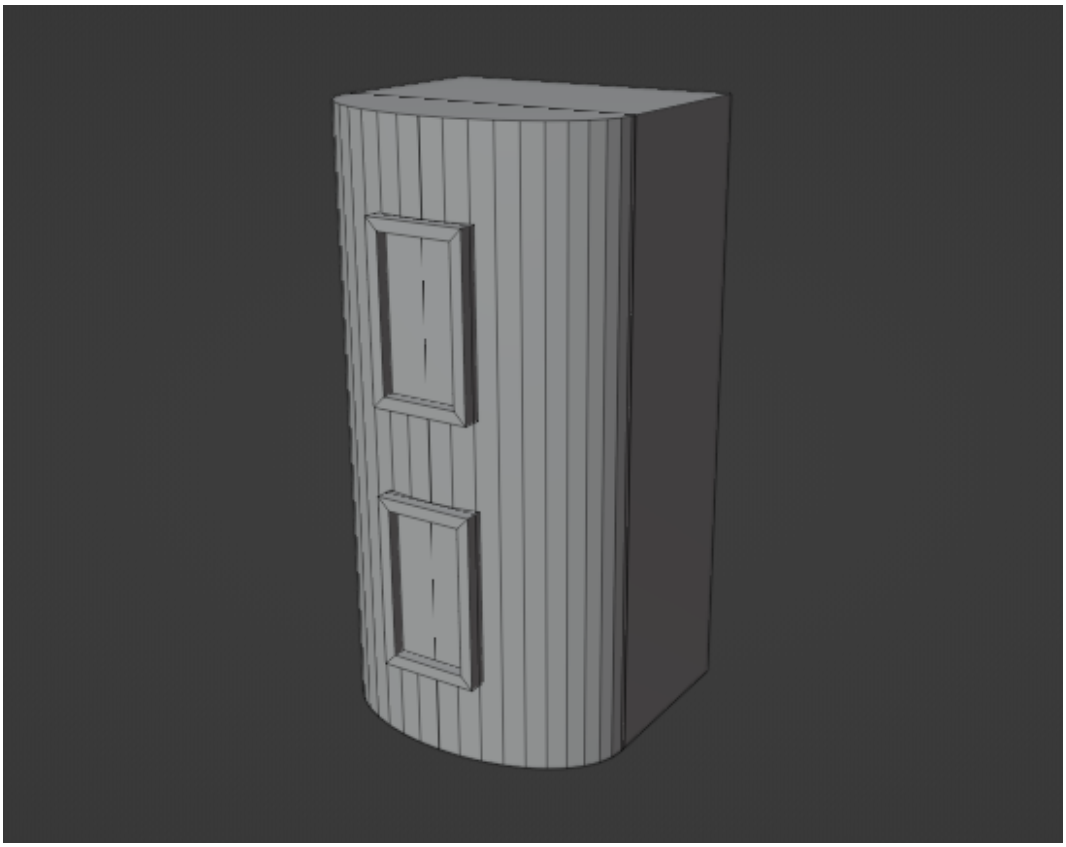
Código-fonte 9 – Regras para geração de janelas com bordas

```
#C1 Variables
label = "tower"; width = 5; depth = 3; height = 10;

#C2 Create construction shape
{<> -> createShape(label, width, depth, height)};

#C3 Add virtual shape
{< descendant() [label=="tower"] / [label=="tower_front"] > -> createGrid("
    main_front_grid", 1, 1)};

#C4 Add volume to selected cells
{< descendant() [label=="tower"] / [label=="tower_front"] / [label=="main_front_grid"]
     / [type=="cell"] [rowIdx in (1)] [colIdx in (1)] [::groupRegions()] > ->
    addVolume("entrance", "tower_front", 1.6, ["entrance_front", "entrance_left", "
    entrance_right"])};

#C5 Add deformation in construction shape
{< descendant() [label=="tower"] / [label=="tower_front"] / [label=="entrance"] / [
    label=="entrance_front"] > -> roundShape("front", "outside", 1, 10, "main_front",
    "vertical")};

#C6 Update cell's grid
{< descendant() [label=="tower"] / [label=="tower_front"] / [label=="entrance"] / [
    label=="entrance_front"] / [label=="entrance_front_grid"] > -> updateGrid(7, 6)};

#C7 Add round facade with frame around construction shape
{<descendant() [label=="tower"] / [label=="tower_front"] / [label=="entrance"] / [
    label=="entrance_front"] / [label=="entrance_front_grid"] / [type=="cell"] [rowIdx
     in (2,3,5,6)] [colIdx in (3,4)] [::groupRegions] > -> addFacadeWithFrame("windows
    ", "frame", 0.2, 0.2)}
```

Observe que a descrição da operação que adiciona as janelas com bordas destacadas

(C7) é semelhante à *action addFacade*, porém possui parâmetros adicionais. É feita a seleção da *grid*, que será a base para aplicar as modificações na forma de construção, assim como as colunas e linhas. Depois, é aplicada a *action* com o tipo de extrusão “frame". O Código-fonte 9 gera o modelo da Figura 40.

Figura 40 – Modelo gerado que aplica o uso da *action addFacadeWithFrame*



Fonte: Próprio autor

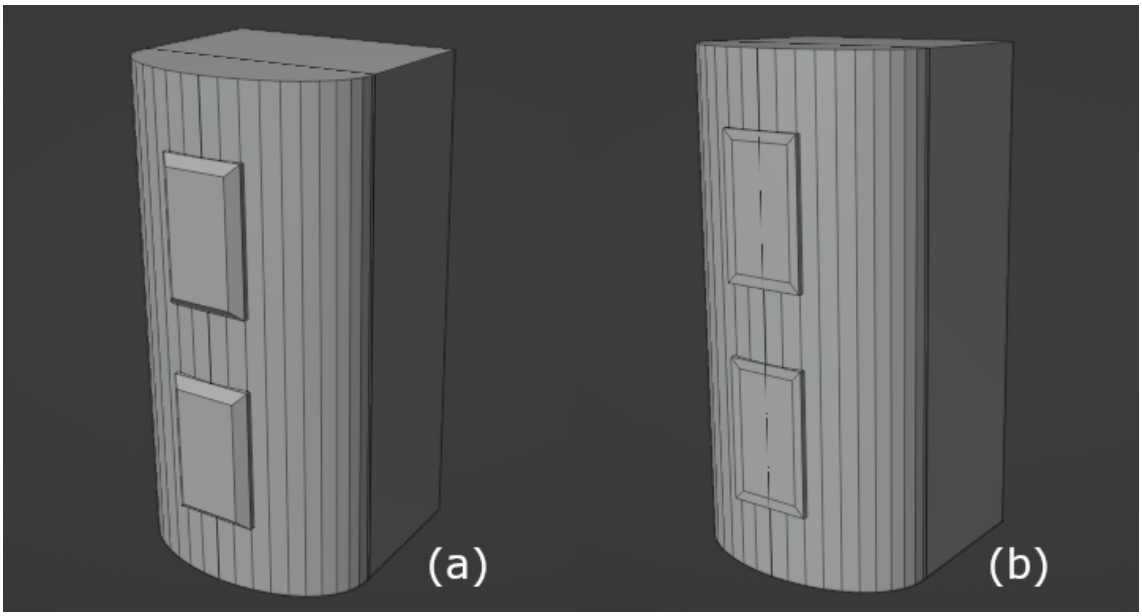
Para o tipo de extrusão especificado pelo Código-fonte 9, é realizada a extrusão apenas nas bordas da fachada adicionada. Para atingir esse efeito, é utilizada uma função presente no *blender* denominada `inset`, que adiciona novas faces à face selecionada, seguindo as suas laterais. Ademais, é utilizada a função `extrude_context_move` para realizar a extrusão das faces criadas. Isso permite que as bordas do elemento adicionado na forma sejam destacadas.

Todavia, o uso do tipo de extrusão “all” inverte a operação em comparação ao tipo anterior. Nesse caso, a extrusão é aplicada na face central do elemento adicionado, de modo que a espessura define a distância a partir do limite definido na fachada até essa face. Para esse efeito, basta utilizar um parâmetro da função `inset`, do *bpy*, que define a profundidade das faces criadas, sem a necessidade de aplicar uma extrusão explícita. A Figura 41(a) apresenta o resultado do uso desse tipo de extrusão, que possui a especificação semelhante ao do Código-fonte 9, com a

diferença sendo apenas no valor do tipo da extrusão.

O uso do tipo de extrusão “none”, por sua vez, cria apenas as bordas, sem aplicar nenhuma extrusão. Por isso, o atributo *width* não possui efeito nesse caso. Na implementação, isso é possível com a aplicação da função `inset` apenas. A Figura 41(b) exemplifica o seu uso.

Figura 41 – Modelo gerado que aplica o uso da *action addFacadeWithFrame*, com o parâmetro *edgeExtrusion* definido como “all” (direita) e “none” (esquerda).



Fonte: Próprio autor

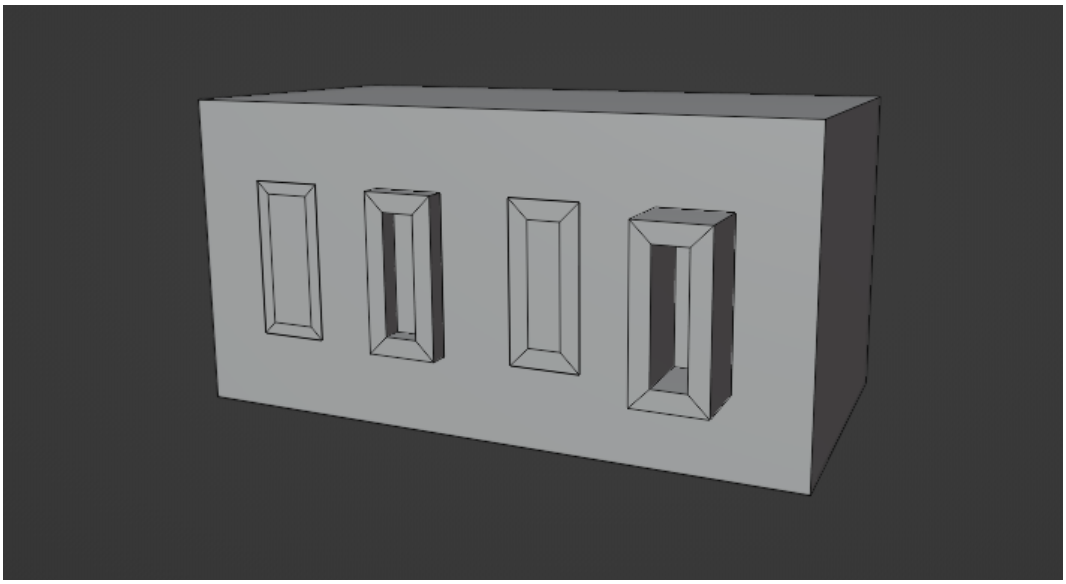
Os valores definidos para o parâmetro “thickness” e “width” são números e dependem da criatividade do usuário. A seguir será apresentado o uso desses parâmetros para diferentes valores. No caso do parâmetro “width”, quando o tipo de extrusão é definido como “none”, o valor especificado é ignorado, uma vez que indica que não haverá extrusão.

Código-fonte 10 – Exemplo das possibilidades de uso dos parâmetros “thickness” e “width”.

```
1  # Variables
2  label = "building"; width = 10; depth = 5; height = 5;
3
4  # Create construction shape
5  {<> -> createShape(label, width, depth, height)};
6
7  # Add virtual shape
8  {< descendant() [label=="building"] / [label=="building_front"] > -> createGrid("
       main_front_grid", 4, 9)};
9
10 # Thickness = 0.2
11 {descendant() [label=="building"] / [label=="building_front"] / [label=="
       main_front_grid"] / [type=="cell"] [rowIdx in (2,3)] [colIdx in (2)] [::
```

```
    groupRegions] > -> addFacadeWithFrame("windows", "none", 0.2, 0)}
12
13  # Width = 0.3
14  {descendant() [label=="building"] / [label=="building_front"] / [label=="
        main_front_grid"] / [type=="cell"] [rowIdx in (2,3)] [colIdx in (4)] [::
        groupRegions] > -> addFacadeWithFrame("windows", "frame", 0.3, 0.3)}
15
16  # Width = 0.3 and EdgeExtrusion = "none"
17  {descendant() [label=="building"] / [label=="building_front"] / [label=="
        main_front_grid"] / [type=="cell"] [rowIdx in (2,3)] [colIdx in (6)] [::
        groupRegions] > -> addFacadeWithFrame("windows", "none", 0.3, 0.3)}
18
19  # Width = 1 (Exaggerated example)
20  {descendant() [label=="building"] / [label=="building_front"] / [label=="
        main_front_grid"] / [type=="cell"] [rowIdx in (2,3)] [colIdx in (8)] [::
        groupRegions] > -> addFacadeWithFrame("windows", "frame", 0.3, 1)}
```

Figura 42 – Exemplo de uso dos parâmetros que define a espessura da borda e a distância da forma em relação a original.



Fonte: Próprio autor

4.2.2.6 addRoundFacadeWithFrame

Com o objetivo de gerar elementos de fachadas com arredondamento que contenham bordas, é definido a *action addRoundFacadeWithFrame*, descrito pela forma:

```
addRoundFacadeWithFrame(label, corners, offset, segments, profile,
  edgeExtrusion, thickness, width),
```

onde:

1. **label:** Define o nome especificado para a fachada;
2. **corners:** Descreve em quais extremidades da fachada adicionada será aplicado o arredondamento, sendo no formato de array que aceita os valores “top_left” (superior esquerdo), “top_right” (superior direito), “bottom_right” (inferior direito), “bottom_left” (inferior esquerdo);
3. **offset:** Indica a distância do deslocamento;
4. **segments:** Representa a quantidade de segmentos (faces) que serão criadas no processo da deformação;
5. **profile:** Define o perfil do elemento. Na prática, pode indicar a direção da deformação (interno ou externo);
6. **edgeExtrusion:** Descreve o tipo da extrusão que será gerada na criação da borda, podendo ser “all", em que a extrusão é realizada na fachada adicionada sem considerar a sua borda, sendo aplicada de forma não perpendicular à construção base; “frame", onde é adicionada a extrusão apenas na borda; “none", onde não será aplicada nenhuma extrusão;
7. **thickness:** Indica a espessura da borda criada.
8. **width:** Define a distância da borda adicionada em relação a construção base. Esse parâmetro não terá efeito para o tipo “none", da propriedade “edgeExtrusion".

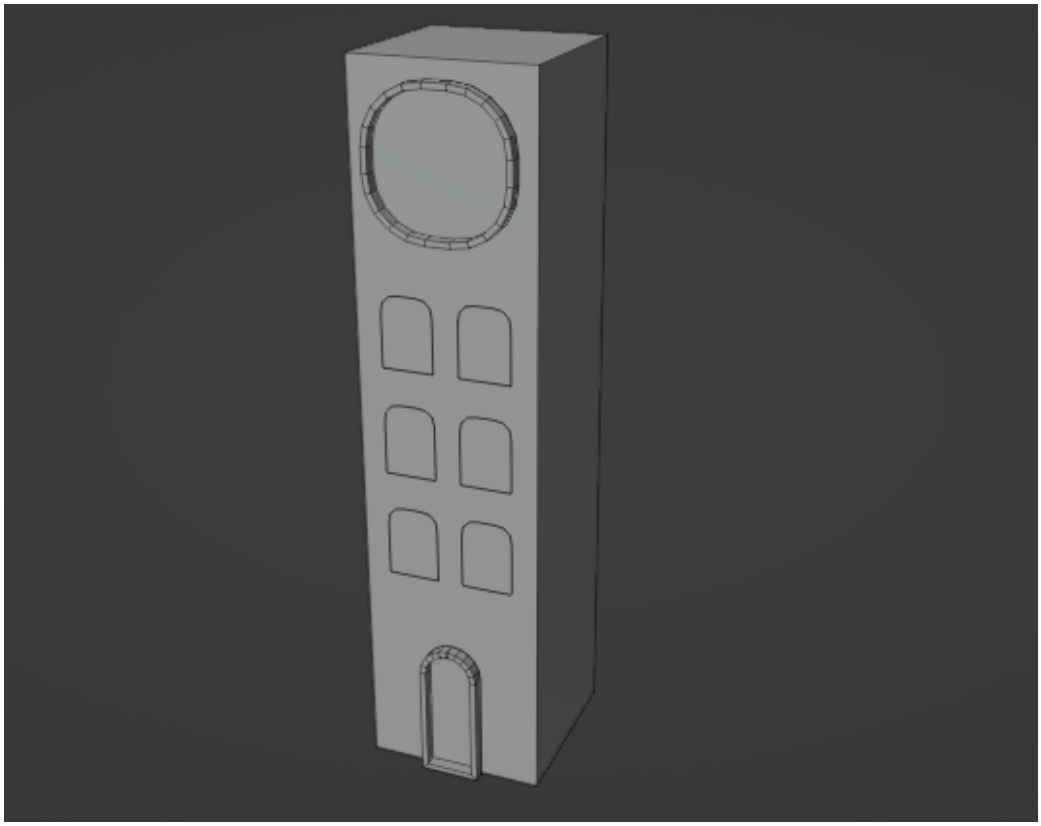
O Código-fonte 11 e a Figura 43 apresenta um exemplo do uso da *action addRoundFacadeWithFrame*.

Código-fonte 11 – Script para geração de janelas com bordas e arredondamento

```
label = "clock_tower"; width = 3; depth = 3; height = 12;

{<> -> createShape(label, width, depth, height)};

{< descendant() [label=="clock_tower"] / [label=="clock_tower_front"] > -> createGrid(
    "main_front_grid", 1, 1)};

{< descendant() [label=="clock_tower"] / [label=="clock_tower_front"] / [label=="
    main_front_grid"] > -> updateGrid(40, 10)};

{<descendant() [label=="clock_tower"] / [label=="clock_tower_front"] / [label=="
    main_front_grid"] / [type=="cell"] [pattern("X{10}(XA{8}X){8}", "A")] [::
    groupRegions] > -> addRoundFacadeWithFrame("clock", ["top_left", "top_right", "
    bottom_left", "bottom_right"], 1, 5, 0.5, "frame", 0.1, 0.2)}

{< descendant() [label=="clock_tower"] / [label=="clock_tower_front"] / [label=="
    main_front_grid"] > -> updateGrid(10, 6)};

```

```
13 {<descendant() [label=="clock_tower"] / [label=="clock_tower_front"] / [label=="
       main_front_grid"] / [type=="cell"] [rowIdx in (9,10)] [colIdx in (3,4)] [::
       groupRegions] > -> addRoundFacadeWithFrame("door", ["top_left", "top_right"], 1,
       5, 0.5, "frame", 0.1, 0.2)}
14
15 {< descendant() [label=="clock_tower"] / [label=="clock_tower_front"] / [label=="
       main_front_grid"] > -> updateGrid(20, 7)};
16
17 {<descendant() [label=="clock_tower"] / [label=="clock_tower_front"] / [label=="
       main_front_grid"] / [type=="cell"] [rowIdx in (7,8,10,11,13,14)] [colIdx in
       (2,3,5,6)] [::groupRegions] > -> addRoundFacade("windows", ["top_left", "top_right
       "], 1, 5, 0.7)}
```

Figura 43 – Modelo que aplica o uso da *action addRoundFacadeWithFrame*



Fonte: Próprio autor

De forma geral, a especificação dessa operação pode ser entendida como uma combinação das *actions addFacadeWithFrame* e *addRoundFacade*. O funcionamento é semelhante, tanto na aplicação do arredondamento, como na adição da borda. As funções utilizadas do *bpy* para alcançar esse efeito são as mesma das *actions* apresentadas anteriormente, sendo realizado primeiro o arredondamento, e em seguida a adição da borda. Desse modo, é possível obter o efeito de uma fachada com arredondamento e borda destacada.

4.2.2.7 Métodos utilitários

Com o objetivo de auxiliar a construção das *actions* criadas no presente trabalho e reaproveitar algumas operações, foram desenvolvidas funcionalidades tanto para reutilizar código como isolar uma implementação, tornando a leitura mais fácil. Dessa forma, cada função possui sua responsabilidade.

Conforme definido por Brito *et al.* (2021), a leitura das especificações do usuário é realizada por meio de um arquivo com extensão "*.slx*". Esse arquivo deve conter todas as regras que serão aplicadas nas formas. Com isso, foram desenvolvidas funções que aplicam as operações necessárias em cada regra do arquivo, para as novas *actions* criadas. Essas funções realizam a leitura das regras individualmente e recuperam os parâmetros que serão utilizados nas operações que modificam o objeto, de acordo com a descrição da *action*. São elas:

- `loadUpdateGrid`: que processa a *action UpdateGrid*;
- `loadAddFacade`: que processa a *action AddFacade*;
- `loadAddFacadeWithFrame`: que processa a *action AddFacadeWithFrame*;
- `loadAddRoundFacade`: que processa a *action AddRoundFacade*;
- `loadAddRoundFacadeWithFrame`: que processa a *action AddRoundFacadeWithFrame*.

Como a maioria das etapas para realizar a leitura das regras é igual, foram desenvolvidas funções auxiliares com o objetivo de tornar a implementação mais simples e coesa:

- `getActions`: função que é usada para recuperar os parâmetros das *actions*.
- `getColsAndRows`: utilizada para retornar as colunas e linhas selecionadas nas *selection expressions*.
- `getPattern`: empregada para recuperar os parâmetros que são descritos na operação *pattern*.
- `getSelectionLabels`: usada para agrupar e retornar todas as formas que são descritas nas *selections expressions*.
- `getGroupingType`: função empregada para validar se a operação de agrupamento `groupRegions` é utilizada.

Para auxiliar a aplicação das *actions* criadas, de acordo com as informações obtidas pelas operações descritas anteriormente, também foram desenvolvidas funções auxiliares que foram amplamente utilizadas no desenvolvimento deste trabalho, sendo as principais:

- `selectObject`: utilizada para selecionar um objeto, de acordo com o nome do

objeto.

- `deselectAllObjects`: usada para limpar a seleção de objetos;
- `selectSingleObject`: utilizada para limpar a seleção e depois selecionar o objeto de acordo com seu nome. Essa função utiliza as duas anteriores em conjunto.
- `groupFacadeWithConstruction`: função responsável por criar cada objeto que representa as fachadas individualmente e, depois, anexá-los à forma de construção principal. Para isso, é realizada uma extrusão até a posição inicial da forma de construção, de modo que exista um volume entre a construção e a fachada. Em seguida, são aplicados as bordas, se necessário e, por fim, a fachada é anexada. Essa função é utilizada por todas as *actions* que criam elementos de fachada;
- `addBooleanModifierInFacade`: é utilizada pela função anterior para remover o volume extra gerado na extrusão aplicada no objeto da fachada, para uni-lo ao objeto da construção. Para isso, é utilizada uma funcionalidade do *blender*, que aplica operações *booleanas* nos objetos selecionado;
- `addExtrusionInFrame`: função utilizada para criar as bordas das fachadas, quando necessário;
- `applyCornerRounding`: função utilitária que aplica arredondamento nos vértices de um objeto quadrilátero plano;
- `addRoundnessToFacade`: utilizada para aplicar o arredondamento nos objetos que representam as fachadas. Essa função ordena os vértices do objeto e invoca a função anterior para criar o efeito de arredondamento;
- `selectCells`: esta função é invocada em todas as *actions* e é utilizada para selecionar as células onde serão criadas as fachadas. Isso é feito de acordo com a seleção das colunas e linhas da *grid* ou pelo uso da operação *pattern*, na descrição das *selection expressions*. A seleção das células é aplicada em uma forma auxiliar, que é idêntica a forma virtual escolhida, e depois retornada, uma vez que será utilizada na função a seguir;
- `createFacadeObjs`: essa função é usada para criar os objetos que representam as fachadas, de acordo com a *grid* selecionada. Para isso, após a função anterior selecionar as células, esta função dissolve o objeto auxiliar criado em vários

(ou mesmo em apenas um, de acordo com as células escolhidas), onde cada um representa uma fachada. Essa função é utilizada em todas as *actions*.

Adicionalmente, foram desenvolvidas outras funções auxiliares para selecionar as células, para realizar a operação *pattern*. Sendo elas:

- `processAsterisk`: gera uma *substring* que aplica a operação do símbolo de asterisco;
- `processCurlyBrackets`: gera uma *substring* que aplica a operação do símbolo das chaves;
- `generateString`: função que gera uma *string* que representa as células da *grid* selecionada. Utiliza as funções anteriores para gerar as suas *substrings* e agrupa os resultados.
- `patternGrid`: a partir da *string* gerada pela função anterior, seleciona as células de acordo com o caractere indicado no segundo parâmetro da operação.

4.3 Considerações finais

Neste capítulo, foram descritas novas funcionalidades que complementam o trabalho desenvolvido por Brito *et al.* (2021), ao permitir gerar elementos de fachadas com arredondamento com o uso da ferramenta *Blender*. Para o próximo capítulo, serão apresentados exemplos de modelos de construção que ilustram o uso das operações desenvolvidas.

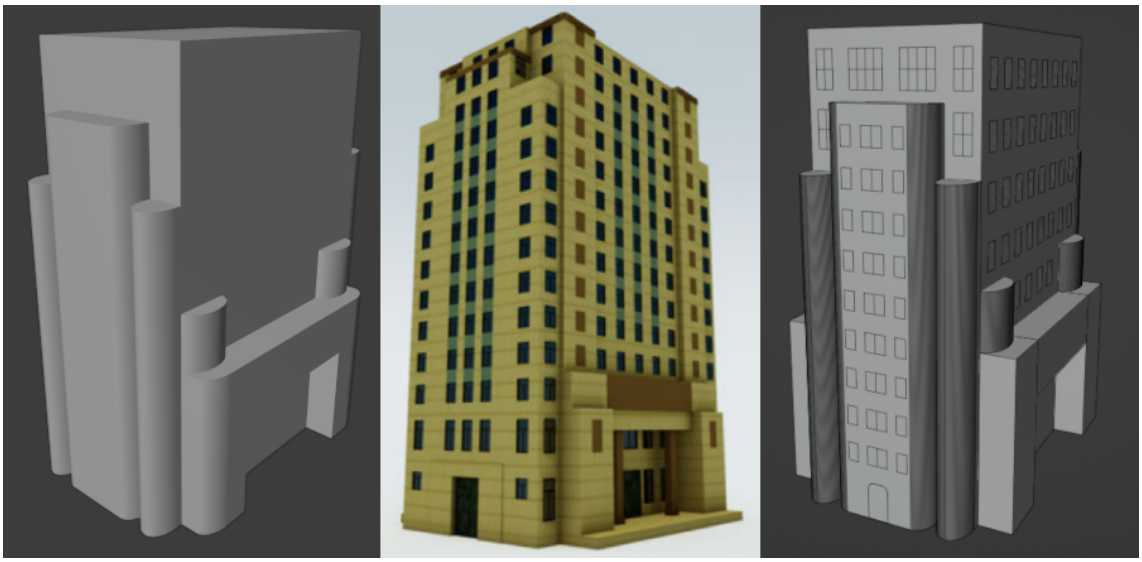
5 RESULTADOS

Neste capítulo, serão apresentados exemplos de modelos de construção, com elementos de fachada gerados a partir das funcionalidades adicionais desenvolvidas pelo presente trabalho, de modo que complementa o que foi implementado por Brito *et al.* (2021). Por fim, serão discutidas as limitações relacionadas ao desenvolvimento.

5.1 Modelos gerados

Os modelos a seguir exemplificam o uso das funcionalidades desenvolvidas, com a aplicação em diferentes construções, com diversos elementos e em diferentes posições. Com o objetivo de demonstrar a evolução, cada exemplo será apresentado com o modelo de massa, a construção de referência e o resultado final após o adição dos elementos de fachadas. Esses modelos complementam os gerados por Brito *et al.* (2021), sendo uma continuação de suas construções. O Código-fonte dos modelos está descrito no Apêndice A.

Figura 44 – Modelo de construção sem e com adição de elementos de fachada



Fonte: Brito *et al.* (2021) e próprio autor

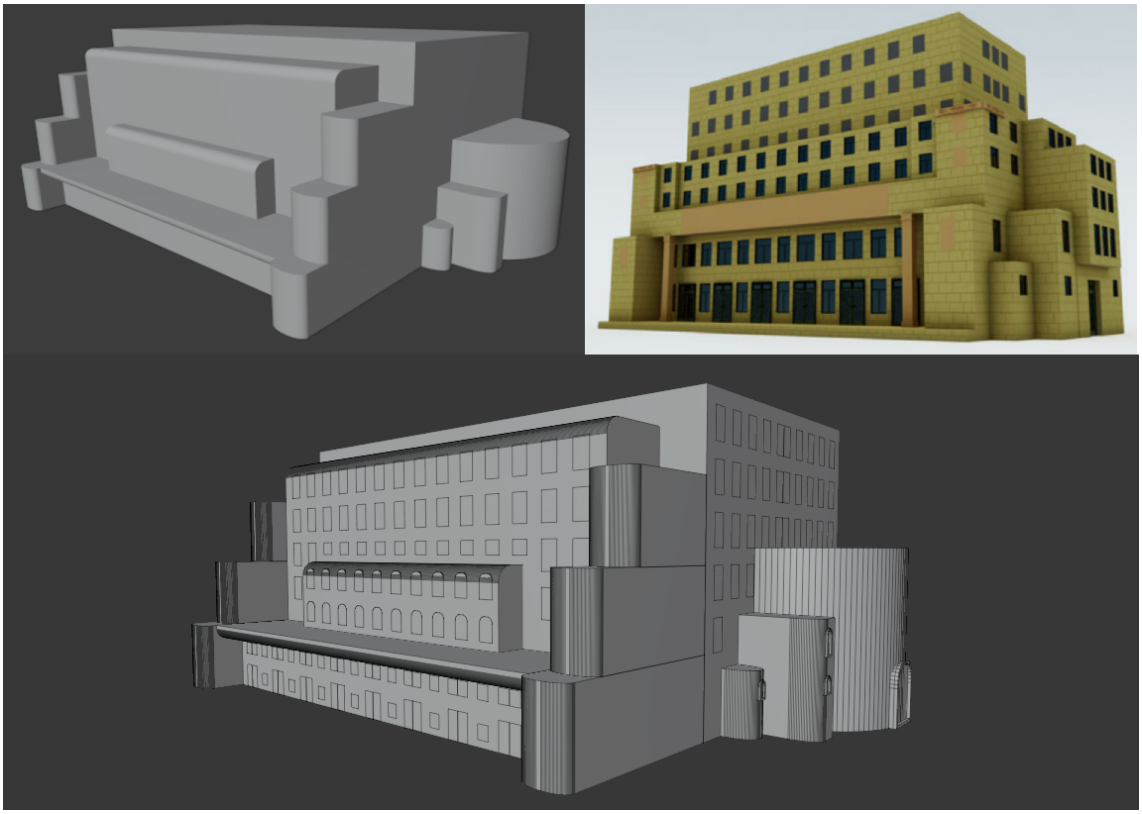
Na Figura 44, à esquerda, é apresentado o modelo gerado por Brito *et al.* (2021). À direita, está o resultado desse mesmo modelo com a adição de elementos de fachada, baseado na construção de Jiang *et al.* (2018), ao centro. Na criação do modelo, descrito no Código-fonte 12, é utilizado a *action updateGrid*, que simplifica a adição das fachadas, eliminando a necessidade de criar novas *grids* para adicionar elementos na construção. Ademais, utiliza-se a função *pattern* para seleção das células, que torna o processo mais rápido, não sendo necessário especificar

manualmente as colunas e linhas de um conjunto de células. Para isso, basta selecionar as células desejadas, considerando que cada letra representa um célula.

Nesse modelo, para gerar alguns elementos da fachada não é utilizada a função de agrupamento *groupRegions*. Logo, os elementos adjacentes não são agrupados, de modo que permite a criação de janelas vizinhas distintas, por exemplo. Isso é evidenciado pelas janelas presentes no topo do edifício.

Na Figura 45, a parte superior esquerda representa o modelo gerado por Brito *et al.* (2021), baseado na construção de Jiang *et al.* (2018), apresentado na figura superior direita. Na parte inferior, é mostrado o resultado da aplicação das fachadas para esse modelo. Observe que é demonstrado um exemplo da aplicação de fachadas em modelos de massa que possuem sobreposição de elementos, com a aplicação da *action addVolume*. Note que os elementos abaixo da construção, que representa sua entrada, estão disposto abaixo de outro elemento que está acima deles. Ainda assim, as janelas e portas são aplicadas corretamente, uma vez que cada um possui sua própria forma virtual. Além disso, esse modelo demonstra o uso das *actions addRoundFacade*, para arredondar as bordas superiores das janelas, e *addRoundFacadeWithFrame*, na criação das janelas e da porta lateral. Essas operações estão descritas no Código-fonte 13.

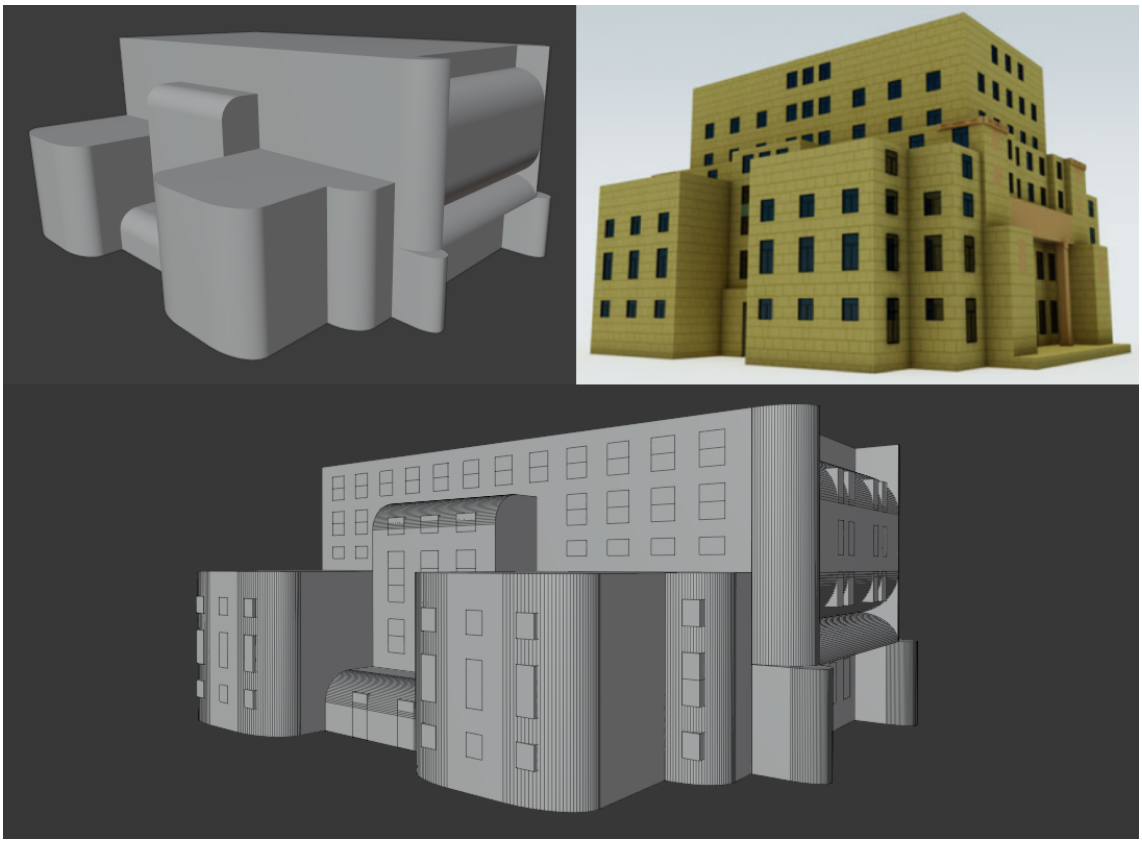
Figura 45 – Modelo que aplica o uso das *actions* de arredondamento de fachadas



Fonte: Brito *et al.* (2021) e próprio autor

A Figura 46 apresenta o comportamento de elementos de fachada, neste caso as janelas, em superfícies vertical e horizontamente arredondadas. Como já especificado anteriormente, é realizada a adição de um volume na fachada criada para permitir que seja anexada à construção principal, independente do grau de arredondamento. O modelo de massa pode ser observado na parte superior esquerda da figura, construído por Brito *et al.* (2021), onde o mesmo é apresentado com fachadas na parte inferior. Ambos os modelos são baseados no gerado por (Jiang *et al.*, 2018), apresentado na parte superior direita. O Código-fonte 14 fornece a descrição do modelo com fachada.

Figura 46 – Modelo de construção com fachadas em superfícies verticalmente arredondadas



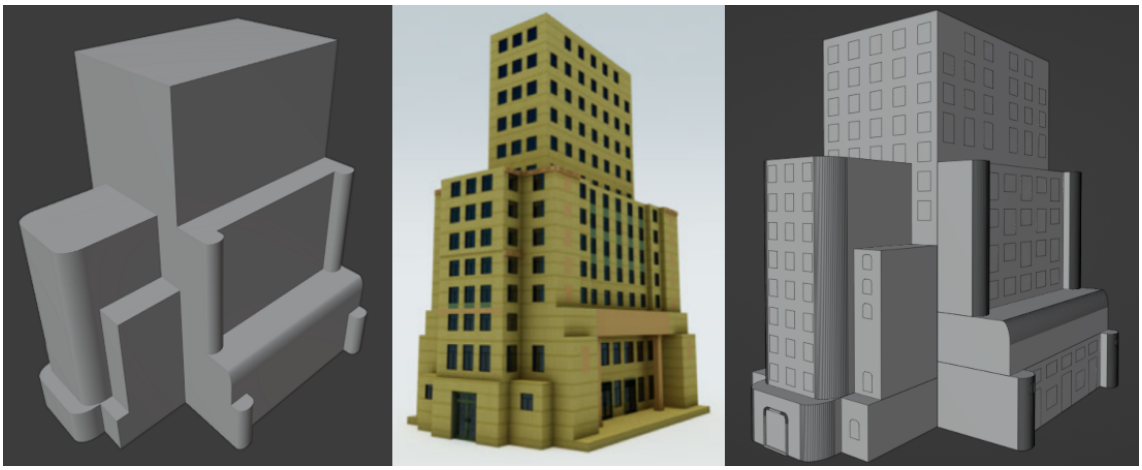
Fonte: Brito *et al.* (2021) e próprio autor

Por fim, a Figura 47 ilustra um modelo de prédio com diferentes formas de seleção de células para a geração das fachadas. Observe que nas janelas laterais é apresentado um padrão mais dinâmico, com diferentes maneiras de organização. Isso é possível devido o uso da função *pattern*, utilizando expressões mais elaboradas, que possibilita uma maior variedade de seleções e a criação de padrões mais sofisticados.

Nesse modelo, o uso da *action updateGrid* permitiu a criação simultânea de diversos elementos, sem a necessidade de criar formas virtuais adicionais. Isso facilita a criação de

fachadas com grande número de elementos, como a parte frontal da construção gerada. Na Figura 47, à esquerda, é apresentado o modelo de massa gerado por Brito *et al.* (2021), com base na construção de (Jiang *et al.*, 2018), que está no centro. À direita, é exposto o modelo com os elementos de fachada, onde as regras podem ser encontradas no Código-fonte 15.

Figura 47 – Modelo de construção com diversos padrões de janelas



Fonte: Brito *et al.* (2021) e próprio autor

5.2 Restrições

As operações desenvolvidas no presente trabalho adicionam possibilidades para a criação de construções com maior detalhamento. No entanto, ainda existem algumas restrições que devem ser observadas.

Com as funcionalidades desenvolvidas, é possível gerar facilmente elementos como janelas e portas, com base nas especificações já apresentadas. Todavia, para elementos pequenos como molduras complexas ou cornijas, o processo pode tornar-se mais desafiador, uma vez que depende totalmente da especificação das células da forma virtual. Nesses casos, pode ser necessário criar várias células para gerar esses tipos de detalhes.

Ao utilizar as *actions* desenvolvidas pelo presente trabalho, para criar elementos de fachada, não é possível especificar a posição desses elementos em locais da construção que possuem arredondamento, em relação a forma pai. Ou seja, não existe um modo de rotacionar ou ajustar a orientação dos componentes gerados na especificação da operação. As posições dos elementos são determinadas com base na *grid* e nas células selecionadas, mas não permite ajustes específicos relacionados à orientação dos elementos.

5.3 Considerações Finais

Este capítulo apresentou diversos modelos que demonstram as possibilidades para a geração de elementos de fachadas, de modo que permite a criação de uma variedade de modelos de construção com diferentes características. Esses modelos podem servir como base para a criação de estruturas mais complexas. No próximo capítulo, será apresentada a conclusão deste trabalho e possíveis temas que podem ser abordados.

6 CONCLUSÕES

Neste capítulo, serão apresentadas as considerações finais do que foi descrito no decorrer do presente trabalho. Além disso, serão propostos alguns temas para trabalhos futuros que busquem aprofundar os tópicos discutidos.

6.1 Considerações

O presente trabalho teve como objetivo apresentar uma abordagem para a geração de elementos de fachada, que parte da implementação de Brito *et al.* (2021), para gerar o modelo de massa, por meio da linguagem *SELEX*, definida por Jiang *et al.* (2018). Espera-se que as operações implementadas auxiliem na geração de elementos de fachada, servindo como base na modelagem de construções mais complexas.

Com o uso da plataforma de criação 3D *Blender*, o *designer* pode utilizar da implementação para gerar modelos base para suas criações. Com a especificação dos parâmetros, conforme descrito ao longo do presente trabalho, pode ser gerado o modelo de massa e os componentes brutos da fachada. Esses elementos podem ser refinados com as ferramentas visuais do *Blender*, ou mesmo com o uso de sua biblioteca de desenvolvimento.

Com os modelos apresentados no capítulo anterior, foram demonstradas diversas aplicações das operações desenvolvidas na geração procedural de fachadas, que permitem gerar diferentes modelos a partir das regras que forem descritas, conforme as necessidades do usuário. As alterações realizadas na implementação de Brito *et al.* (2021) foram aplicadas no seu repositório remoto[1]. Além disso, o código desenvolvido pelo presente trabalho está disponível no Apêndice B.

6.2 Trabalhos futuros

Como complemento ao que foi desenvolvido, podem ser implementadas outras operações da linguagem *SELEX*, que auxiliem na criação de elementos de fachada, como as funções de agrupamento *groupCols* e *groupEach*, descritas por Jiang *et al.* (2018). Essas funções otimizam a seleção de células nas formas virtuais, durante o processo de criação de fachadas.

Ademais, podem ser incluídos outros tipos de objetos para a criação de formas de construção, como cone ou cilindros. Esses objetos estão disponíveis na biblioteca do *Blender*,

[1] Disponível em https://github.com/DanielBrito/monografia.

o que permite que as formas de construção possam ser especificadas por meio desses objetos. Isso pode gerar mudanças na definição da *action createShape*, ou mesmo tornar necessário criar outras funções para definir essas outras formas, uma vez que atualmente essa *action* cria a forma como um prisma retangular reto. Também podem ser adicionadas funcionalidades que permite anexar formas de construção umas às outra, conforme especificado por Jiang *et al.* (2018).

Além disso, pode ser realizada a implementação de estruturas de telhados, com a criação de novas operações que realizem esse processo. Por exemplo, na especificação do *SELEX*, Jiang *et al.* (2018) define a *action finalRoof*, que é responsável por criar o telhado em uma forma de construção.

Por fim, a criação de funcionalidades que permitem ao usuário a aplicação de texturas e cores, com o uso das funções da biblioteca do *Blender*. Isso é interessante, pois permite gerar modelos de construção mais realista, com a possibilidade de simular superfícies como tijolos, madeira e vidro, por exemplo.

REFERÊNCIAS

BLENDER FOUNDATION. **Blender**. 2021. Free and Open 3D Creation Software. Disponível em: https://www.blender.org. Acesso em: 12 out. 2021.

BLENDER FOUNDATION. **Blender Python API Documentation**. 2021. Python API for Blender. Disponível em: https://docs.blender.org/api/current/index.html. Acesso em: 12 out. 2021.

BRITO, D. H. d.; VILA NOVA, A. B.; RIBEIRO, I. M. d. S. **Geração Procedural de Modelos Arquiteturais com Geometria Arredondada utilizando Selection Expressions (SELEX)**. Monografia (Graduação em Ciência da Computação) – Universidade Federal do Ceará, Campus Crateús, 2021.

CARLI, D. M. D.; BEVILACQUA, F.; POZZER, C. T.; D'ORNELLAS, M. C. A survey of procedural content generation techniques suitable to game development. In: IEEE. **2011 Brazilian Symposium on Games and Digital Entertainment**. [*S. l.*], 2011. p. 26–35.

CHOMSKY, N. Three models for the description of language. **IRE Transactions on information theory**, IEEE, v. 2, n. 3, p. 113–124, 1956.

COGO, E.; PRAZINA, I.; HODZIC, K.; HASELJIC, H.; RIZVIC, S. Survey of integrability of procedural modeling techniques for generating a complete city. In: IEEE. **2019 XXVII International Conference on Information, Communication and Automation Technologies (ICAT)**. [*S. l.*], 2019. p. 1–6.

EBERT, D. S.; MUSGRAVE, F. K.; PEACHEY, D.; PERLIN, K.; WORLEY, S. **Texturing and Modeling: A Procedural Approach**. 3rd. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002. ISBN 1558608486. Disponível em: https://dl.acm.org/doi/book/10.5555/572337. Acesso em: 22 ago. 2021.

EDELSBRUNNER, J.; HAVEMANN, S.; SOURIN, A.; FELLNER, D. W. Procedural modeling of architecture with round geometry. **Computers & Graphics**, v. 64, p. 14 – 25, 2017. ISSN 0097-8493. Cyberworlds 2016. Disponível em: http://www.sciencedirect.com/science/article/pii/S0097849317300134. Acesso em: 22 ago. 2021.

FINKENZELLER, D. Detailed building facades. **IEEE Computer Graphics and Applications**, IEEE, v. 28, n. 3, p. 58–66, 2008.

Jiang, H.; Yan, D.; Zhang, X.; Wonka, P. Selection expressions for procedural modeling. **IEEE Transactions on Visualization and Computer Graphics**, v. 26, n. 4, p. 1775–1788, 2018. Disponível em: https://ieeexplore.ieee.org/document/8502874. Acesso em: 22 ago. 2021.

KELLY, T.; WONKA, P. Interactive architectural modeling with procedural extrusions. **ACM Transactions on Graphics (TOG)**, ACM New York, NY, USA, v. 30, n. 2, p. 1–15, 2011.

KLAVDIANOS, P.; ZHANG, Q.; IZQUIERDO, E. A concise survey for 3d reconstruction of building façades. In: IEEE. **2013 14th International Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS)**. [*S. l.*], 2013. p. 1–4.

KNIGHT, T. W. Shape grammars in education and practice: history and prospects. **International Journal of Design Computing**, Sydney, 2000. Disponível em: http://www.mit.edu/~tknight/IJDC. Acesso em: 22 ago. 2021.

MARSON, F.; JUNG, C.; MUSSE, S. Modelagem procedural de cidades virtuais. In: **Symposium on Virtual Reality**. [*S. l.*: *s. n.*], 2003. p. 103–116.

MüLLER, P.; WONKA, P.; HAEGLER, S.; ULMER, A.; GOOL, L. V. Procedural modeling of buildings. **ACM Trans. Graph.**, Association for Computing Machinery, New York, NY, USA, v. 25, n. 3, p. 614–623, jul. 2006. ISSN 0730-0301. Disponível em: https://doi.org/10.1145/1141911.1141931. Acesso em: 22 ago. 2021.

PARISH, Y. I.; MÜLLER, P. Procedural modeling of cities. In: **Proceedings of the 28th annual conference on Computer graphics and interactive techniques**. [*S. l.*: *s. n.*], 2001. p. 301–308.

PYTHON SOFTWARE FOUNDATION. **Python Language Site: Documentation**. 2021. Página de documentação. Disponível em: https://www.python.org/doc. Acesso em: 12 out. 2021.

RODRIGUES, F. C. M.; NETO, J. B. C.; VIDAL, C. A. Split grammar evolution for procedural modeling of virtual buildings. In: IEEE. **2015 XVII Symposium on Virtual and Augmented Reality**. São Paulo, 2015. p. 75–83.

SCHWARZ, M.; MüLLER, P. Advanced procedural modeling of architecture. **ACM Trans. Graph.**, Association for Computing Machinery, New York, NY, USA, v. 34, n. 4, jul. 2015. ISSN 0730-0301. Disponível em: https://doi.org/10.1145/2766956. Acesso em: 22 ago. 2021.

SILVEIRA, I.; CAMOZZATO, D.; MARSON, F.; DIHL, L.; MUSSE, S. R. Real-time procedural generation of personalized facade and interior appearances based on semantics. In: IEEE. **2015 14th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)**. [*S. l.*], 2015. p. 89–98.

SMELIK, R. M.; KRAKER, K. J. D.; TUTENEL, T.; BIDARRA, R.; GROENEWEGEN, S. A. A survey of procedural methods for terrain modelling. In: **Proceedings of the CASA Workshop on 3D Advanced Media In Gaming And Simulation (3AMIGAS)**. [*S. l.*: *s. n.*], 2009. v. 2009, p. 25–34.

SMELIK, R. M.; TUTENEL, T.; BIDARRA, R.; BENES, B. A survey on procedural modeling for virtual worlds. **Comput. Graph. Forum**, The Eurographs Association & John Wiley & Sons, Ltd., Chichester, GBR, v. 33, n. 6, p. 31–50, set. 2014. ISSN 0167-7055. Disponível em: https://doi.org/10.1111/cgf.12276. Acesso em: 22 ago. 2021.

SMITH, A. R. Plants, fractals, and formal languages. **ACM SIGGRAPH Computer Graphics**, ACM New York, NY, USA, v. 18, n. 3, p. 1–10, 1984.

STINY, G. Introduction to shape and shape grammars. **Environment and planning B: planning and design**, SAGE Publications Sage UK: London, England, v. 7, n. 3, p. 343–351, 1980.

STINY, G.; GIPS, J. Shape grammars and the generative specification of painting and sculpture. In: **IFIP congress (2)**. [*S. l.*: *s. n.*], 1971. v. 2, n. 3, p. 125–135.

THALLER, W.; KRISPEL, U.; ZMUGG, R.; HAVEMANN, S.; FELLNER, D. W. Shape grammars on convex polyhedra. **Computers & Graphics**, Elsevier, v. 37, n. 6, p. 707–717, 2013.

WONKA, P.; WIMMER, M.; SILLION, F.; RIBARSKY, W. Instant architecture. In: **ACM SIGGRAPH 2003 Papers**. New York, NY, USA: Association for Computing Machinery, 2003. (SIGGRAPH '03), p. 669–677. ISBN 1581137095. Disponível em: https://doi.org/10.1145/1201775.882324. Acesso em: 22 ago. 2021.

APÊNDICE A – ESPECIFICAÇÃO DAS REGRAS PARA GERAÇÃO DE CONSTRUÇÕES

A seguir serão descritas as regras utilizadas para a geração dos modelos apresentados no Capítulo 5. As regras definem apenas as operações para gerar os elementos de fachada. Para que o modelo seja gerado corretamente, é necessário previamente especificar o modelo de massa.

Para complementar o modelo descrito a seguir, é necessário adicionar as regras para a geração do modelo de massa, que pode ser obtido atráves de um repositório remoto[1].

Código-fonte 12 – Especificação do modelo gerado da Figura 44

```
1  {< descendant() [label=="building"] / [label=="building_front"] / [label=="south_3"] /
       [label=="south_3_front"] / [label=="south_3_front_grid"] > -> updateGrid(21, 10)}
2
3  {< descendant() [label=="building"] / [label=="building_front"] / [label=="south_3"] /
       [label=="south_3_front"] / [label=="south_3_front_grid"] / [type=="cell"] [
      pattern("X{10}(XXAXAAXAXXX{10})*X{20}", "A")] > -> addFacade("windows_front")}
4
5  {< descendant() [label=="building"] / [label=="building_front"] / [label=="south_3"] /
       [label=="south_3_front"] / [label=="south_3_front_grid"] / [type=="cell"] [::
      groupRegions] [pattern("(X)*(X{4}A{2}X{4}){2}", "A")] > -> addRoundFacade("door",
      ["top_left", "top_right"], 0.35, 5, 0.7)}
6
7  {< descendant() [label=="building"] / [label=="building_front"] / [label=="
      main_front_grid"] > -> updateGrid(45, 20)}
8
9  {< descendant() [label=="building"] / [label=="building_front"] / [label=="
      main_front_grid"] / [type=="cell"] [pattern("(X{20}(XAAAXA{10}XAAAX){2}){2}(X{20}(
      XAAX{14}AAX){2}){2}", "A")] > -> addFacade("windows_front_top")}
10
11 {< descendant() [label=="building"] / [label=="building_left"] / [label=="
      main_left_grid"] > -> updateGrid(17, 17)}
12
13 {< descendant() [label=="building"] / [label=="building_left"] / [label=="
      main_left_grid"] / [type=="cell"] [::groupRegions] [pattern("(X{17}(XA){8}X){4}X
      {22}(AX){4}", "A")] > -> addFacade("windows_left")}
14
15 {< descendant() [label=="building"] / [label=="building_right"] / [label=="
      main_right_grid"] > -> updateGrid(17, 17)}
16
17 {< descendant() [label=="building"] / [label=="building_right"] / [label=="
      main_right_grid"] / [type=="cell"] [::groupRegions] [pattern("(X{17}(XA){8}X){4}X
      {22}(AX){4}", "A")] > -> addFacade("windows_right")}
```

[1] Disponível em https://github.com/DanielBrito/monografia/tree/main/Resultados/Modelo_04.

No Código-Fonte a seguir, foi necessário realizar alterações em regras definidas no modelo existente[2], devido às nomenclaturas utilizadas no processo de extrusão, realizado pela *action addVolume*.

Código-fonte 13 – Especificação do modelo gerado da Figura 45

```
#C24: Selecting region and performing extrusion
{< descendant() [label=="building"] / [label=="building_right"] / [label=="
    main_right_grid"] / [type=="cell"] [rowIdx in (17, 18, 19, 20)] [colIdx in (2)]
    [::groupRegions()] > -> addVolume("east_1", "building_right", 1, ["east_1_1", "
    east_1_2", "east_1_3"])};

#C25: Applying roundShape deformation
{< descendant() [label=="building"] / [label=="building_right"] / [label=="east_1"] /
    [label=="east_1_1"] > -> roundShape("front", "outside", 0.09, 30, "main_right", "
    vertical")};

#C26: Selecting region and performing extrusion
{< descendant() [label=="building"] / [label=="building_right"] / [label=="
    main_right_grid"] / [type=="cell"] [rowIdx in (indexRange(14, 20))] [colIdx in (3)
    ] [::groupRegions()] > -> addVolume("east_2", "building_right", 2.5, ["east_2_1",
    "east_2_2", "east_2_3"])};

#C27: Applying roundShape deformation
{< descendant() [label=="building"] / [label=="building_right"] / [label=="east_2"] /
    [label=="east_2_1"] > -> roundShape("left", "outside", 0.09, 30, "main_right")};

#C28: Selecting region and performing extrusion
{< descendant() [label=="building"] / [label=="building_right"] / [label=="
    main_right_grid"] / [type=="cell"] [rowIdx in (indexRange(14, 20))] [colIdx in
    (10)] [::groupRegions()] > -> addVolume("east_3", "building_right", 2.5, ["
    east_3_1", "east_3_2", "east_3_3"])};

#C29: Applying roundShape deformation
{< descendant() [label=="building"] / [label=="building_right"] / [label=="east_3"] /
    [label=="east_3_1"] > -> roundShape("right", "outside", 0.09, 30, "main_right")};

#C30: Selecting region and performing extrusion
{< descendant() [label=="building"] / [label=="building_right"] / [label=="
    main_right_grid"] / [type=="cell"] [rowIdx in (indexRange(10, 20))] [colIdx in (
    indexRange(4, 9))] [::groupRegions()] > -> addVolume("east_4", "building_right",
    3.5, ["east_4_1", "east_4_2", "east_4_3"])};

#C31: Applying roundShape deformation
{< descendant() [label=="building"] / [label=="building_right"] / [label=="east_4"] /
    [label=="east_4_1"] > -> roundShape("front", "outside", 0.6, 30, "main_right", "
    vertical")};
```

[2] Disponível em https://github.com/DanielBrito/monografia/tree/main/Resultados/Modelo_05.

```
24
25 # ------------------------------------------------------
26
27 {< descendant() [label=="building"] / [label=="building_front"] / [label=="south_3"] /
       [label=="south_3_front"] / [label=="south_3_front_grid"] > -> updateGrid(9, 41)}
28
29 {< descendant() [label=="building"] / [label=="building_front"] / [label=="south_3"] /
       [label=="south_3_front"] / [label=="south_3_front_grid"] / [type=="cell"] [
      pattern("X{41}((XAXXA){8}X){2}X{82}(XAXXX){8}X((XAXXA){8}X){2}(XAXXX){8}X", "A")]
      [::groupRegions()] > -> addFacade("front_part1")}
30
31 {< descendant() [label=="building"] / [label=="building_front"] / [label=="south_3"] /
       [label=="south_3_front"] / [label=="south_3_front_grid"] / [type=="cell"] [
      pattern("X{41}((XXAXX){8}X){2}X{82}((XXAXX){8}X){4}", "A")] [::groupRegions()] >
      -> addFacade("front_part2")}
32
33 {< descendant() [label=="building"] / [label=="building_front"] / [label=="south_10"]
      / [label=="south_10_front"] / [label=="south_10_front_grid"] > -> updateGrid(10,
      40)}
34
35 {< descendant() [label=="building"] / [label=="building_front"] / [label=="south_10"]
      / [label=="south_10_front"] / [label=="south_10_front_grid"] / [type=="cell"] [
      pattern("X{40}(X(AAXX){9}AAX){3}X{80}(X(AAXX){9}AAX){3}", "A")] [::groupRegions()]
       > -> addRoundFacade("windows_1", ["top_right", "top_left"], "none", 0.2, 10, 0.5)
      }
36
37 {< descendant() [label=="building"] / [label=="building_front"] / [label=="south_7"] /
       [label=="south_7_front"] / [label=="south_7_front_grid"] > -> updateGrid(14, 29)}
38
39 {< descendant() [label=="building"] / [label=="building_front"] / [label=="south_7"] /
       [label=="south_7_front"] / [label=="south_7_front_grid"] / [type=="cell"] [
      pattern("(X{29}(X(AX){14}){2}){2}X{29}X(AX){14}XAXAX{20}XAXAXX{29}(XAXAX{20}XAXAX)
      {2}", "A")] [::groupRegions()] > -> addFacade("windows_2")}
40
41 {< descendant() [label=="building"] / [label=="building_right"] / [label=="east_1"] /
      [label=="east_1_1"] / [label=="east_1_1_grid"] > -> updateGrid(7, 3)}
42
43 {< descendant() [label=="building"] / [label=="building_right"] / [label=="east_1"] /
      [label=="east_1_1"] / [label=="east_1_1_grid"] / [type=="cell"] [pattern("(X{3}(
      XAX){2}", "A")] [::groupRegions()] > -> addRoundFacadeWithFrame("east_windows_1",
      ["top_left", "top_right"], 1, 5, 0.5, "frame", 0.05, 0.05)
44
45 {< descendant() [label=="building"] / [label=="building_right"] / [label=="east_2"] /
      [label=="east_2_1"] / [label=="east_2_1_grid"] > -> updateGrid(13, 3)
46
47 {< descendant() [label=="building"] / [label=="building_right"] / [label=="east_2"] /
      [label=="east_2_1"] / [label=="east_2_1_grid"] / [type=="cell"] [pattern("(X{3}(
      XAX){3}X{6}(XAX){2}", "A")] [::groupRegions()] > -> addRoundFacadeWithFrame("
      east_windows_2", ["top_left", "top_right"], 1, 5, 0.5, "frame", 0.05, 0.05)
```

```
48
49 {< descendant() [label=="building"] / [label=="building_right"] / [label=="east_3"] /
       [label=="east_3_1"] / [label=="east_3_1_grid"] > -> updateGrid(13, 3)
50
51 {< descendant() [label=="building"] / [label=="building_right"] / [label=="east_3"] /
       [label=="east_3_1"] / [label=="east_3_1_grid"] / [type=="cell"] [pattern("(X{3}(
       XAX){3}X{6}(XAX){2}", "A")] [::groupRegions()] > -> addRoundFacadeWithFrame("
       east_windows_3", ["top_left", "top_right"], 1, 5, 0.5, "frame", 0.05, 0.05)
52
53 {< descendant() [label=="building"] / [label=="building_right"] / [label=="east_4"] /
       [label=="east_4_1"] / [label=="east_4_1_grid"] > -> updateGrid(11, 8)
54
55 {< descendant() [label=="building"] / [label=="building_right"] / [label=="east_4"] /
       [label=="east_4_1"] / [label=="east_4_1_grid"] / [type=="cell"] [pattern("((X{8})
       {7}(XXXAAXXX){4}", "A")] [::groupRegions()] > -> addRoundFacadeWithFrame("
       east_windows_3", ["top_left", "top_right"], 1, 10, 0.6, "frame", 0.09, 0.1)
56
57 {< descendant() [label=="building"] / [label=="building_right"] / [label=="
       main_right_grid"] > -> updateGrid(20, 20)
58
59 {< descendant() [label=="building"] / [label=="building_right"] / [label=="
       main_right_grid"] / [type=="cell"] [pattern("(X{20}((XA){4}XAX(XA){4}X){2}){3}X
       {20}((XA){3}X{12}AX){2}X{20}XAX{18}XA", "A")] [::groupRegions()] > -> addFacade("
       east_windows")}
60
61 {< descendant() [label=="building"] / [label=="building_right"] / [label=="
       main_right_grid"] / [type=="cell"] [pattern("(X{20}(X{10}AX{9}){2}){3}", "A")] [::
       groupRegions()] > -> addFacade("east_windows_pt2")}
```

Para a geração correta do modelo descrito pelo Código-Fonte a seguir, deve ser adicionada a especificação do modelo de massa, disponibilizada em endereço eletrônico[3].

Código-fonte 14 – Especificação do modelo gerado da Figura 46

```
1 {< descendant() [label=="building"] / [label=="building_front"] / [label=="south_1"] /
       [label=="south_1_front"] / [label=="south_1_front_grid"] > -> updateGrid(18, 11)}
2
3 {< descendant() [label=="building"] / [label=="building_front"] / [label=="south_1"] /
       [label=="south_1_front"] / [label=="south_1_front_grid"] / [type=="cell"] [
       pattern("(X{33}(X(XAX){3}X){2}x{22}(X(XAX){3}X){4}X{22}(X(XAX){3}X){2}", "A")] [::
       groupRegions()] > -> addFacade("facade_front_1")}
4
5 {< descendant() [label=="building"] / [label=="building_front"] / [label=="south_2"] /
       [label=="south_2_front"] / [label=="south_2_front_grid"] > -> updateGrid(18, 11)}
6
```

[3] Disponível em https://github.com/DanielBrito/monografia/tree/main/Resultados/Modelo_06.

```
7  {< descendant() [label=="building"] / [label=="building_front"] / [label=="south_2"] /
       [label=="south_2_front"] / [label=="south_2_front_grid"] / [type=="cell"] [
     pattern("(X{33}(X(XAX){3}X){2}x{22}(X(XAX){3}X){4}X{22}(X(XAX){3}X){2}", "A")] [::
     groupRegions()] > -> addFacade("facade_front_2")}
8
9  {< descendant() [label=="building"] / [label=="building_front"] / [label=="south_4"] /
       [label=="south_4_front"] / [label=="south_4_front_grid"] > -> updateGrid(4, 8)}
10
11 {< descendant() [label=="building"] / [label=="building_front"] / [label=="south_4"] /
       [label=="south_4_front"] / [label=="south_4_front_grid"] / [type=="cell"] [
     pattern("X{8}(XXAXXAXX){3}", "A")] [::groupRegions()] > -> addFacade("doors_front"
     )}
12
13 {< descendant() [label=="building"] / [label=="building_front"] / [label=="south_5"] /
       [label=="south_5_front"] / [label=="south_5_front_grid"] > -> updateGrid(10, 7)}
14
15 {< descendant() [label=="building"] / [label=="building_front"] / [label=="south_5"] /
       [label=="south_5_front"] / [label=="south_5_front_grid"] / [type=="cell"] [
     pattern("X{7}X(AX){3}X{7}(X(AX){3}){3}X{7}(X(AX){3}){2}", "A")] > -> addFacade("
     windows_front")}
16
17 {< descendant() [label=="building"] / [label=="building_front"] / [label=="south_3"] /
       [label=="south_3_front"] / [label=="south_3_front_grid"] > -> updateGrid(8, 3)}
18
19 {< descendant() [label=="building"] / [label=="building_front"] / [label=="south_3"] /
       [label=="south_3_front"] / [label=="south_3_front_grid"] / [type=="cell"] [
     pattern("X{3}XAXX{3}(XAX){2}X{3}XA", "A")] > -> addFacade("windows_3")}
20
21 {< descendant() [label=="building"] / [label=="building_front"] / [label=="
     main_front_grid"] > -> updateGrid(20, 25)}
22
23 {< descendant() [label=="building"] / [label=="building_front"] / [label=="
     main_front_grid"] / [type=="cell"] [pattern("X{25}((XA){12}X){2}X{25}((XA){3}X
     {10}(XA){4}X){2}X{25}(XA){3}X{10}(XA){4}", "A")] > -> addFacade("windows_4")}
24
25 {< descendant() [label=="building"] / [label=="building_right"] / [label=="east_6"] /
       [label=="east_6_1"] / [label=="east_6_1_grid"] > -> updateGrid(10, 16)}
26
27 {< descendant() [label=="building"] / [label=="building_right"] / [label=="east_6"] /
       [label=="east_6_1"] / [label=="east_6_1_grid"] / [type=="cell"] [pattern("(X{16}(X
     {3}AXAX{4}AXAX{3}){2}){3}", "A")] [::groupRegions()] > -> addFacade("east_windows"
     )}
28
29 {< descendant() [label=="building"] / [label=="building_right"] / [label=="
     main_right_grid"] > -> updateGrid(21, 13)}
30
31 {< descendant() [label=="building"] / [label=="building_right"] / [label=="
     main_right_grid"] / [type=="cell"] [pattern("(X)*(XX(AX){5}X){3}X{26}", "A")] [::
     groupRegions()] > -> addFacade("east_windows_2")}
```

Da mesma forma que o Código-Fonte 13, foram alteradas algumas regras devido à *action addVolume*. Elas precisam ser atualizadas nas regras que geram o modelo de massa[4], assim como nas regras para a criação de fachadas.

Código-fonte 15 – Especificação do modelo gerado da Figura 47

```
# Regras que devem ser atualizadas das regras do Modelo de Massa

#C13: Selecting region and performing extrusion
{< descendant() [label=="building"] / [label=="building_right"] / [label=="
    main_right_grid"] / [type=="cell"] [rowIdx in (22, 23, 24, 25)] [colIdx in (1)]
    [::groupRegions()] > -> addVolume("east_1", "building_right", 6, ["east_1_1", "
    east_1_2", "east_1_3"])};

#C15: Selecting region and performing extrusion
{< descendant() [label=="building"] / [label=="building_right"] / [label=="
    main_right_grid"] / [type=="cell"] [rowIdx in (22, 23, 24, 25)] [colIdx in (10)]
    [::groupRegions()] > -> addVolume("east_2", "building_right", 6, ["east_2_1", "
    east_2_2", "east_2_3"])};

#C17: Selecting region and performing extrusion
{< descendant() [label=="building"] / [label=="building_right"] / [label=="
    main_right_grid"] / [type=="cell"] [rowIdx in (22, 23, 24, 25)] [colIdx in (
    indexRange(2, 9))] [::groupRegions()] > -> addVolume("east_3", "building_right",
    5, ["east_3_1", "east_3_2", "east_3_3"])};

#C24: Selecting region and performing extrusion
{< descendant() [label=="building"] / [label=="building_right"] / [label=="
    main_right_grid"] / [type=="cell"] [rowIdx in (indexRange(10, 18))] [colIdx in (
    indexRange(2, 9))] [::groupRegions()] > -> addVolume("east_7", "building_right",
    2, ["east_7_1", "east_7_2", "east_7_3"])};

# ------------------------------------------------
# Regras que devem ser adicionadas

{< descendant() [label=="building"] / [label=="building_front"] / [label=="south_1"] /
     [label=="south_1_front"] / [label=="south_1_front_grid"] > -> updateGrid(4, 6)}

{< descendant() [label=="building"] / [label=="building_front"] / [label=="south_1"] /
     [label=="south_1_front"] / [label=="south_1_front_grid"] / [type=="cell"] [
    pattern("X{6}(XXAAXX){3}", "A")] [::groupRegions()] > -> addRoundFacadeWithFrame("
    door", ["top_left", "top_right"], 0.5, 20, 0.7, "frame", 0.1, 0.1)}

{< descendant() [label=="building"] / [label=="building_front"] / [label=="south_2"] /
     [label=="south_2_front"] / [label=="south_2_front_grid"] > -> updateGrid(40, 16)}

```

[4] Disponível em https://github.com/DanielBrito/monografia/tree/main/Resultados/Modelo_07.

```
{< descendant() [label=="building"] / [label=="building_front"] / [label=="south_2"] / [label=="south_2_front"] / [label=="south_2_front_grid"] / [type=="cell"] [pattern("X{16}((XX(XAAX){3}XX){3}X{32}){8}", "A")] [::groupRegions()] > -> addFacade("windows_front")}

{< descendant() [label=="building"] / [label=="building_front"] / [label=="south_5"] / [label=="south_5_front"] / [label=="south_5_front_grid"] > -> updateGrid(30, 6)}

{< descendant() [label=="building"] / [label=="building_front"] / [label=="south_5"] / [label=="south_5_front"] / [label=="south_5_front_grid"] / [type=="cell"] [pattern("X{6}((XXAAXX){3}X{12}){3}(X)*(XXAAXX){4}X{6}", "A")] [::groupRegions()] > -> addRoundFacade("windows_front_1_top", ["top_left", "top_right"], 0.2, 5, 0.5)}

{< descendant() [label=="building"] / [label=="building_front"] / [label=="south_7"] / [label=="south_7_front"] / [label=="south_7_front_grid"] > -> updateGrid(30, 6)}

{< descendant() [label=="building"] / [label=="building_front"] / [label=="south_7"] / [label=="south_7_front"] / [label=="south_7_front_grid"] / [type=="cell"] [pattern("X{6}((XXAAXX){3}X{12}){3}(X)*(XXAAXX){4}X{6}", "A")] [::groupRegions()] > -> addRoundFacade("windows_front_2_top", ["top_left", "top_right"], 0.2, 5, 0.5)}

{< descendant() [label=="building"] / [label=="building_front"] / [label=="south_4"] / [label=="south_4_front"] / [label=="south_4_front_grid"] > -> updateGrid(3, 3)}

{< descendant() [label=="building"] / [label=="building_front"] / [label=="south_4"] / [label=="south_4_front"] / [label=="south_4_front_grid"] / [type=="cell"] [pattern("XXXXAXXXX", "A")] [::groupRegions()] > -> addRoundFacade("windows_front_1_down", ["top_left", "top_right"], 0.2, 5, 0.5)}

{< descendant() [label=="building"] / [label=="building_front"] / [label=="south_6"] / [label=="south_6_front"] / [label=="south_6_front_grid"] > -> updateGrid(3, 3)}

{< descendant() [label=="building"] / [label=="building_front"] / [label=="south_6"] / [label=="south_6_front"] / [label=="south_6_front_grid"] / [type=="cell"] [pattern("XXXXAXXXX", "A")] [::groupRegions()] > -> addRoundFacade("windows_front_2_down", ["top_left", "top_right"], 0.2, 5, 0.5)}

{< descendant() [label=="building"] / [label=="building_front"] / [label="main_front_grid"] > -> updateGrid(42, 20)}

{< descendant() [label=="building"] / [label=="building_front"] / [label="main_front_grid"] / [type=="cell"] [pattern("(X{20}(XAAX){10}){5}(X{20}(XAAXX{12}XAAX){2}){2}", "A")] [::groupRegions()] > -> addFacade("windows_front_top")}

{< descendant() [label=="building"] / [label=="building_right"] / [label="main_right_grid"] > -> updateGrid(42, 26)}

{< descendant() [label=="building"] / [label=="building_right"] / [label="main_right_grid"] / [type=="cell"] [pattern("x{26}(X{4}(XAAX){2}XX(XAAX){2}X{4})
```

```
    {2}(x{26}((XAAX){3}XX(XAAX){3}){2}){3}(XAAX){3}XX(XAAX){3}", "A")] [::groupRegions
    ()] > -> addFacade("windows_right_top")}
49
50  {< descendant() [label=="building"] / [label=="building_right"] / [label=="east_7"] /
    [label=="east_7_1"] / [label=="east_7_1_grid"] > -> updateGrid(16, 16)}
51
52  {< descendant() [label=="building"] / [label=="building_right"] / [label=="east_7"] /
    [label=="east_7_1"] / [label=="east_7_1_grid"] / [type=="cell"] [pattern("X{16}(X(
    AAX){5}){2}XAAX{4}AAX{4}AAXX{16}(X(AAX){5}){2}X{4}AAX{4}AAX{4}X{16}(X(AAX){5}){2}
    XAAX{4}AAX{4}AAXX{16}(X(AAX){5}){2}", "A")] [::groupRegions()] > -> addFacade("
    windows_right_1")}
53
54  {< descendant() [label=="building"] / [label=="building_right"] / [label=="east_3"] /
    [label=="east_3_1"] / [label=="east_3_1_grid"] > -> updateGrid(10, 16)}
55
56  {< descendant() [label=="building"] / [label=="building_right"] / [label=="east_3"] /
    [label=="east_3_1"] / [label=="east_3_1_grid"] / [type=="cell"] [pattern("X{16}(X(
    AAX){5}){2}X{16}XAAX{4}AAX{4}AAX(X(AAX){5}){3}(XAAX{4}AAX{4}AAX){2}", "A")] [::
    groupRegions()] > -> addFacade("windows_right_1")}
57
58  {< descendant() [label=="building"] / [label=="building_right"] / [label=="east_1"] /
    [label=="east_1_1"] / [label=="east_1_1_grid"] > -> updateGrid(12, 10)}
59
60  {< descendant() [label=="building"] / [label=="building_right"] / [label=="east_1"] /
    [label=="east_1_1"] / [label=="east_1_1_grid"] / [type=="cell"] [pattern("X{10}(
    XXXAAAAXXX){2}", "A")] [::groupRegions()] > -> addRoundFacade("windows_circle_1",
    ["top_left", "top_right", "bottom_right", "bottom_left"], 0.2, 5, 0.5)}
61
62  {< descendant() [label=="building"] / [label=="building_right"] / [label=="east_2"] /
    [label=="east_2_1"] / [label=="east_2_1_grid"] > -> updateGrid(12, 10)}
63
64  {< descendant() [label=="building"] / [label=="building_right"] / [label=="east_2"] /
    [label=="east_2_1"] / [label=="east_2_1_grid"] / [type=="cell"] [pattern("X{10}(
    XXXAAAAXXX){2}", "A")] [::groupRegions()] > -> addRoundFacade("windows_circle_2",
    ["top_left", "top_right", "bottom_right", "bottom_left"], 0.2, 5, 0.5)}
```

APÊNDICE B – IMPLEMENTAÇÃO PARA GERAÇÃO PROCEDURAL DE ELEMENTOS DE FACHADAS

A seguir são descritas as operações que foram alteradas, conforme especificadas no Capítulo 4.2.1, onde as as adições são úteis para a geração dos elementos de fachada.

Código-fonte 16 – Implementação das atualizações na geração de construções

```
def placeMainVirtualShape(side, virtualShape):
    sideCopy = side
    bpy.context.object.rotation_euler[y] = 1.5708

    while sideCopy.getParent() != None:
        if "front" in sideCopy.getLabel():
            bpy.context.object.rotation_euler[x] = 1.5708
            bpy.context.object.location[y] = -side.getParent().getDimY() / 2
            break

        elif "back" in sideCopy.getLabel():
            bpy.context.object.rotation_euler[x] = -1.5708
            bpy.context.object.location[y] = side.getParent().getDimY() / 2
            break

        elif "left" in sideCopy.getLabel():
            bpy.context.object.rotation_euler[z] = 3.14159
            bpy.context.object.location[x] = -side.getParent().getDimX() / 2
            break

        elif "right" in sideCopy.getLabel():
            bpy.context.object.location[x] = side.getParent().getDimX() / 2
            break

        sideCopy = sideCopy.getParent()

def addVolume(label, parent, extrusionSize, sidesLabels, gridLabel, rows, columns):
    gridRows, gridColumns = len(rows), len(columns)
    subgrid = createGridAux(label, parent, gridLabel, sidesLabels[0])
    grandparent = parent.getParent()

    bpy.ops.object.mode_set(mode = 'OBJECT')
    bpy.ops.object.mode_set(mode = 'EDIT')

    regionIndex = len(grandparent.obj.data.polygons)-1

    bpy.ops.mesh.select_all(action = 'DESELECT')
    bpy.ops.object.mode_set(mode = 'OBJECT')

```

```
40      bpy.data.objects[grandparent.getLabel()].data.polygons[regionIndex].select = True
41
42      bpy.ops.object.mode_set(mode = 'EDIT')
43      bpy.ops.mesh.select_mode(type = 'FACE')
44
45      if "front" in parent.getLabel():
46          bpy.ops.mesh.extrude_region_move(TRANSFORM_OT_translate={"value":(0, -
        extrusionSize, 0)})
47          x, y, z = (0, -extrusionSize, 0)
48
49      elif "left" in parent.getLabel():
50          bpy.ops.mesh.extrude_region_move(TRANSFORM_OT_translate={"value":(-
        extrusionSize, 0, 0)})
51          x, y, z = (-extrusionSize, 0, 0)
52
53      elif "right" in parent.getLabel():
54          bpy.ops.mesh.extrude_region_move(TRANSFORM_OT_translate={"value":(
        extrusionSize, 0, 0)})
55          x, y, z = (extrusionSize, 0, 0)
56
57      elif "back" in parent.getLabel():
58          bpy.ops.mesh.extrude_region_move(TRANSFORM_OT_translate={"value":(0,
        extrusionSize, 0)})
59          x, y, z = (0, extrusionSize, 0)
60
61      bpy.ops.mesh.select_all(action = 'DESELECT')
62      bpy.ops.object.mode_set(mode = 'OBJECT')
63
64      regionFrontMesh = bpy.data.objects[grandparent.getLabel()].data.polygons[
        regionIndex+1]
65      regionLeftMesh = bpy.data.objects[grandparent.getLabel()].data.polygons[
        regionIndex+3]
66      regionRightMesh = bpy.data.objects[grandparent.getLabel()].data.polygons[
        regionIndex+5]
67
68      element = parent.descendant(label)
69
70      element.addChild(Construction(sidesLabels[0], regionFrontMesh, element))
71      element.addChild(Construction(sidesLabels[1], regionLeftMesh, element))
72      element.addChild(Construction(sidesLabels[2], regionRightMesh, element))
73
74      regionFront = element.descendant(sidesLabels[0])
75
76      regionFront.addChild(Virtual(subgrid.name, subgrid, regionFront, gridRows,
        gridColumns+1))
77
78      bpy.ops.object.mode_set(mode = 'OBJECT')
79      subgrid.location.x += x
80      subgrid.location.y += y
```

```
81      subgrid.location.z += z
82
83      return regionIndex + 1
84
85  def loadAddVolume(data):
86      selection, action = data.split("->")
87
88      selectionLabels = selection.split('"')[1::2]
89      selectionLabels.pop()
90      rowsIndices, columnsIndices = re.findall('\(([^)]+)', selection)
91      rows, columns = setIndices(rowsIndices, columnsIndices)
92      actionContent = re.findall('\(([^)]+)', action)[0]
93      label, parentLabel, extrusionSize, frontLabel, leftLabel, rightLabel =
        actionContent.split(", ")
94      label = label.replace('"', "")
95      parentLabel = parentLabel.replace('"', "")
96
97      sideLabels = []
98
99      sideLabels.append(frontLabel.replace('"', "").replace("[", ""))
100     sideLabels.append(leftLabel.replace('"', ""))
101     sideLabels.append(rightLabel.replace('"', "").replace("]", ""))
102
103     grid = selectNode(root, selectionLabels)
104     gridCopy = duplicateShape(grid.getLabel())
105     gridCopyNode = Virtual(gridCopy.name, gridCopy, grid.getParent(), grid.getRows()
        +1, grid.getColumns()+1)
106     selectToBeVolume(gridCopyNode, gridCopyNode.getRows(), gridCopyNode.getColumns(),
        rows, columns)
107     parent = gridCopyNode.getParent()
108     gridLabel = gridCopyNode.getLabel()
109
110     regionIndex = addVolume(label, parent, float(extrusionSize), sideLabels, gridLabel
        , rows, columns)
111
112     return regionIndex
113
114 def computeInstructions(rules):
115     regionIndex = 0
116
117     for index, rule in enumerate(rules):
118         if index == 0:
119             loadSettings(rule)
120
121         if "createShape" in rule:
122             print("# createShape: Ran after loadSettings()")
123
124         if "createGrid" in rule:
125             loadCreateGrid(rule)
```

```
126
127         if "addVolume" in rule:
128             regionIndex = loadAddVolume(rule)
129
130         if "updateGrid" in rule:
131             loadUpdateGrid(rule)
132
133         if "roundShape" in rule:
134             loadRoundShape(rule, regionIndex)
135
136         if "addRoundFacadeWithFrame" in rule:
137             loadAddRoundFacadeWithFrame(rule)
138             continue
139
140         if "addFacadeWithFrame" in rule:
141             loadAddFacadeWithFrame(rule)
142             continue
143
144         if "addRoundFacade" in rule:
145             loadAddRoundFacade(rule)
146             continue
147
148         if "addFacade" in rule:
149             loadAddFacade(rule)
150             continue
151
152         print()
```

No Código-Fonte 17, são apresentadas as operações adicionadas pelo presente trabalho, conforme definido na Capítulo 4.2.2.

Código-fonte 17 – Implementação que permite adicionar elementos de fachadas

```
1  def rangeIndex(idxBegin, idxEnd):
2      return [i for i in range(idxBegin, idxEnd)]
3
4  def regionSelected():
5      polygons = bpy.context.active_object.data.polygons
6      for polygon in polygons:
7          if(polygon.select == True):
8              return polygon.index
9
10 def createGridAux(label, parent, gridLabel, front_label):
11     bpy.ops.mesh.separate(type='SELECTED')
12
13     bpy.data.objects[gridLabel + ".001"].select_set(False)
14     grandparent = parent.getParent()
15     bpy.ops.object.mode_set(mode = 'OBJECT')
```

```
16      bpy.ops.object.select_all(action='DESELECT')
17
18      bpy.context.view_layer.objects.active = bpy.data.objects[gridLabel]
19      bpy.data.objects[gridLabel].select_set(True)
20      bpy.ops.object.delete()
21
22      region = bpy.data.objects[gridLabel + ".001"]
23      region.name = label
24
25      bpy.data.objects[label].select_set(True)
26      bpy.context.view_layer.objects.active = bpy.data.objects[label]
27
28      bpy.ops.object.mode_set(mode = 'OBJECT')
29      parent.addChild(Construction(label, region, parent))
30      bpy.ops.object.origin_set(type='ORIGIN_GEOMETRY', center='BOUNDS')
31      bpy.ops.object.duplicate()
32
33      regionGrid = bpy.data.objects[label + ".001"]
34      regionGrid.name = front_label + "_grid"
35      region = parent.descendant(label)
36
37      bpy.data.objects[region.getLabel()].select_set(False)
38      bpy.data.objects[regionGrid.name].select_set(False)
39      bpy.context.view_layer.objects.active = bpy.data.objects[label]
40      bpy.data.objects[label].select_set(True)
41
42      bpy.ops.object.mode_set(mode = 'OBJECT')
43      bpy.ops.object.mode_set(mode = 'EDIT')
44      bpy.ops.mesh.select_all(action = 'SELECT')
45      bpy.ops.object.mode_set(mode = 'OBJECT')
46      bpy.ops.object.mode_set(mode = 'EDIT')
47
48      bpy.ops.mesh.region_to_loop()
49
50      bpy.ops.object.mode_set(mode = 'OBJECT')
51
52      ids = list(range(len(region.getEdges())))
53
54      selectedEdges = [e.index for e in region.getEdges() if e.select]
55
56      for e in selectedEdges:
57          ids.remove(e)
58
59      bpy.ops.object.mode_set(mode = 'EDIT')
60      bpy.ops.mesh.select_all(action = 'DESELECT')
61      bpy.ops.object.mode_set(mode = 'OBJECT')
62
63      for id in ids:
64          bpy.data.objects[label].data.edges[id].select = True
```

```
65
66      bpy.ops.object.mode_set(mode = 'EDIT')
67      bpy.ops.mesh.dissolve_edges()
68      bpy.ops.object.mode_set(mode = 'OBJECT')
69      bpy.data.objects[label].select_set(True)
70
71      bpy.context.view_layer.objects.active = bpy.data.objects[grandparent.getLabel()]
72      bpy.data.objects[grandparent.getLabel()].select_set(True)
73
74      bpy.ops.object.join()
75
76      deselectIndex = bpy.data.objects[grandparent.getLabel()].pass_index
77
78      bpy.data.objects[grandparent.getLabel()].select_set(False)
79      bpy.data.objects[deselectIndex].select_set(False)
80
81      return regionGrid
82
83  def translateObjectBySide(objName, value):
84      bpy.ops.object.mode_set(mode = 'OBJECT')
85      if 'front' in objName:
86          bpy.ops.transform.translate(value=(0, value, 0))
87      elif 'right' in objName:
88          bpy.ops.transform.translate(value=(value, 0, 0))
89      elif 'left' in objName:
90          bpy.ops.transform.translate(value=(-value, 0, 0))
91      elif 'back' in objName:
92          bpy.ops.transform.translate(value=(0, -value, 0))
93
94  def groupFacadeWithConstruction(principalObjectName, objectName, edgeExtrusion='
        noExtrusion', thickness=0.09, depth=0.05, direction=""):
95      obj = bpy.data.objects[objectName]
96      principal = bpy.data.objects[principalObjectName]
97      selectSingleObject(objectName)
98
99      acc = 0.5
100     translateObjectBySide(direction, acc)
101     translateSize = 0.005
102
103     while True:
104         isInside = meshInsideMesh(obj, principal)
105
106         if not isInside:
107             translateObjectBySide(direction, translateSize)
108             acc += translateSize
109         else:
110             break
111
112         if acc >= 2: # tamanho da translacao maxima
```

```
113             break
114
115     translateObjectBySide(direction, -acc)
116
117     bpy.ops.object.mode_set(mode = 'EDIT')
118     bpy.ops.mesh.select_mode(type="FACE")
119     bpy.ops.mesh.select_all(action='SELECT')
120
121     bpy.ops.mesh.solidify(thickness=acc)
122
123     bpy.ops.object.mode_set(mode = 'OBJECT')
124
125     if 'front' in direction:
126         bpy.ops.transform.translate(value=(0, -0.025, 0))
127
128     print(objectName + '--' + direction)
129
130     addBooleanModifierInFacade(objectName, obj, principal)
131
132     if 'front' in direction:
133         bpy.ops.transform.translate(value=(0, 0.025, 0))
134
135     if edgeExtrusion in ['all', 'frame', 'none']:
136         addExtrusionInFrame(objectName, edgeExtrusion, thickness, depth)
137
138     bpy.ops.object.mode_set(mode = 'OBJECT')
139     selectObject(principalObjectName)
140     bpy.ops.object.join()
141
142 def addBooleanModifierInFacade(objectName, objectOrigin, objectTarget):
143     selectSingleObject(objectName)
144     nameOperation = "op_diff_" + objectName
145     bool_one = objectOrigin.modifiers.new(type="BOOLEAN", name=nameOperation)
146     bool_one.object = objectTarget
147     bpy.context.object.modifiers[nameOperation].operation = 'DIFFERENCE'
148
149     if 'left' in objectName:
150         bpy.context.object.modifiers[nameOperation].use_hole_tolerant = True
151
152     if meshInsideMesh(objectOrigin, objectTarget) is False:
153         bpy.context.object.modifiers[nameOperation].use_self = True
154
155     bpy.ops.object.modifier_apply(modifier=nameOperation)
156
157 def addExtrusionInFrame(objectName, edgeExtrusion, thickness, depth):
158     bpy.ops.object.mode_set(mode = 'EDIT')
159     bpy.ops.mesh.select_all(action='DESELECT')
160     bpy.ops.object.mode_set(mode = 'OBJECT')
161     bpy.data.objects[objectName].data.polygons[0].select = True
```

```
162     bpy.ops.object.mode_set(mode = 'EDIT')
163 
164     if edgeExtrusion in 'all':
165         bpy.ops.mesh.inset(thickness=thickness, depth=depth, release_confirm=True,
    use_even_offset=True)
166     elif edgeExtrusion in 'frame':
167         bpy.ops.mesh.inset(thickness=thickness, depth=0, use_select_inset=True,
    release_confirm=True, use_even_offset=True)
168         bpy.ops.mesh.extrude_context_move(TRANSFORM_OT_translate={"value":(0, 0, depth
    ), "orient_axis_ortho":'X', "orient_type":'NORMAL'})
169     else:
170         bpy.ops.mesh.inset(thickness=thickness, depth=0, release_confirm=True,
    use_even_offset=True)
171 
172     bpy.ops.mesh.select_all(action='DESELECT')
173 
174 def meshInsideMesh(source, target):
175     for p in source.bound_box:
176         location = getVertexLocationInWorldSpace(source, Vector((p[0], p[1],p[2])))
177 
178         if not pointInsideMesh(location, target):
179             return False
180 
181     return True
182 
183 def pointInsideMesh(point, ob):
184     axes = [ Vector((1,0,0)), Vector((0,1,0)), Vector((0,0,1)) ]
185     outside = False
186     for axis in axes:
187         mat = Matrix(ob.matrix_world)
188         mat.invert()
189         orig = mat@point
190         count = 0
191         while True:
192             _,location,_,index = ob.ray_cast(orig,orig+axis*10000.0)
193             if index == -1: break
194             count += 1
195             orig = location + axis*0.00001
196 
197         if count%2 == 0:
198             outside = True
199             break
200     return not outside
201 
202 def getVertexLocationInWorldSpace(obj, vertex):
203     return obj.matrix_world.copy() @ vertex
204 
205 def hideAll():
206     for obj in bpy.data.objects:
```

```
207         obj.hide_set(True)
208
209 def deselectAllObjects():
210     for obj in bpy.context.selected_objects:
211         obj.select_set(False)
212
213 def selectObject(objectName):
214     bpy.data.objects[objectName].select_set(True)
215     bpy.context.view_layer.objects.active = bpy.data.objects[objectName]
216
217 def selectSingleObject(objectName):
218     deselectAllObjects()
219     selectObject(objectName)
220
221 def getIndex(total_cols, rowIndex, columnIndex):
222     return (total_cols * rowIndex) - total_cols + (columnIndex - 1)
223
224 def applyCornerRounding(corners, degree, segments, profile, object, indexes):
225     selectSingleObject(object.name)
226     bpy.ops.object.mode_set(mode = 'OBJECT')
227
228     if(corners[0]):
229         object.data.vertices[indexes[0]].select = True # top-left
230
231     if(corners[1]):
232         object.data.vertices[indexes[1]].select = True # top-right
233
234     if(corners[2]):
235         object.data.vertices[indexes[2]].select = True # bottom-right
236
237     if(corners[3]):
238         object.data.vertices[indexes[3]].select = True # bottom-left
239
240     selectSingleObject(object.name)
241
242     bpy.ops.object.mode_set(mode = 'EDIT')
243     bpy.ops.mesh.select_mode(type="VERT")
244     bpy.ops.mesh.bevel(offset=degree, offset_pct=0, profile=profile, segments=segments
    , affect="VERTICES", clamp_overlap=True)
245
246 def setIndices(rowsIndices, columnsIndices):
247     if "indexRange" in rowsIndices:
248         r = list(map(int, re.findall(r'\d+', rowsIndices)))
249
250         rows = rangeIndex(r[0], r[-1]+1)
251     else:
252         rows = [int(i) for i in re.findall(r'\d+', rowsIndices)]
253
254     if "indexRange" in columnsIndices:
```

```
255         c = list(map(int, re.findall(r'\d+', columnsIndices)))
256 
257         columns = rangeIndex(c[0], c[-1]+1)
258     else:
259         columns = [int(i) for i in re.findall(r'\d+', columnsIndices)]
260 
261     return rows, columns
262 
263 def addRoundnessToFacade(corners, degree, segments, profile, facade):
264     verts = sorted([(v.co.x, v.co.y, v.co.z, v.index) for v in facade.data.vertices],
    reverse=False)
265     indexes = [vert[3] for vert in verts]
266     applyCornerRounding(corners, degree, segments, profile, facade, indexes)
267 
268 def selectCells(sideLabels, pattern, rows, cols):
269     deselectAllObjects()
270 
271     grid = selectNode(
272         root=root,
273         labels=sideLabels)
274 
275     bpy.ops.object.mode_set(mode = 'EDIT')
276     bpy.ops.mesh.select_all(action = 'DESELECT')
277 
278     total_cols = grid.getColumns()
279     gridCopy = duplicateShape(grid.getLabel())
280 
281     if len(rows) == 0 and len(cols) == 0:
282         rows = [row+1 for row in range(grid.getRows())]
283         cols = [col+1 for col in range(grid.getColumns())]
284 
285     if len(pattern) == 0:
286         for rowIndex in rows:
287             for columnIndex in cols:
288                 cellIndex = getIndex(total_cols, rowIndex, columnIndex)
289                 bpy.ops.object.mode_set(mode = 'OBJECT')
290                 bpy.data.objects[gridCopy.name].data.polygons[cellIndex].select = True
291                 bpy.ops.object.mode_set(mode = 'EDIT')
292     else:
293         patternGrid(pattern[0], pattern[1], gridCopy, rows, cols)
294 
295     parent = grid.getParent()
296 
297     return parent, gridCopy
298 
299 def createFacadeObjs(grid, groupingType="none"):
300     bpy.ops.object.mode_set(mode = 'OBJECT')
301     bpy.ops.object.mode_set(mode = 'EDIT')
302     bpy.ops.mesh.select_mode(type="FACE")
```

```
303     bpy.ops.mesh.select_all(action='INVERT')
304     bpy.ops.mesh.delete(type='FACE')
305     bpy.ops.mesh.select_mode(type="VERT")
306     bpy.ops.mesh.select_all(action='SELECT')
307 
308     if groupingType not in "none":
309         bpy.ops.mesh.dissolve_limited()
310 
311     bpy.ops.mesh.select_all(action='DESELECT')
312     bpy.ops.mesh.separate(type='LOOSE')
313 
314     objects = bpy.data.objects
315     facades = [obj for obj in objects if grid.name in obj.name]
316 
317     return facades
318 
319 def updateGrid(label, selectionLabels, rows, columns):
320     parent = selectNode(root, selectionLabels)
321     old_grid = parent.descendant(label)
322     objectWillRemove = bpy.data.objects[label]
323 
324     old_x, old_y, old_z = objectWillRemove.location
325     parent.dimX = objectWillRemove.dimensions.y
326     parent.dimZ = objectWillRemove.dimensions.x
327 
328     bpy.data.objects.remove(objectWillRemove)
329     parent.children.remove(old_grid)
330 
331     deselectAllObjects()
332 
333     mesh = createGrid(label, parent, rows, columns)
334     placeMainVirtualShape(parent, mesh)
335 
336     bpy.ops.object.mode_set(mode = "OBJECT")
337 
338     if bpy.data.objects[label].location == Vector((0,0,0)):
339         bpy.ops.object.mode_set(mode = "EDIT")
340         bpy.ops.transform.translate(value=(old_x, old_y, old_z), orient_axis_ortho='X'
    )
341         bpy.ops.object.mode_set(mode = "OBJECT")
342 
343 def addFacade(label, sideLabels, rows, cols, pattern, groupingType="none"):
344     parent, grid = selectCells(sideLabels, pattern, rows, cols)
345 
346     facades = createFacadeObjs(grid, groupingType)
347 
348     direction = sideLabels[1].split('_')[1]
349 
350     for i, facade in enumerate(facades):
```

```
351         parent.addChild(Construction(f"{label}_{i}", facade, parent))
352         groupFacadeWithConstruction(sideLabels[0], facade.name, direction=direction)
353 
354 def addRoundedFacade(label, sideLabels, rows, cols, corners, degree, segments, profile
    , pattern, groupingType="none"):
355     parent, grid = selectCells(sideLabels, pattern, rows, cols)
356     facades = createFacadeObjs(grid, groupingType)
357 
358     direction = sideLabels[1].split('_')[1]
359 
360     for i, facade in enumerate(facades):
361         addRoundnessToFacade(corners, degree, segments, profile, facade)
362         parent.addChild(Construction(f"{label}_{i}", facade, parent))
363         groupFacadeWithConstruction(sideLabels[0], facade.name, direction=direction)
364 
365 def addFacadeWithFrame(label, sideLabels, rows, cols, pattern, edgeExtrusion,
    thickness, width, groupingType="none"):
366     parent, grid = selectCells(sideLabels, pattern, rows, cols)
367     facades = createFacadeObjs(grid, groupingType)
368 
369     direction = sideLabels[1].split('_')[1]
370 
371     for i, facade in enumerate(facades):
372         parent.addChild(Construction(f"{label}_{i}", facade, parent))
373         groupFacadeWithConstruction(sideLabels[0], facade.name, edgeExtrusion,
    thickness, width, direction=direction)
374 
375 def addRoundFacadeWithFrame(label, sideLabels, rows, cols, corners, degree, segments,
    profile, pattern, edgeExtrusion, thickness, width, groupingType="none"):
376     parent, grid = selectCells(sideLabels, pattern, rows, cols)
377     facades = createFacadeObjs(grid, groupingType)
378 
379     direction = sideLabels[1].split('_')[1]
380 
381     for i, facade in enumerate(facades):
382         parent.addChild(Construction(f"{label}_{i}", facade, parent))
383         addRoundnessToFacade(corners, degree, segments, profile, facade)
384         groupFacadeWithConstruction(sideLabels[0], facade.name, edgeExtrusion,
    thickness, width, direction=direction)
385 
386 def loadUpdateGrid(data):
387     selection, action = data.split("->")
388 
389     selectionLabels = selection.split('"')[1::2]
390     gridLabel = selectionLabels.pop()
391 
392     rows, columns = action[action.find("(")+1:action.find(")")].split(", ")
393 
394     updateGrid(gridLabel, selectionLabels, int(rows), int(columns))
```

```
395
396
397 def patternGrid(regex, pat, grid, cols, rows):
398     result = generateString(regex, len(cols)*len(rows))
399
400     list_result = list(result)
401     bpy.ops.object.mode_set(mode = 'EDIT')
402     bpy.ops.mesh.select_mode(type = 'FACE')
403
404     for i in range(len(list_result)):
405         if(list_result[i] == pat):
406             bpy.ops.object.mode_set(mode = 'OBJECT')
407             bpy.data.objects[grid.name].data.polygons[i].select = True
408             bpy.ops.object.mode_set(mode = 'EDIT')
409
410 def processCurlyBrackets(tokens, index, result, value):
411     if tokens[index] == '{':
412         s_number = ''
413         index += 1
414
415         while tokens[index] != '}':
416             s_number += tokens[index]
417             index += 1
418
419         number = int(s_number) - 1
420
421         while number > 0:
422             result += value
423             number -= 1
424
425     return index, result
426
427 def processAsterisk(tokens, index, result, start, end, max):
428     if tokens[index] == '*':
429         substring = result[start:end:1]
430         list_substring = list(substring)
431
432         tam = max - end
433         j = 0
434         while tam > 0:
435             if(j == len(list_substring)):
436                 j = 0
437
438             result += list_substring[j]
439
440             j += 1
441             tam -= 1
442
443     return index, result
```

```
444
445 def generateString(regex, max_value):
446     tokens = list(regex)
447
448     if regex.count('*') > 1:
449         raise Exception('Regex not valid')
450
451     result = ''
452     group = []
453     # (Operation asterisk?, start, str_before, str, position asterisk)
454     asterisk_op = (False, 0, '', '', -1)
455
456     i = 0
457
458     while i < len(tokens):
459
460         if tokens[i].isalpha():
461             result += tokens[i]
462
463         i, result = processCurlyBrackets(tokens, i, result, tokens[i-1])
464
465         if tokens[i] == '(':
466             group.append(len(result))
467
468         if tokens[i] == ')':
469             i += 1
470             start = group.pop()
471             end = len(result)
472
473             if i >= len(tokens):
474                 break
475
476             if tokens[i] == '{':
477                 i, result =
478                     processCurlyBrackets(tokens, i, result, result[start:end:1])
479
480             if tokens[i] == '*':
481                 if i == len(tokens)-1:
482                     i, result =
483                         processAsterisk(tokens, i, result, start, end, max_value)
484                 else:
485                     asterisk_op =
486                         (True, start, result[0:start:1], result[start:end:1], i)
487
488                     result = ''
489         i += 1
490
491     if asterisk_op[0]:
492         tam = max_value - len(result)
```

```
493         end = len(asterisk_op[3]) + len(asterisk_op[2])
494         _, res = processAsterisk(tokens, asterisk_op[4], asterisk_op[3], 0, end, tam)
495         result = asterisk_op[2] + res + result
496 
497     return result
498 
499 
500 def getActions(action):
501     actionContent = re.findall('\(([^)]+)', action)[0]
502 
503     actionParameters = actionContent.split(", ")
504     return actionParameters
505 
506 def getColsAndRows(selection):
507     rowsIndices = re.findall('rowIdx in ([^)]+)', selection)
508     columnsIndices = re.findall('colIdx in ([^)]+)', selection)
509 
510     if len(rowsIndices) == 0 and len(columnsIndices) == 0:
511         rows, columns = [], []
512     else:
513         rows, columns = setIndices(rowsIndices[0], columnsIndices[0])
514     return rows,columns
515 
516 def getPattern(selection):
517     pattern = re.findall('pattern\(([^]]+)', selection)
518 
519     if len(pattern) > 0:
520         pattern = pattern[0][:-1].replace('"', "").replace(" ", "").split(",")
521     else:
522         pattern = []
523     return pattern
524 
525 def getSelectionLabels(selection):
526     selectionLabels = selection.split('"')[1::2]
527 
528     index_remove = selectionLabels.index("cell")
529     del selectionLabels[index_remove:]
530     return selectionLabels
531 
532 def getGroupingType(selection):
533     groupingType = re.search(r"\[::(.)+?\]", selection)
534 
535     if groupingType:
536         groupingType = groupingType[0].replace("[::", "").replace("]", "")
537     else:
538         groupingType = "none"
539 
540     return groupingType
541 
```

```
542 def loadAddFacade(data):
543     selection, action = data.split("->")
544 
545     groupingType = getGroupingType(selection)
546     selectionLabels = getSelectionLabels(selection)
547     pattern = getPattern(selection)
548     rows, columns = getColsAndRows(selection)
549     actionParameters = getActions(action)
550     label = actionParameters[0].replace('"', "").replace(" ", "")
551 
552     addFacade(label, selectionLabels, rows, columns, pattern, groupingType)
553 
554 def loadAddFacadeWithFrame(data):
555     selection, action = data.split("->")
556 
557     groupingType = getGroupingType(selection)
558     selectionLabels = getSelectionLabels(selection)
559     pattern = getPattern(selection)
560     rows, columns = getColsAndRows(selection)
561     actionParameters = getActions(action)
562 
563     label = actionParameters[0].replace('"', "").replace(" ", "")
564     edgeExtrusion = actionParameters[1].replace('"', "").replace(" ", "")
565     thickness = float(actionParameters[2])
566     width = float(actionParameters[3])
567 
568     addFacadeWithFrame(label, selectionLabels, rows, columns, pattern,
569         edgeExtrusion, thickness, width, groupingType)
570 
571 def loadAddRoundFacade(data):
572     selection, action = data.split("->")
573 
574     groupingType = getGroupingType(selection)
575     selectionLabels = getSelectionLabels(selection)
576     pattern = getPattern(selection)
577     rows, columns = getColsAndRows(selection)
578     actionParameters = getActions(action)
579 
580     label = actionParameters[0].replace('"', "").replace(" ", "")
581     roundingDegree = float(actionParameters[-3])
582     segments = int(actionParameters[-2])
583     profile = float(actionParameters[-1])
584 
585     cornersSelect = []
586 
587     for i in range(1, len(actionParameters) - 3):
588         actionParameters[i] = actionParameters[i].replace('"', "").replace("[", "").
    replace("]", "")
589         cornersSelect.append(actionParameters[i])
```

```
590
591     corners = [
592         "top_left" in cornersSelect,
593         "top_right" in cornersSelect,
594         "bottom_right" in cornersSelect,
595         "bottom_left" in cornersSelect
596     ]
597
598     addRoundedFacade(label, selectionLabels, rows, columns, corners, roundingDegree,
    segments, profile, pattern, groupingType)
599
600 def loadAddRoundFacadeWithFrame(data):
601     selection, action = data.split("->")
602
603     groupingType = getGroupingType(selection)
604     selectionLabels = getSelectionLabels(selection)
605     pattern = getPattern(selection)
606     rows, columns = getColsAndRows(selection)
607     actionParameters = getActions(action)
608
609     label = actionParameters[0].replace('"', "").replace(" ", "")
610     roundingDegree = float(actionParameters[-6])
611     segments = int(actionParameters[-5])
612     profile = float(actionParameters[-4])
613     edgeExtrusion = actionParameters[-3].replace('"', "").replace(" ", "")
614     thickness = float(actionParameters[-2])
615     width = float(actionParameters[-1])
616
617     if edgeExtrusion not in ['all', 'frame', 'none']:
618         edgeExtrusion = 'noExtrusion'
619
620     cornersSelect = []
621
622     for i in range(1, len(actionParameters) - 6):
623         actionParameters[i] = actionParameters[i].replace('"', "").replace("[", "").
    replace("]", "")
624         cornersSelect.append(actionParameters[i])
625
626     corners = [
627         "top_left" in cornersSelect,
628         "top_right" in cornersSelect,
629         "bottom_right" in cornersSelect,
630         "bottom_left" in cornersSelect
631     ]
632
633     addRoundFacadeWithFrame(label, selectionLabels, rows, columns, corners,
    roundingDegree, segments, profile, pattern, edgeExtrusion, thickness, width,
    groupingType)
```