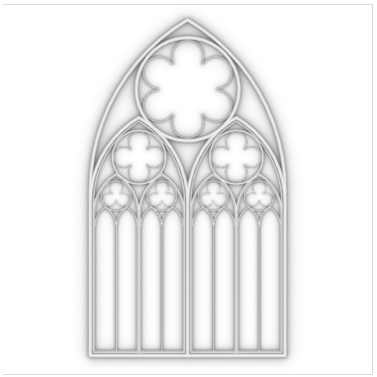
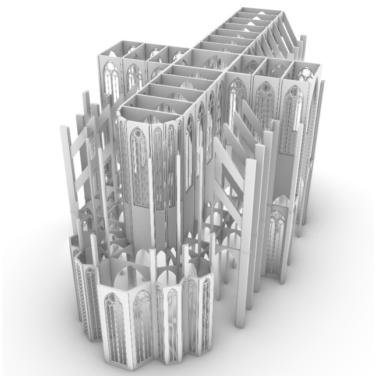
TÉCNICO LISBOA





Procedural Modeling of Architectonic Shapes:

An Approach Focused on Structural and Ornamental Variations

Lourenço Isidro do Carmo Neiva Mourão

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. Dr. António Paulo Teles de Menezes Correia Leitão

Examination Committee

Chairperson: Prof. Dr. Miguel Ângelo Marques de Matos
Supervisor: Prof. Dr. António Paulo Teles de Menezes Correia Leitão
Member of the Committee: Prof. Dr. Daniel Simões Lopes

November 2024

Declaration
I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

Lembro-me bem do primeiro contacto com o meu orientador, Professor António Menezes Leitão. Foi através do seu espaço pedagógico, nas primeiras aulas de Programação Avançada, que me apercebi das suas competências técnicas e qualidade relacional de excepção—bastou uma aula.

Recebeu-me de braços abertos e com o seu entusiasmo característico na forma como expõe e envolve os alunos, como quem estende uma mão amiga, dando liberdade de escolha e reforçando sempre a importância do propósito e valor que cada um encontra naquilo que faz. Foi assim que escolhemos o tema que deu origem a este estudo.

Foi, entre conversas marcadas pelo apreço à Cultura e ao Conhecimento, e numa pausa sentados na escadaria do Técnico, que surgiu a inspiração para a minha visita à Carnegie Mellon University (CMU). Acompanhou-me e ajudou-me a crescer, dando-me sempre valiosos conselhos e partilhando experiências de vida que em muito me serviram e servirão.

A vontade de potenciar a autonomia dos seus alunos, aliada à sensibilidade, cuidado, e atenção para com o próximo, foram e continuam a ser os pilares mais importantes que enriquecem e dão sentido ao que, para mim, define o contexto Académico. Ajudou-me, muitas vezes sem se aperceber disso, e os seus apertos de mão vigorosos muita força me trouxeram em dias de especial dificuldade.

Para além das linguagens de programação que tanto o fascinam, há uma outra que domina—essa, não há computador que replique. Partilho, assim, uma frase que me disse durante a primeira semana de adaptação em Pittsburgh e que muito me marcou; pois, tal como a Ciência avança à base da partilha, também nós crescemos na troca de afetos e palavras: "*Um Ser Humano é um Ser Humano em qualquer parte do mundo.*"

Agradeço-lhe por isto e por tudo o que ficou por dizer. É um Professor excepcional e, acima de tudo, um Amigo para a vida. Conto consigo para as etapas vindouras.

Quero, também, agradecer a paciência e atenção de todos os que fazem parte deste grupo de investigação, em especial à Inês Caetano, Inês Pereira, e Renata Castelo Branco pelos valiosos conselhos, revisões, e toda a ajuda dedicada a este trabalho.

Agradeço, igualmente, a todos os meus Amigos, em particular ao André, David, Filipe, Francisco, Gonçalo, Manuel, Pedro, Serhii, à Catarina e à Maria, pela vossa companhia, amizade, compan-

heirismo, e sensibilidade.

Agradeço à Fundação para a Ciência e Tecnologia (FCT), através do Programa CMU Portugal, pela oportunidade de realizar uma visita de três meses, aos Estados Unidos da América, na CMU (em Pittsburgh, Pennsylvania), que foi fundamental para reforçar a minha vontade de seguir para Doutoramento. Agradeço, também, à CMU, que me proporcionou um ambiente interdisciplinar enriquecedor, e aos membros do departamento que me acolheram, criando um ambiente de trabalho inspirador e permitindo-me fazer valiosas amizades que contribuíram de forma significativa para o meu percurso Académico.

Agradeço à minha Família—a todos.

Agradeço, por fim, a uma Mulher que muito admiro e que gostava de ver atingir os seus objetivos académicos e pessoais—para que alcance a liberdade que tanto merece. A essa mesma Mulher, aguardo com entusiasmo o nosso reencontro e os passeios pelos jardins de Viena de Áustria que tanto nos marcaram.

São estes os fatores que nos constituem e que elevam o propósito das nossas vidas para patamares inabaláveis. Só assim se erguem os pilares que nos sustêm e que, em conjunto, se constitui a Catedral repleta de vitrais que assinalam a nossa História e que iluminam os extensos corredores que percorremos e construímos ao longo da nossa vida.

Abstract

Limitations regarding the ornamental aspects of a shape are predominantly related to the time it takes a designer to perform shape refinement processes, mainly due to the complex interdependencies between structural and ornamental representations. An extensive body of research on Computer-Aided Drafting (CAD)—and corresponding sub-domains—has been conducted to facilitate complex modeling processes, ranging from rule-based to scripting-based approaches, with the latter being much more flexible and extensive than the former. One possible case study that stresses the inclusion of complex ornamentation in structural assets is that of Gothic Architecture. Albeit expressive, these previous studies conform to outdated, rigid, and convoluted scripting-based approaches that contribute to a lesser degree of intelligibility in a designer's workflow. Consequently, this study aims to refer to Gothic examples (windows and cathedrals) as a basis for developing an approach—through a friendlier scripting-based Algorithmic Design (AD) tool—that isolates the definition of structural and ornamental assets from one another, such that the former incorporates the latter with no additional changes to the structural model. This allows for the derivation of different designs and reduces the time it takes for a designer to apply these modeling processes. Finally, to evaluate our solution, this study contributes with a generative solution capable of dynamically deriving multiple cathedral floor plans incorporating various ornamental elements, suggesting a potential application to different case studies.

Keywords

Algorithmic Design, Computer-Aided Drafting, Procedural Modeling, Ornamentation

Resumo

As limitações no que às características ornamentais de formas geométricas dizem respeito, devem-se predominantemente ao tempo de execução de processos de sofisticação ornamental por parte de designers—estando estes relacionados com as interdependências, de natureza complexa, entre representações estruturais e ornamentais. Com o intuito de simplificar processos de modelação complexos, realizaram-se estudos no domínio de Desenho Assistido por Computador (DAC)—e subdomínios correspondentes—, desde abordagens à base de derivação de regras a abordagens de programação, sendo esta última particularmente mais flexível e extensível que a anterior. Um estudo de caso que enfatiza a inclusão de ornamentações complexas em representações estruturais é o da Arquitectura Gótica. Embora expressivos, os estudos previamente mencionados recorrem a abordagens de programação rígidas, desactualizadas, e complexas; contribuindo, assim, para ferramentas de design menos compreensíveis. Consequentemente, este estudo pretende recorrer a exemplos de Arquitectura Gótica (janelas e catedrais) como base para desenvolver uma abordagem—através de um ferramenta de Design Algorítmico (DA) mais favorável—capaz de isolar definições estruturais e ornamentais, de forma a que esta última seja incorporada pela anterior sem que sejam necessárias mudanças adicionais aos modelos estruturais de base; permitindo, assim, a geração de designs variados e reduzindo o tempo que leva a aplicar estes processos de modelação. Por último, no sentido de avaliar a nossa solução, este estudo contribui com uma solução generativa capaz de projectar diversas plantas arquitectónicas de catedrais que incorporam elementos ornamentais variados; sugerindo, assim, uma potencial aplicação a diferentes estudos de caso para além do Gótico.

Palavras Chave

Design Algorítmico, Desenho Assistido por Computador, Modelação Procedimental, Ornamentação

Contents

List of Figures

Listings

Acronyms

2D	Bi-Dimensional
3D	Three-Dimensional
AD	Algorithmic Design
AI	Artificial Intelligence
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CAD	Computer-Aided Drafting
CB-Rep	Combined Boundary Representation
CD	Computational Design
CG	Computer Graphics
CGA	Computer Generated Architecture
DP	Design Procedure
DSL	Domain-Specific Language
GML	Generative Modeling Language
GUI	Graphical User Interface
ML	Machine Learning
PC	Parametric Combination
PD	Parametric Design
PH	Parametric Hybridization
PM	Procedural Modeling
PV	Parametric Variation
SG	Shape Grammar

TPL Textual Programming Language

VPL Visual Programming Language

1

Introduction

Contents

Over the last decades, research and development efforts have been conducted aimed at further enriching technological integration in graphically oriented environments, such as the Architectural, Design, and Engineering domains [2]. This movement, spanning from its early conception in the early 1960s onwards, resorted to computation-based approaches that ultimately revolutionized the aforementioned domains due to the, now indelible, benefits brought alongside them. Amongst many other advantages, as we can observe in Sutherland's seminal contribution [3], and equally relevant works [4], automation and increase in productivity served as the pivotal catalysts of notable sequential breakthroughs resulting in the emergence, and consequent development, of Computer-Aided Drafting (CAD) and Computational Design (CD) technologies: two closely related and interchangeable approaches where computational processes and algorithms, respectively, serve as the basis of Bi-Dimensional (2D) and Three-Dimensional (3D) modeling. As a result, the use of CAD and CD solutions extended to a multitude of different Areas of Knowledge—its incorporation in Architecture being one of the most notable examples.

With an increase in use throughout the years, CAD software—with *Autodesk's AutoCAD*[1] being a pinnacle—has robustly established itself and eventually contributed with accessible Graphical User Interfaces (GUIs) capable of serving the graphical needs inherent to the previously mentioned domains. However, despite the broad range of 3D operations that such GUIs offer, their interactive capabilities prove burdensome, time-consuming, and insufficient when scaling them to complex models [5]. One can point out a wide range of examples covering these cases; one such example that caught our attention pertains to architectural workflows where style variations in already-defined models (e.g., buildings) are desired: something that requires changes in ornamental details without directly affecting their structural representations. In cases such as these, where the complexity of a model entails a network of interconnected dependencies (structural/ornamental), and the application of changes in already-established models for producing different stylistic variants is desired, the aforementioned traditional interactive capabilities do not suffice:

> "*as hardware graphics capabilities continue to improve, the key limiting factor for the inclusion of rich ornamentation in the built structures of computer games is the time it takes a human artist to create such patterns*" [6, p. 2]

Such observation acknowledges that the difficulty in enriching a shape's ornamentation and, by consequence, its complexity, does not necessarily derive from hardware limitations; rather, from the time it takes a CAD user to execute the required modeling processes to attain desired outputs; thus stressing and reflecting on the aforementioned time-consuming related limitations of conventional CAD means. It is then evident that, in order to mitigate this concern, one must facilitate the processes allied to already-established modeling conventions and contribute with appropriate abstractions capable of providing possible solutions that complement these requirements. Furthermore, with the ever-increasing relevance of virtual environments in fields such as video games [7] and cultural heritage [8], the previous observation proves significant and reveals an underlying limitation. On that note, these previous observations constitute the motivation that drives this thesis.

Following this motivation, this study aims to address these limitations on complex modeling processes pertaining to the inclusion, variation, and combination of structural and ornamental details in built structures, while studying, conjugating, and reflecting upon appropriate measures to tackle them. In this regard, we have devised the following research questions pertaining to the inclusion of ornamental characteristics in virtual environments:

- Provided a standardized set of Gothic floor plan conventions, how can one resort to a generalized Algorithmic Design (AD) solution as a means of adapting to particular designs? (e.g., *Cathédrale*

[1] https://autodesk.com/autocad

Notre-Dame de Paris[2] or *Cathédrale Notre-Dame d'Amiens*[3]);

- Provided a set of walls, how can one change a specific wall configuration to include assorted ornamental elements? (e.g., windows, arches, rosettes, among others);
- Provided a set of pillars, how can one change a specific pillar configuration to incorporate different designs? (e.g., Gothic style or Eclectic style);
- Provided a set of windows, how can one change its ornamental characteristics such that various stylistic outputs are derived with minimal effort?

To address these research questions, we delineated a set of hypotheses based on a methodology that we will delve into throughout this document. This methodology should be capable of addressing optimized modeling processes, conforming to use cases of structural and ornamental variations, by employing the generalization of structural representations that later embody equally generalized shapes capable of including different ornamental designs. With these generalizations in place, it is then possible for both these modeling aspects to encompass a multitude of different styles by applying accessible changes with minimal modeling efforts and little to no impact on the model's core structure. As such, it seems appropriate to consider convoluted architectural case studies where 2D/3D modeling procedures are heavily put to the test. Following this rationale, and referring back to the previously presented research questions, we have chosen Gothic Architecture as our case study due to its intricate and ornate style: encompassing a wide range of ornamental and structural schemata that serve as complete examples of scalability in design.

Regarding the limitations allied to CAD environments, CD fell subject to an inevitable finer-grained partitioning encompassing different subdomains—with varying degrees of contrast and purpose—and surfaced as an extension to CAD. This partitioning manifested itself through the derivation of different paradigms, based upon the combination of fields, and the consequent emergence of different terminologies referring to them [9]. Amongst many other branches, this led to two interrelated approaches that gained prominent traction: Parametric Design (PD) and Procedural Modeling (PM). At the core of these approaches lies AD, where modeling processes are employed via algorithmic procedures [10], instead of direct geometrical manipulation. Both PD and PM involve the use and conjugation of parameters to describe designs [11], with the latter relying on a sequential/procedural application of modeling processes that recurrently resort to elements of the former. Furthermore, these approaches allow for much more flexible and scalable means of modeling when compared to conventional procedures [12]; thus deeming these new branches a suitable alternative. As emphasized in the coming sections, these techniques frequently serve as a basis for describing highly complex geometric shapes where flexibility

[2] https://en.wikipedia.org/wiki/Notre-Dame_de_Paris
[3] https://en.wikipedia.org/wiki/Amiens_Cathedral

and design variations are of the utmost necessity: something which we will be focusing on (with Gothic cathedrals being at the epicenter of our study).

1.1 Document Structure

This document is broken down into six chapters: (1) Introduction, this very chapter, where we briefly introduced CAD, its limitations when the modeling of complex architectural models is concerned (tracing the motivation that drives this study), and a set of research questions and hypotheses on how to address them; (2) Background, providing an overview of conventional CD procedures, such as PD, PM, and AD, followed by an emphasis on their importance to modeling workflows, and additional explanations to better understand the chapters to come; (3) Related Work, where we delve upon a finer-grained list of solutions (rule-based and scripting-based), allied to the previously mentioned approaches, and devise a reflection on the advantages and disadvantages of such methods applied to our study; (4) Solution, where we expand on the purposes of our study, together with a short explanation on how previous approaches have tackled our case study, and how we re-implemented and extended these ideas following optimized methods; (5) Evaluation, where we explain and test our solution to the modeling of Gothic windows and cathedrals encompassing various different styles; and (6) Conclusion, where we present concluding remarks, followed by observations on the trade-offs of our approach and future work.

2

Background

Contents

In this chapter, we present a brief introduction of the relevant methodologies, techniques, and modeling paradigms that lay the foundation of the studies explored in the coming chapters: ranging from core modeling approaches to additional techniques that extend them by allowing for greater control and variations in design (based on a model's parametric schemata).

2.1 Parametric Design

PD, a subdomain of AD, is a CD approach capable of generating and manipulating complex geometric entities through the identification, and subsequent definition, of parameters that describe the relations and interdependencies between various objects that, when combined, define these geometric entities as a whole [13]. Naturally, by following the benefits of CD and by further expanding on them, PD approaches

quickly established as a fundamental tool for the exploration and handling of architectural challenges that inherently contain complex networks of object relations [14]. To exemplify, certain building elements can be parameterized by defining their structural parts, namely floor plans, walls, windows, among others, with each of these entailing different parametric features. These features, ranging from dimensional data to structural topology, can later be referred to as a means of flexibly controlling and changing a model's representation in such a way that, despite its complex underlying network of relations, pays due respect to the corresponding constraints held between them [15]. This, in turn, allows a designer to search for alternative variations of a given model by simply changing its parameters, which can be accomplished with great combinatorial outcomes by the sheer nature of computational processes.

PD has already proven its interdisciplinary and extensive potential, as can be observed by its broad applications ranging from architectural modeling [16] and structural optimization [17], to the planning of urban environments and video games [18]. Furthermore, several well-established PD categories/ conventions are recurrently applied within these contexts, the most remarkable ones being Parametric Variations (PVs), Parametric Combinations (PCs), and Parametric Hybridizations (PHs).

The coming notions, based upon Hernandez's study [16] and presented alongside a set of examples from the same study, reflect upon parametric changes when applied to the schemata of column designs:

Parametric Variations: A PD approach responsible for performing variations on parameterized entities that hold no effect on topological characteristics (i.e., its components and corresponding relations); rather, its dimensions, scaling, among others. This is accomplished by controlling, modifying, and performing different value combinations via the model's parameters.

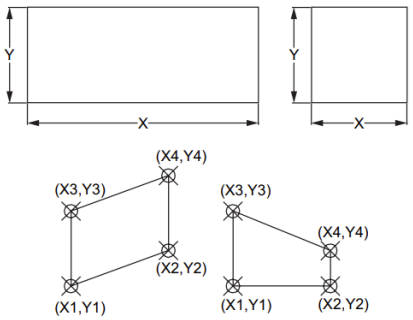
Hernandez refers to a basic quadrilateral that can have multiple representations formally defined by different parametric schemata: its width/height and its vertex coordinates. To visualize this, we have Figure 2.1;

Parametric Combinations: An additional PD category accountable for generating new designs based on the combination of geometric entities. Here, the focus lies in combining multiple shapes constituting a model and corresponding spatial relations. These components can then be replaced and combined by different shapes, ultimately resulting in different designs/outputs.

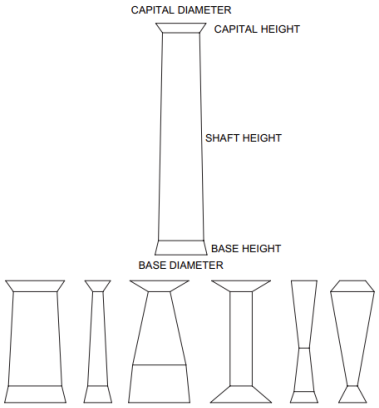
Serving as an example, the author refers to the structural partitioning of columns by dividing them into three constituents: base, shaft, and capital. In doing so, one could perform different design combinations in each of the mentioned components; thus allowing for a multitude of different column designs/outputs. This can be observed in Figure 2.2;

Parametric Hybridizations: As the name implies, PH is a hybrid PD approach that takes into effect the combination of both PV and PC procedures. Figure 2.3 showcases such a combination.

With these three approaches in mind, one can effectively generate a variety of different designs stemming from a model's initial parametric representation, provided one conceives a parametrically flexible



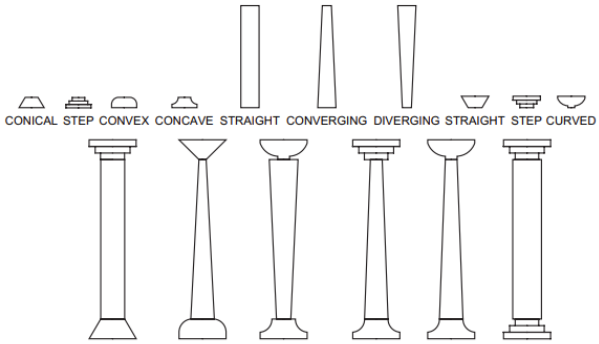
(a) Variations on two distinct quadrilateral schemata: width/height (top) and vertices coordinates (bottom).



(b) Column constituents schema (top) and corresponding variations (bottom).

Source: PV examples from [16].

Figure 2.1: Application of PV procedures to two distinct models: (a) depicting variations in a simple quadrilateral and (b) showcasing variations applied to column constituent dimensions.



Source: PC examples from [16].

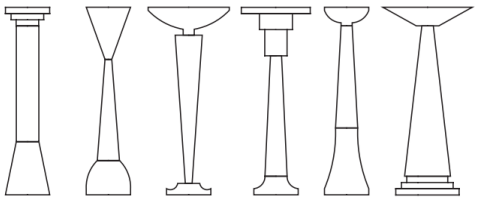
Figure 2.2: Different column constituents models (top) and corresponding application of PC procedures (bottom).

representation—something that should be carefully and thoroughly considered. Finally, to apply these principles in design, PD procedures are often incorporated within a myriad of modeling approaches, with PM being the most flexible and overarching choice [19].

2.2 Procedural Modeling

As described in Krispel et al. survey [20], to better define PM one must first acknowledge the generative modeling paradigm: the application of modeling techniques where shape description relies upon a set of sequential operations that, when performed, conceive the shape's final form.

Having specified that, and following additional definitions in Smelik et al. survey [21], PM is a gener-



Source: PH examples from [16].

Figure 2.3: PH procedures: a combination of PVs and PCs applied to column schemata (in this case its dimensions) and constituents, respectively.

ative modeling technique capable of creating a wide gamut of generative representations, be it nature-inspired processes (e.g., recursive phenomena observable in tree growth) or man-centered processes (e.g., buildings). This is accomplished by employing algorithmic procedures directly or indirectly defined via human intervention: one can program the whole procedure, leading to static outputs, or define flexible procedures that generate content semi-automatically; hence increasing the dynamic capabilities of such representations. As a result, PM is broadly applied to a panoply of contexts—ranging from Physics-based simulations to architectural modeling—and currently stands as an active research topic. Additional examples, as mentioned in the previously cited survey, comprise the modeling of virtual environments and their constituents: terrains, vegetation, water bodies, roads, city layouts, buildings, among others.

Apart from its extensive application scope, the authors also refer to the numerous advantages offered by PM—the most relevant ones being data amplification and data compression. They proceed to emphasize that, while data amplification refers to the ability to yield a wide variety of outputs derived from the combination of a representation's input parameters, data compression refers to the compact way that geometric models are represented, given that, in most PM approaches, these are represented by means of programming functions, rather than statically stored geometric primitives (i.e., triangular meshes, among others). One can later execute such functions and obtain a 3D model on-demand while steering away from wasteful resource management related to physical storage overheads.

Useful and well-defined methodologies following PM notions have established themselves throughout the years, with their nuclear branches being rule-based and scripting-based approaches. The following subsections will dive into each of them.

2.2.1 Rule-Based Approaches

Drawing clear inspiration from grammars [22,23], and the recurrent application of such concepts to computational procedures [24] (e.g., finite-state processes), shape generation via rule-based approaches involves incrementally transforming a shape by applying a series of predefined rules. This is first em-

ployed starting with basic geometric elements (i.e., initial states), followed by continuous derivations until a determined terminal state is achieved.

Two particular branches following this paradigm are actively applied within shape-generation contexts: *Shape Grammars (SGs)* and *L-systems*.

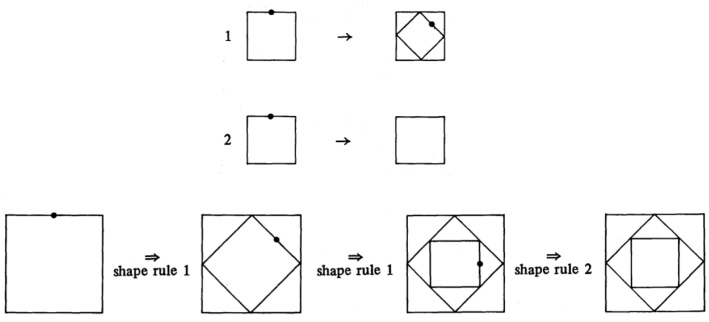
Shape Grammars: Formally introduced by Stiny [25], *SGs* constitute a framework that allows for the definition and manipulation of shapes via shape sets and rules. More specifically, Stiny defines a shape as being:

> "*a limited arrangement of straight lines defined in a Cartesian coordinate system with real axes and an associated Euclidean metric.*" [25, p. 1]

Once these aspects are defined, it is then possible to employ succeeding derivations, via shape combinations, that follow simple set theory-based *Boolean* operations (*union*, *intersection*, and *difference*) or *Euclidian* transformations (*translation*, *rotation*, *reflection*, *scale*, and corresponding compositions).

In order to identify intermediate states apt for transformation, a constituent of its corresponding shape must be labeled (e.g., point). Finally, a *SG* is derived, as illustrated in Figure 2.4, whilst obeying the following formalism:

1. A finite set of shapes S
2. A finite set of symbols L (each symbol associated with a label pertaining to a given shape)
3. A finite set of shape rules R, with each rule following the form $\alpha \rightarrow \beta$ (where α and β are labeled shapes belonging to the set S)
4. An initial labeled shape I



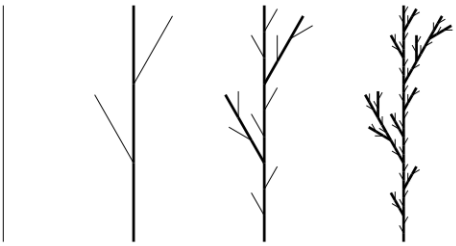
Source: *SG* derivation steps from [25].

Figure 2.4: *SG* derivation (bottom) following *SG* rules (top/middle).

This was later expanded to include the definition of 3D shapes by elevating sets of lines and points to a higher degree of freedom, and by entailing volumetric spaces defined by additional parameters

such as width, length, and height; thus bringing wider applicability to this approach. Examples include Konig and Eizenberg's [26] method to generate Frank Lloyd Wright's prairie houses.

L-systems: Albeit similar to *SGs*, *L-systems* were formally introduced by Aristid Lindenmayer [27] as a system capable of describing and modeling the behavior of botanic growth/development; thus allowing for a visual representation of such phenomena [28,29]. More specifically, as stated in Krispel et al. [20] survey, *L-systems* share the same paradigm as that of *SG*, but with a significant difference: instead of resorting to sets of points to formulate a shape, *L-systems* make use of strings and symbols as the entities to which rule application is applied. Consequently, in order for these strings to be visually translated, there needs to be a mechanism capable of interpreting them. This mechanism is often referred to as *turtle graphics*: a way of drawing through vector graphics by following simple commands pertaining to three attributes, those being a *Cartesian* location, orientation, and, finally, a pen containing characteristics such as color, width, and current status (on/off) [30, 31]. As stated in the survey, these commands are then interpreted through the strings and symbols of the given *L-system's* implementation and rendered by specific engines. Figure 2.5 illustrates the derivation steps of an *L-system*.



Source: *L-system* derivation from [32].

Figure 2.5: *L-system* derivations forming recursive branching structures (for simplification purposes, sets of strings, rules, and interpreters are omitted).

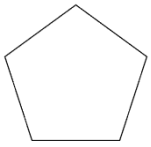
In spite of being an isolated approach on its own, *SGs* are often paired with dataflow-based environments, most of which resort to Visual Programming Languages (VPLs), due to their accessible and user-friendly qualities [33]. However, as with anything, trade-offs regarding user-friendliness and flexibility must be taken into consideration when choosing the best strategy to handle the modeling purposes at hand. On that note, several studies have made clear that despite the friendliness of VPLs as a design tool, these constitute notorious scalability concerns when the modeling of complex architectural projects is concerned, with a suitable alternative for these cases being Textual Programming Languages (TPLs) [12].

2.2.2 Scripting-Based Approaches

Following the definitions in Krispel et al. [20] survey, scripting-based approaches make use of TPL paradigms for the definition of shapes. Consequently, generating geometric entities becomes a matter of leveraging the set of operations and data structures inherent to a given programming language of choice—most of which are facilitated by external libraries specifically designed for geometrical representations [34]. On that note, a multitude of libraries—exclusively dedicated to these purposes—are referred to in this survey (these being much more prominent in scripting-based approaches as opposed to their counterparts). These offer a higher-level framework where low-level modeling operations are abstracted away from its users through an Application Programming Interface (API); thus serving basic and complex modeling needs through much more convenient and approachable means. As such, shape generation can be as simple as applying a sequence of invocations encompassing conveniently named functions.

With an extensive spectrum of frameworks and an equally substantial set of operations inherent to this approach, plenty of examples can be derived. However, focusing on our framework of choice (*Khepri*)—whose choice reasoning will later be explained—, we will illustrate a small subset of operations fitting within ensuing definitions as categorized by the previously mentioned survey. Furthermore, by exemplifying them through *Khepri*, these illustrations will also serve as a basis for understanding future modeling choices of similar nature. Finally, it is worth mentioning that with proper and complete implementation of these groups, one can model a wide spectrum of shapes with various degrees of complexity.

Instantiations: These are operations that are responsible for creating, often basic, shapes (e.g., polygons and platonic solids). Figure 2.6 illustrates two programming functions capable of instancing two models with different degrees of freedom;



(a) Output of function `regularpolygon(5)` (its parameter being the number of sides).

(b) Output of function `box(xyz(2,1,1), xyz(3,4,5))` (bottom left and upper right corners parameters).

Figure 2.6: Scripting-based instantiations: (a) and (b) depicting 2D and 3D instantiations, respectively.[1]

Binary Creations: A series of procedures that perform set theory-based operations involving two or more shapes (e.g., *Boolean* operators, such as union and intersection). Figure 2.7 showcases a set

[1] Original examples from https://web.ist.utl.pt/~antonio.menezes.leitao/Docs/Books/BookJulia/Modeling/index.html

of *Boolean* operations when applied to 2D surfaces;

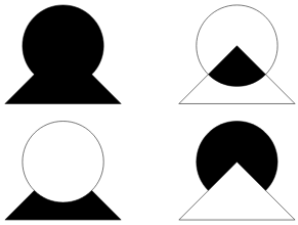
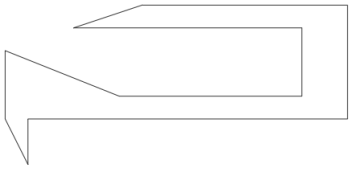


Figure 2.7: *Boolean* operations when applied to two overlapping shapes of the form `<boolean_operation>(a, b)`, with *'a'* and *'b'* being either shape (depending on the desired effects); *union* and *intersection* operations are presented at the top row, followed by two *subtraction* operations on the bottom row (with the order of the parameters deciding which shape is kept).[2]

Deformations and Manipulations: The backbone of most complex modeling operations. These are particularly useful when performing shape morphing, displacements, and many more (e.g., scaling, offset, extrusion). Figure 2.8 depicts a surface to which an extrusion is applied.



(a) A polygonal surface representation stored in a variable `polygon`.

(b) An extrusion performed to the polygonal surface in (a) `extrusion(polygon, vz(1))`.

Figure 2.8: Scripting-based manipulations: (a) and (b) depicting a basic polygonal surface and its extrusion, respectively.[3]

As previously mentioned, one of the core advantageous aspects related to PM is its ability to drastically reduce the workload of modeling tasks. In the case of architectural assets, as mentioned in the survey, such is particularly useful when dealing with repetitive properties, be it of ornamental or structural nature (e.g., ornamentally equivalent windows, among others). At the basis of this case, the survey touches upon the fact that once a final shape description is achieved by algorithmic means—if such is recurrently found within a given general structure (e.g., equal windows in a building)—it is then possible to reuse this code and apply it to missing forms with no additional effort. Furthermore, if one wishes to perform parametric modifications that ultimately lead to different stylistic outputs, such can be accomplished by appropriately tweaking the parameters/arguments of the function—provided, of course, that said function was carefully programmed in such a way that allows for a flexible parametric description. This further emphasizes a particular instance where PD and PM work in unison and are interchangeably used.

[2]Original example from https://web.ist.utl.pt/~antonio.menezes.leitao/Docs/Books/BookJulia/Complex_Shapes/Surfaces/index.html

[3]Original examples from https://web.ist.utl.pt/~antonio.menezes.leitao/Docs/Books/BookJulia/Complex_Shapes/sec_extrusions/index.html

2.3 Artifical Intelligence as a Possible Approach to Modeling

In addition to the already expanded-upon approaches, a series of supplementary techniques are equally found in CAD/CD workflows. Nonetheless, due to these being either less prominent and/or of lesser appeal to our study, we will merely refer to one approach that is currently in vogue for its interdisciplinary potential and that may prove promising to our case studies in the near future. On that note, what follows is but a brief observation that may be later referred to, but never to the same extent as the approaches described in the previous sections.

Following recent promising developments in Artificial Intelligence (AI), Machine Learning (ML), and corresponding multimodal generative systems, one could argue that adopting and leveraging their generative capabilities would effectively address the already-stated limitations. However, except for certain cases (data sets containing vectorized floor plan CAD drawings for semantic labeling [35]), accurate data sets for training purposes are mostly underdeveloped. This is especially evident when considering architectural styles and ornaments, where previous studies suggest *Style Transfer*, through the use of Neural Networks, as a possible approach to automatically incorporate and/or change a building's architectural style [36]. However, such means exclusively resorted to 2D data (i.e., images) rather than their corresponding 3D models [37]. As such, this suggests that ML currently proves unsuitable to address 3D modeling procedures that preserve a finer-grained control, on the user's end, over a given model—especially when higher and more complex levels of detail are concerned.

2.4 A Brief Introduction to *PostScript*

Provided a relevant study that will be introduced in the coming sections, which makes use of *PostScript*, together with its characteristics—that greatly contributed to our solution's trajectory—this section serves as a brief introduction to *PostScript*.

In the aforementioned study, which refers to a modeling framework, the authors decided to opt for *PostScript* as a TPL for modeling purposes: since it was closely related to 2D graphic operations that could, according to their understanding, be seamlessly adapted to modeling operations akin to those of CAD environments—the reason being that *PostScript* was an industry-standard control language used in printing devices, where the interpretation of a program would lead to document printing as a side effect.[4] The authors have then repurposed *PostScript* into a Domain-Specific Language (DSL), by applying the same syntax and contributing with further extensions for shape modeling. To better understand how one would resort to the aforementioned DSL as an AD tool, we will now present a brief introduction to the *PostScript* programming paradigm.

[4] https://en.wikipedia.org/wiki/PostScript

PostScript is a stack-based programming language, following postfix/reverse polish notation,[5] where the interpretation of a program is accomplished by adding (*pushing*) and removing (*popping*) operands to and from a stack, respectively. Consequently, executing a given function requires *popping* operands from the stack, processing them, and *pushing* them back in, whilst making use of said stack as a buffer for any intermediate operations across the function's execution—the result, then present in the stack, will be used by any procedures to come.

A broader and complete reference covering all possible *PostScript* rules can be found in *Adobe's PostScript Reference Book* [38]. However, the coming example serves as a simple explanation to better understand the previously mentioned modeling framework. Let us, then, suppose that we wish to perform the following arithmetic procedure: $(5 + 7) \cdot 8$. Translating this to *PostScript* would result in the following code snippet: `5 7 add 8 mul`. To illustrate how *PostScript* interprets this procedure, Figure 2.9 demonstrates how a stack-based programming language works.

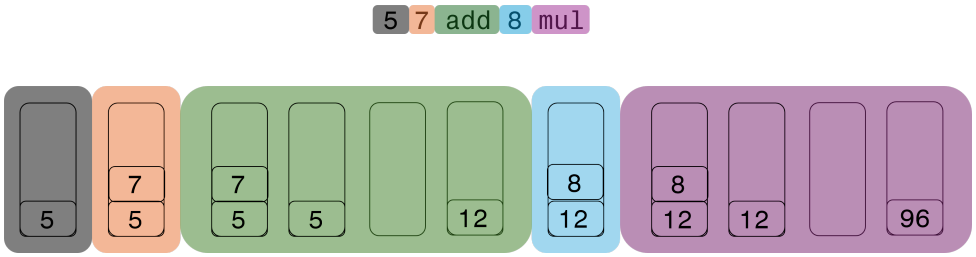


Figure 2.9: The inner workings of a stack-based programming language following the *operand stack's* states when interpreting a basic mathematical procedure.

Looking at the figure above, each operand and operator of a given procedure is tokenized. For illustration purposes, each token is conveniently labeled, with a given color, to serve as an association measure with what happens in the *operand stack* at the stage in which the corresponding labeled token is interpreted. From here, we can understand what happens when interpreting a procedure: whenever an operand is identified, it is pushed into the stack (examples include the gray, orange, and blue stages of Figure 2.9). However, when an operator is recognized, the stack's items are individually popped by as many times as the arguments required for the operator (e.g., multiplication requires two arguments), followed by an application of that procedure, whose output is then pushed back into the stack (examples include the green and purple stages of Figure 2.9).

[5] https://en.wikipedia.org/wiki/Reverse_Polish_notation

3

Related Work

Contents

In this chapter, we explore different approaches to AD that are particularly oriented toward architectural use cases, due to our focus on simplifying the inclusion of ornamental representations in structural models. Furthermore, since many of the solutions that fit within our purpose follow either rule-based, parametric-based, or scripting-based approaches, we have conveniently partitioned them into different sections, those being Section 3.1, Section 3.3, and Section 3.4, respectively. Finally, we derive a reflection on the strengths, weaknesses, and trade-offs inherent to these techniques and later refer to them as a basis for establishing a rationale that drives our solution.

3.1 Rule-Based Solutions

Most CD solutions that make use of rule-based approaches fall within the realm of VPLs and are often coupled with GUIs as a means of establishing grammar rules and defining start, intermediate, and terminal shapes. This has to do with the fact that, as previously explained in Section 2.2.1, such solutions often pair well with dataflow-based environments.

The ensuing subsections will delve into rule-based solutions specifically targeted towards the modeling of complex urban environments: Section 3.1.1 presents *CityEngine*[1]—a system that automatically generates virtual cities—and focuses on the techniques responsible for the generation of buildings, façades, and windows; Section 3.1.2 represents an overview on *Computer Generated Architecture (CGA)*[2]—an extension to *CityEngine* for building generation improvements—and mentions important aspects on avoiding incoherent architectural designs; finally, Section 3.2 comprises a reflection on the advantages, disadvantages, and trade-offs of rule-based approaches to better guide our solution.

Despite our intentions to study particular cases of structural and ornamental variation on Gothic styles, and the slight diversion that the forthcoming solutions represent, these are still of interest as they encompass worthy aspects related to the modeling of buildings and corresponding subsets—where parametric elements pertaining to façades and windows are considered. Additionally, it is worth reminding that focusing on Gothic Architecture as a case study was purposefully considered due to its inherent complexity; thus allowing us to apply our methodology to general use cases of simpler nature. Finally, as presented in Section 6.1, these may represent useful considerations for further studies if one wishes to incorporate our solution within workflows where the automatic generation of virtual environments is desired.

3.1.1 *CityEngine*

CityEngine, stemming from a study by Parish and Müller [18], is a system capable of modeling a virtual city whose initial implementation relied on *L-systems* for the generation of complex urban environments and the main sub-systems that comprise them. This approach, taking into consideration the slower changing rate of urban sub-systems [39], reduces the modeling of a city to the generation of traffic networks and buildings alone. On that note, *CityEngine* is supported by a pipeline comprised of (1) roadmap generation, via 2D map processing and *L-system* derivation; (2) building allotments definition, inside subdivided empty areas between roads; (3) building generation, through *L-system* derivation and binary creations, and (4) output data interpretation, for visualization purposes, via a parser.

Despite its interesting properties to automatically derive a city's layout, together with its buildings, later iterations of *CityEngine* incorporated additional measures to mitigate some of its limitations. These

[1] https://www.esri.com/en-us/arcgis/products/arcgis-cityengine/overview
[2] https://doc.arcgis.com/en/cityengine/latest/help/help-cga-essential-concepts.htm

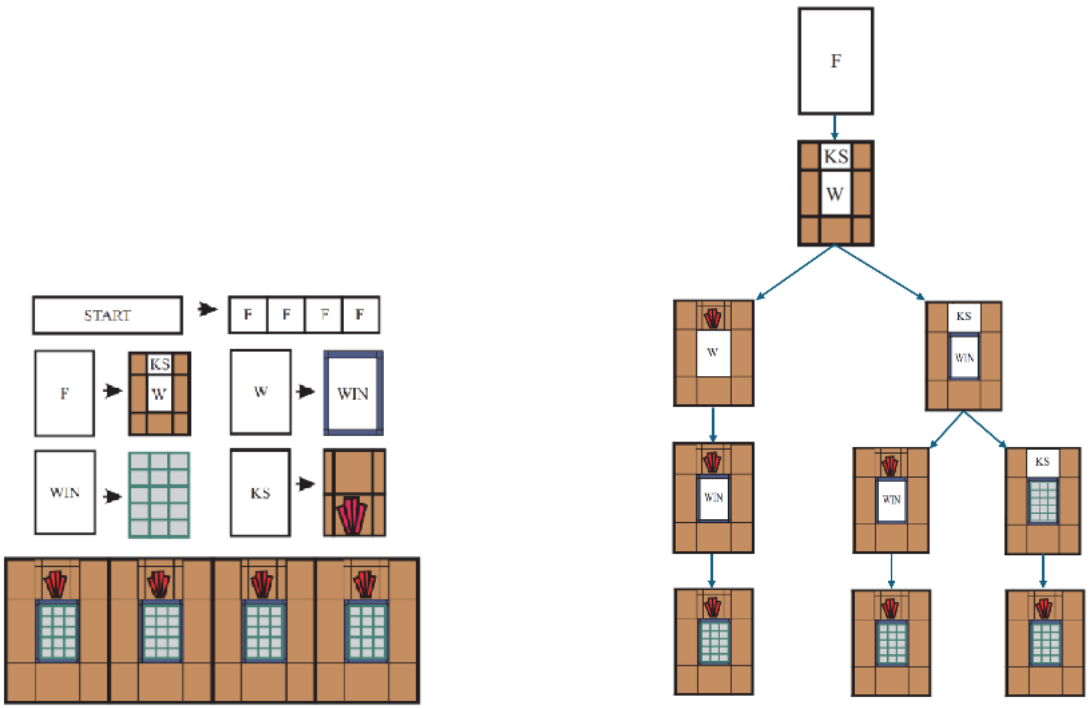
limitations, as mentioned in a study by Wonka et al. [40], concern the restricted nature of *L-systems* when the generation process of buildings is concerned: its inadequacy when conforming to strict spatial constraints; thus stressing the fact that these are much more appropriate when simulating growth in open spaces where spatial constraints are not as rigorous, such as roadmaps. Because of that, *L-systems* were then solely constrained for the generation of roadmaps. Consequently, this new study proposed a different approach for the automatic modeling of buildings that resorts to a combination of two distinct design grammars based on the main concepts derived from *SGs*. To that end, the authors proposed factoring out design ideas from the grammar itself while resorting to a mechanism capable of distributing them spatially, in a coherent manner, by selecting appropriate rules and excluding unsuitable instances. As a result, the main contributions of this study are (1) split grammars, (2) control grammars, and (3) an attribute-matching system. When geometric finishes are concerned, the authors resort to additional measures that make use of attributes associated with symbols attached to either shapes or rules of a given grammar. These attributes serve two distinct purposes: (1) to encode and propagate material information (e.g., building style, wall color, etc), and (2) to aid in rule derivation processes by selecting specific rules that preserve architectural coherence; thus playing a major influence in the derivation process. For that purpose, the authors implemented three possible ways of assigning attributes within the framework: (1) associating attributes with start symbols, (2) transferring attributes from a parent shape to succeeding generated shapes via the split grammar, and (3) associating attributes via a control grammar.

Attribute-Matching System: The attribute-matching system, as stated by the authors, includes, excludes, prioritizes, and provides preference distributions over the rules within a given grammar. This is done so, in a particular derivation step, by comparing the attributes specified in the rules with the attributes associated with the current grammar's symbol and, finally, by selecting the most appropriate rule;

Split Grammars: As explained by the authors, a split grammar is a modified type of set grammars responsible for deriving 3D layouts of buildings out of simple, attributed, and parameterized shapes by, firstly, generating a façade and then splitting it down to its smaller constituents (i.e., windows, cornices, among others). For illustration purposes, Figure 3.1 showcases a split grammar's rules alongside its derivation steps;

Control Grammars: As described by the authors, control grammars spatially distribute design decisions in a way that obeys architectural principles (i.e., attributes pertaining to a given floor either vary significantly or slightly, but should always remain consistent). Its rule selection process is the same as that of split grammars: via an attribute-matching system.

Succinctly speaking, the authors' proposal revolves around an alternated application of split grammars and control grammars, with each one making use of an attribute-matching system for the sake of ar-



(a) Split grammar rules (top) and output derived from the start shape (bottom).

(b) Split grammar derivation: for optimization purposes, the shortest path is usually chosen.

Source: Split grammar application from [40].

Figure 3.1: Split grammar application: (a) showcases the split grammar's rules and outputs, and (b) depicts all possible derivation steps (if more than one design pertaining to the same attribute exists, all derivations are considered and the output is chosen by the attribute-matching system).

chitectural coherence. This *CityEngine* iteration contributed to overcoming notable challenges allied to previous approaches, by allowing for a set of procedures that prevent shapes from intersecting each other—a recurrent problem of *L-systems*-based building derivation—, together with the addition of a rule selection mechanism aimed at avoiding architecturally incoherent derivations.

On a final note, the authors enumerate a set of advantageous and disadvantageous aspects of *SG*s when applied to building design, namely the essential role that a *SG* plays in describing the basis of architectural designs, and its yet-to-be-proven suitability for automatic and semi-automatic modeling, respectively. Moreover, they also discuss their proposal's flexibility, complexity, and usability, by emphasizing their framework's freedom when adding new grammar rules and/or attributes to an already-established grammar, while pointing out that such a case is not necessarily trivial as it requires familiarity with the intricacies of the split grammar approach. Additionally, the authors warn that, despite the framework's ability to derive various outputs (depending on the grammar's extent), it proves limited when the generation of highly complex architectural details is required.

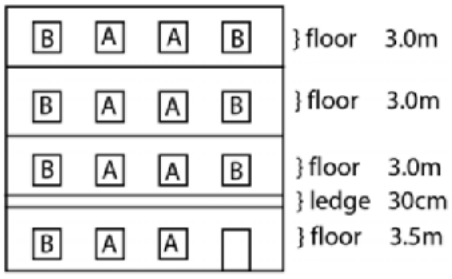
3.1.2 *Computer Generated Architecture*

Müller et al. [41] contributed with an extension to *CityEngine* by the name of *CGA*. This extension aims at addressing hindrances of the previous solution, such as the risk of undesirable geometrical intersections and the insufficiency of split rules when it comes to complex mass model generation. The authors' proposal overcomes these impediments by sequentially adding details via production rules that generate a building's mass model, followed by its façade, windows, doors, and ornamental details. Consequently, *CGA* constitutes a major advantage when compared to previous solutions and incorporates the following contributions: (1) a procedural methodology capable of modeling detailed buildings, with roofs and rotated shapes, stemming from consistent building shells unrestricted to axis alignment, and (2) addressing application-related details through the definition of important shape rules, followed by pre-established modeling examples.

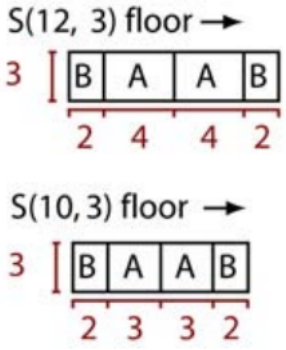
Being an extension of previous work, *CGA* inherits split rules and further extends them with an additional set of rules and splits (splits, as previously mentioned, being a mechanism that divides a building's layout into smaller constituents):

Scope Rules: Rules oriented toward shape modification via translations, rotations, and scaling; thus defining the position, orientation, and size of a shape: 3D scopes are used to mass models, while 2D scopes are to be aligned with the mass model's façade and roof surfaces (via the extraction of faces, from the 3D model, through component splits);

Basic Split Rules: Responsible for splitting a scope along a given axis (e.g., a façade can be split into floors, via the vertical axis; and windows, via the horizontal axis); the extension of such splits is restrained to a pre-defined scale. Figure 3.2 demonstrates these principles;



(a) Basic splits: vertical splits comprise the subdivision of a façade into floors and ledges (following specified scalings); horizontal splits are responsible for partitioning floors into windows and doors.



(b) A closer look into the effects of scaling along a horizontal scope.

Source: Basic split examples from [41].

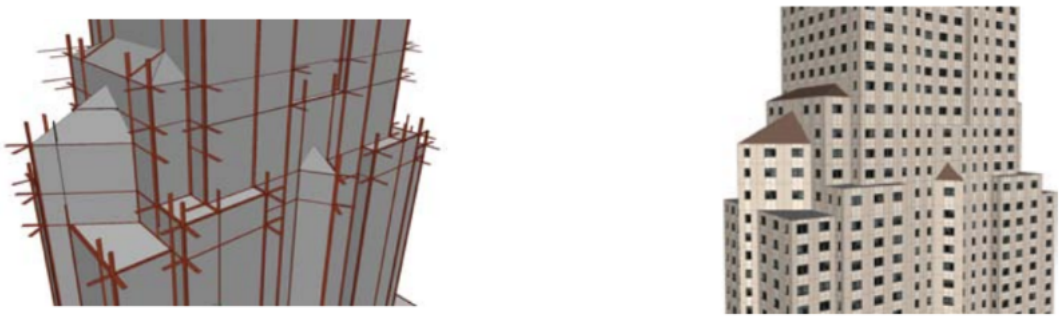
Figure 3.2: Basic split application: (a) illustrates two different splits (vertical and horizontal) when applied to a façade, and (b) depicts the effects of rule scaling.

Repeat Splits: A specific type of split that divides a shape by as many splits as defined, while

adapting its split's scale to the shape's scope according to the number of repetitions;

Component Splits: A split that can be applied to scopes containing information pertaining to one or more degrees below three dimensions (i.e., faces, edges, vertices).

CGA also resorts to two mechanisms—occlusion and snapping—that test for spatial overlaps and search for relevant lines and planes in a shape's configuration; thus providing meaningful information to delimit applicable grammar rules within a given context (e.g., a mass model's surface that is fully occluded can never be converted to a surface with doors or windows). Most importantly, the snapping mechanism, as defined by the authors, extracts snap lines from the intersections between a façade's 2D plane and the global planes (i.e., all other faces of the same shape)—these are the delimiters of the façade's surface area. This snapping mechanism further contributes to conforming split rules within the spatial context that they delimit. Figure 3.3 showcases the application of the snapping mechanism to a mass model, together with its final output as generated by subsequent processes.



(a) Snap lines (in red) of a given mass model, as identified by the snapping mechanism.

(b) A building, as derived from the mass model in (a), after applying all *CGA* procedures.

Source: Snap line examples from [41].

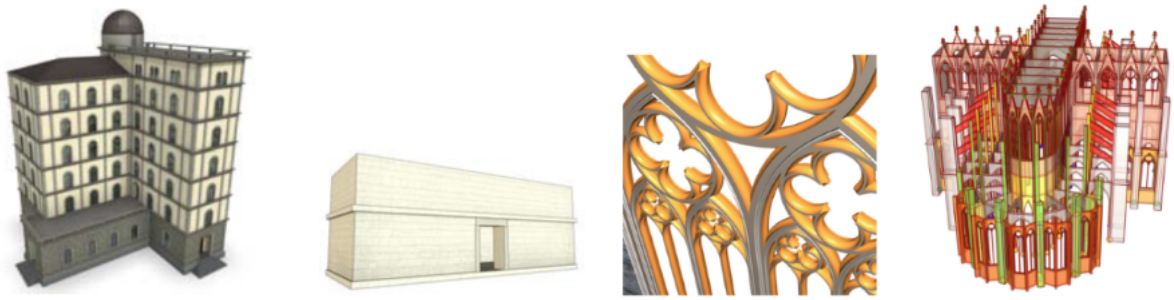
Figure 3.3: An example of a *CGA* output: (a) showcases the derivation of snap lines from a given mass model, while (b) depicts one possible output stemming from the same model.

To conclude, the authors discuss important observations concerning *CGA*, arguing that an invited professional modeler, with minor guidance and support, was able to create a small city with multiple variations. In addition, they underscore *CGA's* significant contribution to the procedural generation landscape—given its ability to drastically reduce modeling times of a given scenario—and emphasize that three software companies have demonstrated interest in *CGA* due to cost-reduction benefits of applying such an approach to the automatic generation of highly dense and detailed virtual cities for various media purposes. Notwithstanding, *CGA* still faces certain limitations: (1) writing *SG* rules can be complex and may produce unwanted side effects (especially as the grammar's vocabulary increases), (2) its application to broader architectural designs is underexplored and may require drastic changes to its implementation, and (3) the lack of real-time rendering tools attached to this workflow's environment.

3.2 A Reflection on Rule-Based Solutions

Over the years, various studies incorporating the previously mentioned systems have been conducted, with valuable outlined conclusions along the way. Saldaña's study [42], for instance, resorts to *CGA* as a means of implementing a set of rules for the generation of 3D models encompassing architectural styles and conventions of the Roman period. One important observation made by the author is that, throughout the modeling process, these rules had to be rewritten multiple times as new parametric definitions of the 3D models had to be designed. Consequently, provided the already identified limitations of *CGA*, such recurring processes may produce unwanted side effects in later stages of the modeling process. Moreover, another observation made by the author is that *SG* usability and extensibility are restrained by the framework within which they are implemented; thus stressing the fact that *CGA* lacks additional procedures to describe curvilinear geometrical entities—further burdening the modeling of building types that encompass these geometric features.

In a different study, Müller et al. [43] also resort to *CGA* for modeling purposes by implementing the necessary rules to describe a sub-type of Mayan architecture. However, this architectural style is rather simplistic when compared to other styles that we wish to incorporate in our proposal. This difference in style complexity is further illustrated in Figure 3.4.



(a) A simple building. **(b)** A Mayan temple. **(c)** A Gothic window. **(d)** A Gothic cathedral.

Source: Modeling examples from [1, 41, 43].

Figure 3.4: Examples of different degrees of ornamental complexity: (a) and (b), both conceived by rule-based approaches; (c) and (d), both modeled following a scripting-based system (explored in the coming sections).

Despite research efforts and analysis of various other studies [42, 44, 45], we have not found any additional studies entailing the generation of 3D models, via rule-based solutions, that encompass highly intricate ornamental details. This, together with the previously stated observations, suggests that resorting to *CGA* for modeling purposes proves burdensome and insufficient when complex ornamentation is required. This has to do with limitations inherent to this approach, such as (1) the recurrent event of *SG* rule rewriting, (2) the complexity that such an activity entails, (3) the unwanted impacts that *SGs* might have on later modeling stages, and (4) the lack of important, and much required, geometric features in

CGA's vocabulary. Furthermore, to our understanding, rule-based approaches are best suited for subdivisions, derivations, and automatic generation purposes (through systems such as *CGA*) when preconceived models, stemming from much more flexible and complete shape modeling approaches, are guaranteed. Finally, since most rule-based approaches resort to VPLs as an interface for design, with these being much less flexible and extensible than TPLs—as previously mentioned in Section 2.2.1—, it is then necessary to consider additional approaches to tackle the modeling requirements of our study.

3.3 Parametric-Based Solutions

The forthcoming subsections present extensions and practical applications of PD procedures as referred in Section 2.1. These procedures can be found within a myriad of approaches and are heavily resorted to by scripting-based solutions. Furthermore, they constitute the building blocks that allow for design variations/combinations of the utmost importance—as these establish a bridge between the combinatorial and expansive benefits of computational procedures and design exploration methodologies. Section 3.3.1 introduces a new procedure that expands upon conventional PD techniques and Section 3.3.2 showcases practical applications of PD, together with a short reflection on the beneficial aspects of such practices when applied to use cases that strictly align with our case study.

3.3.1 Design Procedures

Isolating ornamental and structural definitions is no trivial task; such is further reinforced when faced with examples where the mutual dependency between structural and ornamental factors is so complex that one can not simply isolate them. One such example is that of the architectural works present in Gaudí's *Sagrada Família*, where shape configurations are considered ornaments themselves; thus bringing a higher degree of complexity when encapsulating ornamental assets from their structural counterparts. As stated by Barrios [46], it is nonetheless plausible to model such shape configurations following architectural ideas that are easily reproducible via computational approaches: through shape parametrization and the definition of a set of procedures that act upon those very parameters (e.g., rotations, dimension variations, and *Boolean* operations for shape combination); thus going in line with PD procedures, as stated in Section 2.1.

Despite the multiple applications allowed by conventional PD approaches, Hernandez [16] highlights their topological and geometrical limitations when applied to modeling procedures aimed at incorporating complex shapes. The author then proposes a new methodology—Design Procedures (DPs)—aimed at expanding upon these already-established conventions and bringing much greater expressiveness to 3D modeling where parametric variability is of the utmost necessity as an exploratory measure in design. This proposal differs from these PD approaches by distancing itself from the variation and com-

bination of parameters, to then focus on the mutual inclusion of shape construction and parametrization of schemata. The author exemplifies this distinction by referring to the PV schema of a parallelepiped—composed of parameters such as length, width, and height—where variations can be applied. However, such will in no way be able to transform its original cubic composition to that of a cylindrical one. It is then, in situations such as these, when the author's DPs proposal comes in handy as a means of overcoming these limitations—by including additional attributes to the shape's parametric schema, namely its shape configuration; thus allowing a designer to change the initial shape along the design process.

By resorting to the DPs approach, the author was capable of appropriately describing the complex shapes found in Gaudí's works by performing different shape conjugations and subsequently changing their configuration throughout successive design steps—instead of generating new parametric models from scratch—; hence deviating from the static shape configurations of conventional PD methods. To illustrate DPs' capabilities, we have Figure 3.5.

(a) A path delineating a modeling trajectory.

(b) Possible derivation via compositions along path (a).

Source: DPs examples from [16].

Figure 3.5: DP when applied throughout a given path, as presented in (a), following shape compositions at different trajectory stages; thus resulting in a final model, as depicted in (b).

Here, the modeling procedure is guided by four superimposed shapes, followed by extrusions along the vertical axis and a conjugation of *Boolean* operations to attain the final shape—this further simplifies modeling processes and deviates from applying Gaudí's original and more laborious procedure (resorting to a single shape as a way of obtaining the output showcased in Figure 3.5b).

This methodology proves significant, as it encompasses specific cases where ornament is defined via complex shape conjugations. In addition, it provides us with an approach where shape generation is not limited to conventional PD; consequently allowing for broader parametric definitions, including dynamic attributes that can be changed throughout various design steps to attain a final shape.

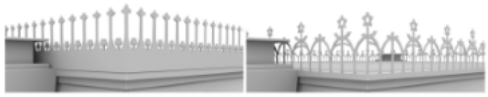
3.3.2 Parametric Solutions in Practice

Kramer and Akleman [47] present an approach to perform different variations in design following architectural rules of the American Second Empire style. Despite solely relying on PCs and PVs, this approach proves nonetheless relevant, as it mentions a strategy to capture differences in ornamental style, followed by their parametrization.

In order to develop procedural methods to describe an architectural style, the authors state that one must first identify the patterns responsible for the materialization of styles. However, with style being often coupled with structure, it is equally as important to identify the structural features that effectively incorporate ornamental ones.

Several categories were identified in this study, the main ones being houses and roofs, followed by their corresponding features, namely doors, windows, cornices, and crestings. By identifying these categories and subdividing them into their different features, the authors organize the concepts in a hierarchical tree structure that can individually be referred to as a means of incorporating different styles; thus resulting in a multitude of design combinations. Figure 3.6 showcases these approaches when applied to gables and crestings.

(a) Different gable styles after applying PD procedures.



(b) Different cresting styles after applying PD procedures.

Source: PD examples from [47].

Figure 3.6: PD in practice: a possible combination of PV and PC results in different stylistic outputs whilst maintaining structural characteristics unchanged.

This study gives insight into the importance of proper identification and corresponding modularization of architectural assets into different sections and features; consequently serving as additional reinforcement on the significance of PD approaches to our study. On that note, and having explored important procedures that can incorporate a multitude of variations in design/style, the coming sections will delve into specific frameworks that allow for the application and implementation of these procedures. These frameworks further incorporate all necessary geometric operations to create algorithmic definitions of architectural design (without which PD procedures would not be possible).

3.4 Scripting-Based Solutions

As opposed to the previously presented rule-based approaches, CD solutions that make use of scripting-based approaches fall within the realm of TPLs and are often dependent on either an already-established programming language or a DSL that was purposefully crafted for modeling purposes. In either case,

the modeling of shapes depends on function declaration—to expand upon basic geometric functions and perform shape compositions—and invocation—for geometric operations, instantiations, and to serve the aforementioned function declarations.

The coming subsections explore scripting-based solutions comprising characteristics matching those which were previously introduced in Section 2.2.2. Furthermore, due to their breadth and better suitability in tackling the requirements of our study, these represent the essence of our solution; thus Section 3.4.1 presents *Generative Modeling Language (GML)*, focusing on important aspects concerning its uses, it also explores low-level requirements for 3D modeling, and explains how these can be transcended by higher-level abstractions; Section 3.4.2 introduces *Khepri*, highlighting its flexible and scalable characteristics, and delineates crucial aspects of greater advantage to those of *GML*. Finally, as previously done in the previous sections, Section 3.4 draws a reflection on the benefits, drawbacks, and trade-offs of scripting-based approaches and further clarifies why these outweigh their rule-based counterparts with regard to our solution.

3.4.1 *Generative Modeling Language*

Havemann [1] developed an approach that makes use of programming paradigms for the modeling of 3D shapes: a stack-based DSL named *GML*. Here, shapes are represented via function/procedure definitions and combinations, just like one would declare and invoke a function in a programming environment, followed by a convenient interpretation through a pipeline where each shape's high-level representation is converted to its low-level counterpart. *GML* gives prevalence to the definition of shapes via programming functions and does so as a means of combining two beneficial factors to modeling: (1) low-level representation abstraction for user-friendly purposes, and (2) parametric definitions, through a function's arguments, such that one is capable of changing a given shape's design by simply modifying its parameters. This, in turn, brings much greater expressiveness to modeling, since programming approaches allow passing functions as arguments (i.e., higher-order functions) and recursive propagation—the former enabling optimized design conceptualization and the latter being an essential practice commonly found in architectural settings.

GML further circumvents common hindrances associated with traditional Computer Graphics (CG) practices, as described by the authors:

Modeling Bottlenecks: Lack of automation and high costs associated with 3D modeling are two of the main impeding factors to much more fluid workflows. Another aspect mentioned in this study is the almost unique nature of a 3D model after several modeling operations, in the sense that performing the slightest change to earlier modeling steps is quite convoluted and comes with undesirable side effects—the invalidation of many latest operations;

Model File Sizes: Traditional 3D graphics files heavily depend on their representation method (i.e.,

triangles, point clouds, among others), most of which occupy a lot of space. Approaches to reduce a model's file size by following compression methods (e.g., progressive meshes) exist. However, despite their lossless quality, they still depend on the order of the input mesh. Additionally, given that most of these require post-processing for decompression, another issue arises—as stated by the authors, the relationship between the model's mesh and corresponding modeling history is lost;

Convoluted 3D Digital Libraries: When resorting to 3D digital libraries, the previous issues are further accentuated, as one is now dealing with a combination of concerns on a wider scale; thus, these libraries' efficiency depends on the 3D models contained therein, with file sizes and representation methods being the most impacting features—a simple triangular mesh representation leads to shape matching problems due to lack of information regarding its representation. Searching for a given model in a repository would then pose significant challenges under these conditions.

As mentioned by the author, overcoming the previously named limitations requires careful consideration of a model's characteristics. Following such an observation, the author conforms to storing strictly necessary information about a shape's representation. For that effect, *GML* distances itself from typical triangular representations—despite allowing such means—and gives preference to the parametric nature of 3D models as a baseline for shape generation. This reduces modeling bottlenecks, as operations are strictly written in the form of sequential procedures, allowing for changes to be performed with much ease; it equally reduces model file sizes, as these are stored in American Standard Code for Information Interchange (ASCII) files; and it allows for better querying in digital libraries due to their parametric nature (giving these objects semantic characteristics).

Being a scripting-based modeling approach, *GML* follows the principles of an interpreted programming language, aims at simplifying modeling processes, and promotes their corresponding extensibility; hence conforming to an API-like tool to modeling. The author resorts to an implementation strategy standing on a combination of different techniques that, when stacked together, constitute a pipeline—whose different layers seamlessly integrate—to realize their intended solution. More specifically, this combination entails *Catmull/Clark subdivision* surfaces [48], *Combined Boundary Representations (CB-Reps)* [49], *Euler* operations [50], and a stack-based DSL (*GML*). On that note, *GML* allows for high-level shape descriptions that are interpreted through such a pipeline down to lower-level representations established through these combinations; therefore enabling the derivation of a myriad of complex-shaped 3D models through much simpler means. For this interpretation to occur, the author designed *GML* alongside additional libraries targeted at recurrent geometric operations. Therefore, *GML's* functionality is structured as follows:

Core Library: Contains basic language-wise operations for stack-based manipulation, flow control, data structures, among others;

Geometry Library: As the name implies, comprises a wide range of functions that perform geomet-

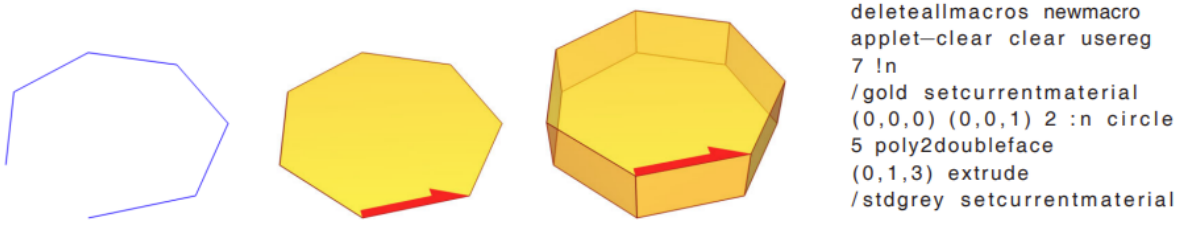
rical operations;

***CB-Rep* Library:** A library that contains all necessary components for low-level shape representations (i.e., *CB-Reps, Catmull/Clark subdivisions*, and *Euler* operations);

Modeling Library: Encapsulates high-level modeling operations (e.g., extrusion, among others) that make use of the *CB-Rep* and Geometry libraries for shape generation purposes;

Interaction Library: Constitutes functionalities capable of handling user input.

To implement these libraries, the author resorted to *C++*[3] and designed *GML's* syntax based on that of *Adobe's PostScript*,[4] which, in our opinion, constitutes *GML's* main limitation—observations and analysis on this syntax of choice will be derived in the coming sections. Finally, to better illustrate *GML's* application, Figure 3.7 showcases the generation of a simple heptagon followed by its generative procedures. One can easily assert that the code snippet responsible for the heptagon's generation is much less perceptible when compared to other languages; thus reinforcing the previously mentioned limitation. Additionally, the previous Figure 3.4c and Figure 3.4d in Section 3.2 were equally generated via *GML*, these being particularly ornate in design and proving relevant to our study. Consequently, these will be further mentioned and explored in ensuing chapters.



Source: Polygon generation example from [1].

Figure 3.7: Polygon generation via *GML* by interpreting the rightmost code snippet.

3.4.2 *Khepri*

A different approach where the modeling of shapes is achieved via programming procedures is *Khepri*.[5] *Khepri* drastically differs in breadth and syntax intelligibility when compared to *GML*—*Khepri* being much broader and intelligible (two useful qualities that may contribute to overcoming *GML's* limitations).

Khepri is an AD tool, of TPL nature, that allows for the definition of geometric shapes through algorithmic descriptions that later propagate to a backend environment of choice [51, 52]. Examples of backend environments supported by *Khepri* include *AutoCAD*, *Rhinoceros*,[6] *Revit*,[7] among others—these

[3] https://isocpp.org/
[4] https://www.adobe.com/products/postscript.html
[5] https://algorithmicdesign.github.io/tools.html
[6] https://www.rhino3d.com/
[7] https://www.autodesk.com/products/revit/architecture

being commonly used in architectural and 3D modeling practices. These algorithmic descriptions are accomplished via *Julia*:[8] a modern programming language—commonly used in scientific computing—known for its smooth learning curve, fast execution, and large-scale development support [53]. Just like any other TPL, generative shape descriptions in *Khepri* are performed by sequentially invoking its predefined functions encompassing a wide range of geometric operations similar, but not limited, to those found in *GML*. Furthermore, the result of these operations can be visualized in real-time due to their automatic propagation to the previously mentioned backend environments; thus transcending a common hindrance of rule-based approaches as previously identified in Section 3.1.2.

Khepri also offers all of the required low-level operations for extensive, complex, and complete geometric modeling, since it works as an interface that relays instructions to a backend environment that already supports them. Naturally, with all previous factors taken into consideration, *Khepri* offers a set of advantages worth considering in our study:

Versatility and Portability: Two particularly useful qualities when one wishes to generate equivalent 3D models—across different backends—stemming from a single set of algorithmic procedures;

Flexibility and Scalability: Two inherent properties of most TPLs that propagate to *Khepri* due to its reliance on *Julia*;

Smooth Learning Curve and Syntactic Intelligibility: Two factors that are preserved by *Khepri* thanks to *Julia's* characteristics.

To further illustrate syntactic intelligibility, when compared to that of *GML*, and to exemplify a simple modeling case via *Khepri*, Figure 3.8 showcases how the modeling procedures depicted in Figure 3.7 can be made simpler.

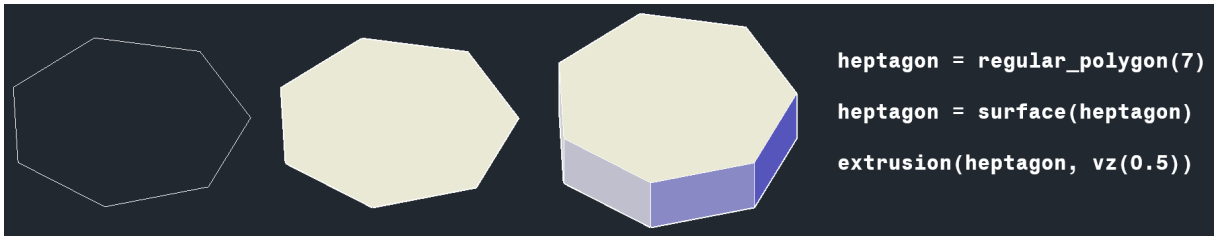


Figure 3.8: Polygon generation via *Khepri* by interpreting the rightmost code snippet (this being much simpler than the snippet found in Figure 3.7).

3.5 A Reflection on Scripting-Based Solutions

Taking into consideration the programmatic nature of scripting-based approaches and the benefits they entail, these ultimately surpass most of the limitations of their rule-based counterparts (as traced in

[8] https://julialang.org/

Section 3.2). Scripting-based approaches show much more promising prospects when flexibility and scalability are concerned, with special emphasis on allowing for the generation of highly complex 3D models in such a way that would simply be unfeasible by resorting to rule-based solutions without greatly sacrificing intelligibility and ease-of-editing. Naturally enough, these two qualities are ultimately coupled with the user's capability of organizing their workflow in an orderly fashion, together with the complexity of the framework in use—with the latter playing a crucial role in optimizing the former. Furthermore, as previously discussed in Section 2.2.2, one of the benefits of procedural techniques is the possibility of applying a single algorithmic description such that repetitive forms can be easily modeled without additional effort—such can be easily accomplished via scripting-based approaches in a much more controllable manner than that of rule-based ones.

As mentioned in Krispel et al. survey [20], despite the benefits of this approach, it also comes with certain disadvantages and thresholds that must be accounted for: the need to learn programming languages and paradigms in order to use and apply these techniques. This is especially noteworthy when considering that most 3D modelers and architects lack Computer Science curricula skills. However, on a positive note, what ultimately brings weight to this factor is the choice of language and the complexities it entails—as stated by the authors, such can be easily reassured by the success of *Processing* as a development environment for non-computer scientists or users that lack programming backgrounds [54]. The authors also refer to the success of *Processing* as being related to two specific factors: the simplicity of the language's syntax and the interactive experience that it provides; thus contributing to an environment that fosters the development of 3D models based upon visual feedback (a quality that *Khepri* supports).

Provided the Gothic Architecture modeling examples that we will delve upon in the coming sections, the previously mentioned hindrances—and corresponding ways of transcending them—suggest that maintaining the powerful and extensible core features of solutions such as *GML*, while applying them through simpler frameworks (*Khepri*), inspires a fruitful research path. Consequently, this would allow for modeling users to have access to powerful, and much more accessible modeling environments, combined with the possibility of complex 3D modeling encompassing structural, ornamental, and stylistic variations. This, in turn, accommodates the required degree of accessibility and interactive experience for professionals in the 3D modeling domain, and aligns with our research goals.

Going back to rule-based approaches, it has been made clear that the combination of *L-systems* and *SGs* is particularly useful when modeling urban environments. The main advantage of these approaches pertains to their expansive and automatic generation qualities. However, their limitations pose additional challenges when modeling highly complex assets: unwanted modeling side effects, scalability issues, and limited potential when modeling highly complex architectural structures. Furthermore, one can observe that most of these resort to stacking a combination of alternate *SGs* for the sake of overcoming their inherent limitations; thus suggesting a not-so-effective alternative, since it inevitably raises equally

complex side effects as its vocabularies increase. Fortunately enough, scripting-based approaches constitute an alternative with much more flexible and effective means to 3D modeling that surpass these limitations. Consequently, as this thesis' scope intends to include the modeling of highly ornate assets, it is wiser to consider scripting-based alternatives and regard rule-based ones only as an hypothetical extension. The difference in modeling complexity attained via rule-based approaches compared to those derived through scripting-based means, as previously illustrated in Figure 3.4 of Section 3.2, further reinforces this decision.

4

Solution

Contents

As previously indicated in Chapter 1, the solution presented herein aims to implement a methodology capable of addressing optimized modeling processes that allow for structural and ornamental variations of Gothic Architecture. It is, however, important to establish a definition of what constitutes a structure and an ornament. Consequently, in our study, we define structural representations as being the elements of a given design that pertain to its topological and sustaining characteristics, as opposed to ornamental representations that, following our definition, are the geometric elements responsible for embellishments and aesthetic purposes. The following examples represent two possible ways of partitioning structural and ornamental features in Gothic Architecture:

Gothic Cathedral: The most basic topological and sustaining features of a Gothic cathedral are none other than its pillars and walls (making these their structural constituents). These constituents can then include additional features for embellishment purposes, such as ornate shapes and forms,

for the pillars; and arcs, windows, and rosettes, for the walls (making these their ornamental elements).

Gothic Window: One can delve deeper into the ornamental features of sub-hierarchical elements of a cathedral and find additional structural and ornamental features. For instance, a Gothic window's topological and sustaining components would be its tracery (i.e., the boundary that delimits a window's surface) and the positioning of its constituents in a *Cartesian* space, whilst its ornamental characteristics would be the window's rosette embellishments, the shape that gives a 3D aspect to its tracery, among others.

It is then clear that both structural and ornamental representations are dependent on one another; thus, the idea that we wish to implement is to create algorithmic descriptions capable of generalizing structural representations, concerning cathedral floor plans, that can later embody a wide range of ornamental assets (e.g., arches, windows, and corresponding constituents), which allow for simple parametric modifications—with minimal modeling effort and little to no impact on the model's core structure—that contribute to different stylistic outputs. More specifically, we wish for our solution to simplify and conform to a designer's requirements when struck with ideas that concern stylistic variations; therefore streamlining and reducing their workload—one possible way of solving the previously identified limitation in Chapter 1. In other words, our solution should be capable of effectively addressing the research questions that were previously presented in Chapter 1.

Having explored useful solutions, in Chapter 3, capable of addressing both simple and convoluted modeling procedures—with most of them inheriting computational benefits—and having established a rationale that justifies our solution's trajectory (Section 3.2 and Section 3.5), Figure 4.1 provides an illustrated bird's-eye view on how one would resort to our methodology as a means of defining AD/CD procedures that conform to the previously mentioned use cases.

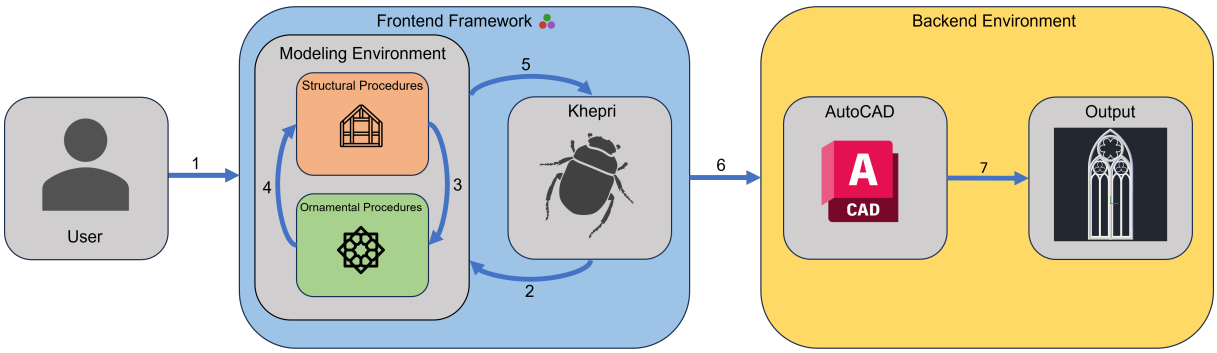


Figure 4.1: An overview of our solution's architecture.

Reinforcing on the previously devised reflections in Section 3.2 and Section 3.5, our solution will heavily resort to scripting-based approaches, with *Khepri* serving as our AD tool, due to greater flexibility, greater scalability, and more powerful modeling leveraging when compared to rule-based solutions.

Moreover, the use of a TPL solution as a means of avoiding poor scalability and complexity overheads relating to VPLs played an equally crucial role in determining scripting-based approaches as an adequate trajectory when compared to rule-based ones—the latter being commonly associated with VPLs, as opposed to the former. Finally, we will be making use of *Julia's* programming paradigms, aligned with *Khepri*, to establish optimized data structures befitting of PD conventions; thus efficiently meeting the requirements of the previously mentioned procedures and allowing for variations and combinations in Gothic styles/designs.

4.1 A Brief Overview of Our Solution

This section delves deeper into the workflow presented in Figure 4.1 and, most importantly, provides a closer inspection of the implementation strategies that made this solution possible. Consequently, it highlights modeling strategies and examples found in studies presented in Section 3.4.1 and traces an optimized alternative to *GML* by implementing and extending its hypotheses via *Khepri*.

As a brief guide to better understand how the implementation strategies of ensuing subsections fit within our solution's architecture, the following numbered list will expand on the homologous numbered arrows in Figure 4.1:

1. User and frontend framework interaction for modeling purposes (code editor of choice);
2. Functionality bundle comprised of geometric/modeling procedures (*Khepri* as an AD tool);
3. Structural algorithmic representations (transferred to ornaments);
4. Ornamental algorithmic representations (transferred to structures);
5. Interpretation of combined algorithmic representations (by *Khepri*);
6. Transferring and translation to low-level representations of the backend environment (by *Khepri*);
7. Output materialization/visual feedback (supported backend environment of choice).

Expanding on these definitions, it is worth mentioning that the user's interaction with the frontend framework is to be performed by resorting to the *Julia* programming language, with *Khepri* as an external AD tool, as a means of modeling and performing variations/combinations on structural and ornamental representations. Additionally, while the backend environment depicted in Figure 4.1 is *AutoCAD*—the reason being that it was the backend tool to which we resorted for testing purposes—, the idea is for the algorithmic procedures defined in the modeling environment to translate to multiple backends; thus making use of *Khepri's* fundamental versatility and portability purposes, as mentioned in Section 3.4.2, and broadening the scope of our solution.

With *Khepri* making use of *Julia*, and *GML* being an adaptation of *PostScript*, it follows that, to implement our solution, either we resort to language interoperation mechanisms—as a means of translating *GML* code to *Julia*—, as explored in Ventura's thesis [55], or we adapt *GML's* modeling strategies, to those of *Khepri*, by adjusting them to *Julia's* programming paradigms. Since we wish to conceive a solution for modeling purposes, the latter seems a much more plausible and desirable trajectory due to simplification and optimization benefits overcoming the syntactic overheads allied to *GML*, rather than resorting back to *GML's* complicated syntax only for it to be translated by an interface connecting to *Khepri*. Therefore, if one wishes to develop an AD tool capable of reducing workloads and allowing for greater modeling expressiveness, to the point of performing stylistic variations in convoluted designs, preserving the aforementioned benefits is of the utmost importance.

Following the arguments above, Section 4.2 expands on the geometric and parametric conventions of Gothic Architecture (windows and cathedrals); Section 4.3 further elaborates *GML* as an AD tool, while emphasizing its limitations; and Section 4.4 devises a rationale on switching from *GML* to *Khepri*, as an optimized AD tool alternative, followed by demonstrations on how we implemented our solution resorting to the latter.

4.2 Parametric Representations of Gothic Architecture

4.2.1 Parametric Representation of a Gothic Window

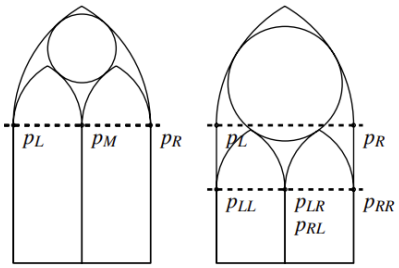
For the parametric representation of a Gothic window, we will follow Havemann and Fellner's analyses [1, 56]. The authors refer to an important modeling factor that will be heavily applied in the coming sections: despite being complex shapes, all forms found within a Gothic window are but mere compositions of circle segments (arcs) and straight lines obtained via a set of modeling operations (e.g., intersections, offsets, and extrusions). Furthermore, they emphasize that, in order to define AD/CD representations of Gothic windows, one must first perform two fundamental steps in design: analysis (identification of construction operations) and synthesis (definition of corresponding parametric representations via an AD tool). Concerning these requirements, the authors consider a default Gothic window as being composed of the following constituents:

Pointed Arc: Standing on top of the window, the pointed arc (main arc) results from an intersection between two arcs;

Sub-Windows: Standing side-by-side, below the main arc, the sub-windows are also composed of pointed arcs (inner arcs) and may, or may not, contain inner constituents;

Rosette: Filling the gap between the main arc and the two inner arcs, we have a rosette that may, or may not, contain inner constituents (a rosette's shape is often a circle tangential to the main arc and the right and left arcs of the left and right inner arcs, respectively).

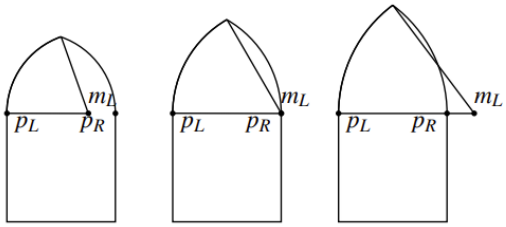
Borrowing an illustration from the authors' studies, Figure 4.2 showcases these components.



Source: Gothic window prototype from [1].

Figure 4.2: A Gothic window prototype composed of a main window and its constituents: two sub-windows and one simple circular rosette.

To model a pointed arc, the authors consider intersecting two symmetric arcs. Consequently, as previously indicated in Section 2.1, one can devise multiple parametric schemata. In the particular case of Gothic geometry, we will consider the following parametric representation: (1) the arc's center/midpoint, (2) a radius, (3) a start point, and (4) an endpoint (the last two parameters following the counterclockwise convention).[1] Furthermore, when circles as a whole are concerned, we will merely consider two parameters: midpoint and radius. All that is left is to apply this representation to constructing a pointed arc.



Source: Pointed arc parameters from [1].

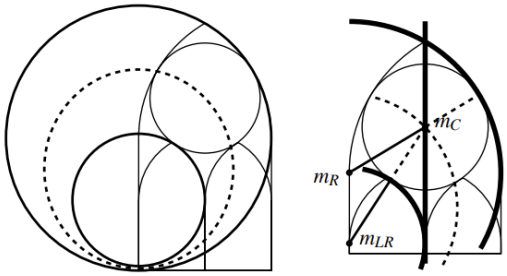
Figure 4.3: A possible parametric representation of a Gothic window's pointed arc.

Following the diagram in Figure 4.3 and taking the main arc's left arc as an example, the previously defined parametric representation of an arc is applied as follows: the midpoint corresponds to m_L, the radius equals the distance between p_L and m_L, the start point corresponds to the pointed arc's intersection point (derived via trigonometric analysis or *Boolean* operations), and the endpoint matches p_L.[2] Once this is done, one applies these modeling principles to the pointed arc's right arc, combines it with the left arc, and with a parametric rectangle of choice; thus effectively modeling the window as

[1] To exemplify a possible geometric function signature: `arc(center, radius, startpoint, endpoint)`
[2] To exemplify a possible invocation: `left_arc = arc(`m_L`, dist(`p_L`, `m_L`), intersection_point, `p_L`)`

a whole. Furthermore, the radius—contributing to the pointedness of the arc—is determined by an additional parameter corresponding to the arc's *excess*, with $excess = radius/distance(p_L, p_R)$; thus, $radius = distance(p_L, p_R) \cdot excess$.

For the sub-windows, it is as simple as applying the previous procedures to new spatial considerations (horizontal reduction by half and vertical offset); hence, we have one new parameter to consider: the vertical distance between the main arc's baseline and the corresponding baseline of the inner arc's (the horizontal reduction should not be parameterized since it is meant to remain static).



Source: Calculating a rosette's parameters from [1].

Figure 4.4: A set of intersections to calculate a Gothic window's rosette parameters.

For the rosette, we are left with a geometrical problem that requires abiding by the previously mentioned tangential requirements. The authors conveniently defined a strategy to do so: for the midpoint, perform an intersection between the vertical axis of symmetry and an ellipsis whose foci are m_R and m_{LR}, as defined in Figure 4.4; for the radius, subtract one of the main arc's arc radius by the distance between that arc's midpoint and the rosette's midpoint.

Finally, to define a planar border that gives surface to the window, the authors consider two additional parameters: the inner and outer offsets (associated with the distance between inner fields and the outer boundary of the window, respectively). To demonstrate these offsets in practice, Figure 4.5 showcases these operations when applied to window structures similar to the ones that were previously illustrated.

Resorting to all the previously defined parameters, it is then possible to model a Gothic window's structure and apply variations in design by controlling its parameters. Therefore, to summarize what has been presented so far, the authors defined the following parameters:

`excess`: The pointedness of the arcs;

`arcDown`: The vertical distance between the main and inner arcs;

`bdInner`: The inner offset;

`bdOuter`: The outer offset;

`wallSetback`: The setback distance between the window and the wall to which it is inserted;

`heightBott`: The vertical distance between the ground and the window's bottom line;

(a) $inner_{offset} = outer_{offset}$ **(b)** $outer_{offset} = 2 \cdot inner_{offset}$ **(c)** $inner_{offset} = 2 \cdot outer_{offset}$

Source: Offset operations in Gothic windows from [1].

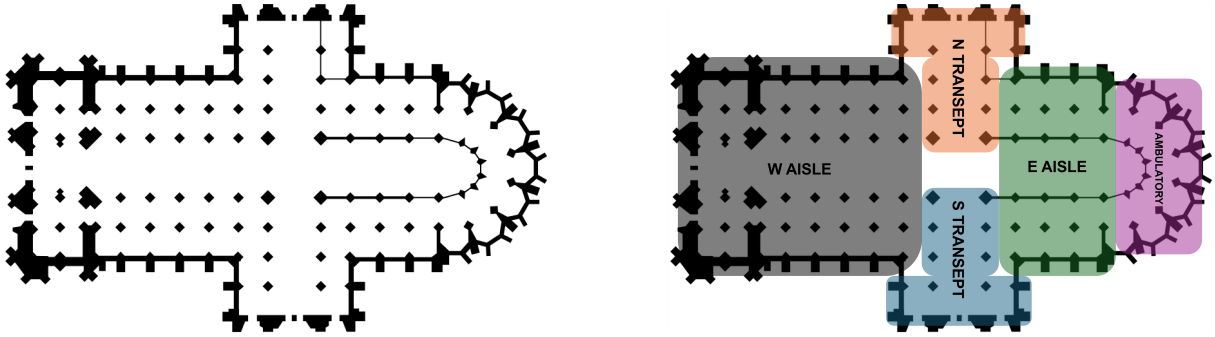
Figure 4.5: Three Gothic window surfaces when inner and outer offsets, with different values, are applied.

`kseg`: The accuracy control for circle segments;

`Style`: An additional parameter responsible for the ornamental qualities of the window (explored in the coming sections).

4.2.2 Parametric Representation of a Gothic Cathedral

When it comes to defining the parametric representation of a Gothic cathedral, this follows a much simpler process than that pertaining to Gothic windows, since we merely resort to its floor plan[3] as a basis for parametrization. Nevertheless, for simplification purposes, we partition a cathedral's floor plan into a simplified and generic version of traditional architectural conventions—preserving, however, its expressive capabilities and allowing for variations in design capable of adapting to different Gothic floor plans. On that note, Figure 4.6 highlights a generic floor plan, where each section, apart from the ambulatory, is prefixed by a direction in space—this will later serve as a useful mechanism to dynamically delineate the number and type of pillars that constitute a cathedral's floor plan.



(a) Cathedral floor plan. **(b)** Cathedral with labeled floor plan.

Figure 4.6: A cathedral's floor plan (a) side-by-side with a labeled version (b).[4]

[3] https://en.wikipedia.org/wiki/Cathedral_floorplan
[4] Original floor plan from https://commons.wikimedia.org/wiki/Category:Plans_of_Cologne_Cathedral

4.3 Modeling a Gothic Window with *Generative Modeling Language*

This section will mainly explore practical modeling examples targeting the parametric representations found in Section 4.2.1, via *GML*, while highlighting its limitations to better understand how *Khepri* can mitigate them.

Firstly, before delving into the *GML* scripts that materialize a Gothic window, we will first emphasize an additional factor, of great importance, that pertains to adding styles to a window. As we have previously observed, the simple structural definition of the prototypical Gothic window is delimited by the main window (main arch), two sub-windows (inner arches), and a rosette. These assets can further be refined by resorting to additional functions such that every constituent incorporates different design elements (e.g., a rosette can contain inner foils, of varied nature, and the main and inner arches 3D profiles can be derived out of different shapes). To that end, Havemann and Fellner define structural and ornamental definitions of Gothic windows in isolation.

4.3.1 Algorithmic Procedures for Ornamental Modeling

When defining the ornamentation of a Gothic window, the authors apply the same parametrization principles as in Section 4.2.1. However, this new implementation follows a different approach: the structural definition is contained within a single block of code, comprised of distinct generative functions (for each element), while the ornamental definitions are constrained to an external dictionary which can then be transferred to the previous code block. This, in turn, constrains the window's structural representation to a static form, while allowing for a dynamic procedure when its ornamental characteristics are concerned; thus, the window can encompass a wide variety of styles—since these are defined externally from, and later transferred to, the structural model.

Borrowing a code snippet, with minor modifications, from Havemann's study [1], Listing 4.1 depicts how one would implement an external style dictionary—responsible for the modeling of ornamental assets—in isolation from the code that generates a window's structure.

```
1 Gothic_Window.Styles.Style_1 begin
2     { usereg !edgeBack !edgeWall
3               !wallSetback !bdOuter !poly
4       /stdGrey setcurrentmaterial
5       :poly 5 poly2doubleface dup edgemate
6       :edgeWall killFmakeRH
7       :bdOuter 4 div :wallSetback neg 5 vector3
8       extrude
```

```
9       } /style_main_arch exch def
10      { usereg !edgeBack !edgeWall !poly
11        /stdGreen setcurrentmaterial
12        :poly 5 poly2doubleface !edge
13        /gold setcurrentmaterial
14        :edge edgemate :edgeWall killFmakeRH
15        [ :edge ] [ ( 0.05,0.3,1) ]
16        extrudestable pop
17      } /style_fillet exch def
18      { usereg !rad !mid !edgeBack !edgeWall !poly
19        /stdRed setcurrentmaterial
20        :poly 5 poly2doubleface !edge
21        :edge edgemate :edgeWall killFmakeRH
22        /gold setcurrentmaterial
23        :edge (0.05,0.3,5) extrude !edge
24      } /style_rosette exch def
25      { usereg !bh !arcL !edgeBack !edgeWall !poly
26        /stdBlue setcurrentmaterial
27        :poly 5 poly2doubleface !edge
28        :edge edgemate :edgeWall killFmakeRH
29        /gold setcurrentmaterial
30        :edge (0.05,0.3,5) extrude !edge
31      } /style_sub_arch exch def
32  end
```

Listing 4.1: Defining a style dictionary for ornamental features of a Gothic window with *GML* (Original from [1]).

Referring to the code snippet above, the authors define a dictionary `(Style_1)` containing four generative functions:

`style_main_arch`: Creates the polygonal face that gives a 3D quality to the main arch's surface border;

`style_fillet`: Decorates the four fillets as delimited by the window's constituents (i.e., the holes between these constituents);

`style_rosette`: Responsible for the decoration of the main arch's rosette;

`style_sub_arch`: Performs similar operations to those found in `style_main_arch`, but applies them to the inner arches.

Due to generic similarities in the operations performed among the different code blocks, and for simplification purposes, we will mainly focus on the definition of `style_main_arch` (lines 2-9 of Listing 4.1):

`lines 2-3:` Loads a list of arguments and stores them (exclamation mark followed by argument);

`line 4:` Sets modeling operations to come `(setcurrentmaterial)` to make use of a specified element `(stdGrey)`;

`line 5:` Generates a surface `(poly2doubleface)` from a polygon `(poly)`, duplicates its output (half-edge) on top of the stack `(dup)`, and creates an opposite half-edge `(edgemate)`;

`line 6:` Applies an *Euler* operator `(killFmakeRH)` to cut a hole into the wall `(edgeWall)` such that it contains the models to come;

`line 7-8:` Creates a 3D vector `(vector3)`, from which an extrusion path is defined, and applies the extrusion `(extrude)` making use of the two values on top of the *operand stack*: the extrusion path and the surface defined in `line 5`;

`line 9:` Defines a function `(style_main_arch)`, switches its position with all the values on the *operand stack* `(exch)`, and declares it `(def)`.

4.3.2 Algorithmic Procedures for Structural Modeling and Ornamental Transferring

Having defined the style dictionary in Section 4.3.1, it is then passed as an argument to yet another dictionary (`line 9` of Listing 4.2). However, this new dictionary holds a different purpose: it is responsible for storing the necessary parametric values (matching those that were previously presented in Section 4.2.1) to be passed onto the generative function that makes up the structure of a window (defined in Listing 4.3).

```
1  dict dup begin !windowdict
2  /excess 1.25 def
3  /arcDown 2. 0 def
4  /bdInner 0.4 def
5  /bdOuter 0.5 def
6  /wallSetback 0.1 def
7  /heightBott 2.0 def
8  /kseg 6 def
9  /Style Gothic_Window.Styles.Style_1 def
10 end
```

Listing 4.2: Defining the parametric values of a Gothic window with *GML* (Original from [1]).

```
usereg !windowdict
!pBaseR !pBaseL !edgeBack !edgeWall
:edgeWall facenormal !nrml

:windowdict begin Style begin

%%% DECORATE MAIN ARCH
:pBaseL :pBaseR excess 0.0 :nrml
gw_pointed_arch pop
heightBott :nrml kseg
gw_polygon_2arcs_height
bdOuter wallSetback :edgeWall :edgeBack
style_main_arch !edgeArch

%%% COMPUTE SUB_ARCS AND ROSETTE
:nrml wallSetback neg mul dup
:pBaseL add !pL
:pBaseR add !pR

:pL :pR :nrml
gw_compute_arcs_rosette
!rosetteRad !rosetteMid
!arcRR !arcRL !arcLR !arcLL

%%% COMPUTE AND DECORATE THE FOUR FILLETS
:arcLL :arcLR :arcRL :arcRR
:rosetteMid :rosetteRad :pL :pR :nrml
gw_compute_fillets
gw_polygon_fillets 4 array
{ :edgeArch :edgeBack style_fillet }
forall

%%% DECORATE THE ROSETTE
:rosetteMid :nrml :rosetteRad kseg 4 mul
circle
:edgeArch :edgeBack :rosetteMid :rosetteRad
style_rosette

```

```
39 %%% DECORATE THE TWO SUB_ARCHES
40 heightBott bdOuter add !heightInnerArc
41
42 :arcRL :arcRR :heightInnerArc :nrml kseg
43 gw_polygon_2arcs_height
44 :edgeArch :edgeBack :arcRL
45 style_sub_arch
46
47 :arcLL :arcLR :heightInnerArc :nrml kseg
48 gw_polygon_2arcs_height
49 :edgeArch :edgeBack :arcLL
50 style_sub_arch
51
52 end end
```

Listing 4.3: Modeling the structure of a Gothic window and applying ornamental features by loading an external dictionary, for stylistic purposes, with *GML* (Original from [1]).

Regarding the listing above, we can see that the authors conveniently segmented the code into different sections (as delimited by the comments therein): (1) computation and decoration of the main arch, (2) computation of the sub arches and rosette, (3) computation and decoration of the four fillets, and (4) decoration of the two sub arches. Consequently, when structural computations are concerned, the authors invoke previously defined functions for modeling purposes, while the decoration (ornaments) of these corresponding elements is achieved by loading the entries of the parametric dictionary (see Listing 4.2) followed by an additional loading of the style dictionary (see Listing 4.1) that receives the parameters of the structure—doing so allows for the values contained within these dictionaries to become available in the current environment.

Similarly to what we have done in previous sections—to avoid redundancy and for simplification purposes—we will delve deeper into a select code block of Listing 4.3:

`line 5:` Loads the Gothic window's structural parameters, together with the ornamental dictionary (as stored in the style parameter);

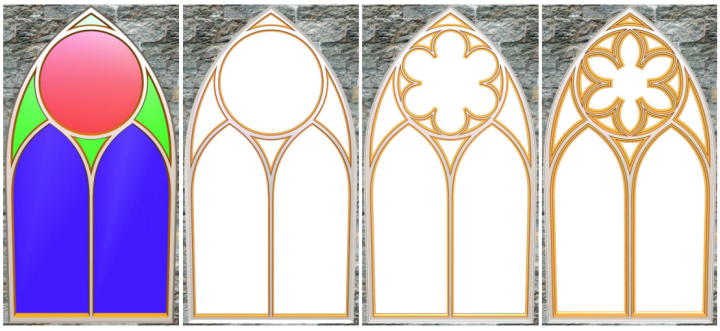
`lines 8-9:` Executes a function that generates the main arc `(gw_pointed_arch)` and discards `(pop)` the last value from the list of values outputted by this function;

`lines 10-11:` Executes a function `(gw_polygon_2arcs_height)` that generates a polygonal model (out of the previous main arc) and that provides additional parameters pertaining to that shape;

`lines 12-13:` Transfers the values in the *operand stack* (from the previous two definitions) to the style generator `(style_main_arch)` and stores the resulting structural and ornamental combination

```
(!edgeArch).
```

Naturally enough, the principles above can be applied to all other blocks of code with the necessary modifications to accommodate different modeling requirements. Nevertheless, the important feature to be highlighted is how one can resort to externally defined data structures as a means of transferring ornamental features to a structure: Figure 4.7 showcases a set of varied Gothic styles from the application of different style dictionaries to the same structure (the leftmost figure being the result of interpreting the combination of Listing 4.1, Listing 4.2, and Listing 4.3)



Source: Gothic window style derivations [1].

Figure 4.7: A set of varied Gothic styles obtained by applying different ornament dictionaries to a static structural definition of a Gothic window.

To summarize the processes shown so far, and to highlight the programming methodologies serving as a basis to structure our solution, the scripting-based approaches that we observed provide the necessary means to tackle convoluted modeling processes that allow for the isolation of structural and ornamental assets, such that the latter can be incorporated by the former without constituting additional overheads pertaining to additional shape modifications; thus allowing for the inclusion of rich ornamentation in virtual environments through means that are capable enough of reducing a 3D modeler's workload, while fostering architectural expressiveness and variations in design—a possible strategy, that we aim to optimize, capable of mitigating a previously stated limitation in Chapter 1.

4.4 Implementation

4.4.1 Switching From *Generative Modeling Language* to *Khepri*

Despite being powerful in the concepts it accomplishes, to our understanding, *GML's* approach is far from optimized—especially when its usability by designers with no programming experience is concerned. *GML's* syntax heavily inherits *PostScript's* postfix notation, making it less accessible to these users; limits a user's workflow, due to its stack-based paradigm; and requires additional measures to

ensure proper use of the *operand stack* and consequent procedural dependencies (i.e., one must accommodate to applying sequential functions in such a way that the required parameters' order matches that of those found in the *operand stack*). Furthermore, the modeling operations supported by *GML* usually make use of low-level modeling concepts, examples of which include those identified in Listing 4.1: direct manipulation of half-edges[5] and *Euler* operators.

Taking these factors into consideration, we strive to develop a modeling solution that inherits the concepts made possible by *GML*—isolating structural and ornamental representations—, such that one can devise a structural asset to later embody a wide set of varied styles. However, such requires optimizing this approach to bring it one step closer to better usability, and syntactic intelligibility, to designers with little to no prior programming experience. On that note, and referring back to our reflection in Section 3.5, we will resort to *Khepri's* inheritance of *Julia* as an effort to circumvent *GML's* limitations. Consequently, this would allow us to provide the following contributions: (1) abstracting the recurrent stack-based operations from the user's end, making these processes much more accessible; (2) promoting a broader understanding of how to operate with a scripting-based approach, by switching from *PostScript's* postfix notation to *Julia's* infix notation (commonly used in mathematical contexts familiar to architects); (3) abstracting away low-level modeling operations through the employment of user-friendly modeling functions (closer to those commonly used within the Architectural/Design domains); and (4) overcoming the outdated and rigid programming syntax of *PostScript*, originally conceived for controlling printing devices, by resorting to *Julia's* friendlier syntax and much more flexible application scope. To illustrate how *Khepri* can mitigate these limitations, Listing 4.4 showcases a *GML* function to model a pointed arc, and Listing 4.5 showcases a *Khepri* function serving the same purpose and using the same naming conventions.

```
1  { usereg !nrml !offset !excess !pR !pL
2    :pR :pL :excess line_2pt !mR
3    :pL :pR :excess line_2pt !mL
4    :pL :pR dist :excess mul :offset sub !rad
5
6    :mL :rad :mR :rad :nrml
7    intersect_circles pop !qT
8
9    [ :qT :mL :pL :pR :offset move_2pt ]
10   [ :pR :pL :offset move_2pt :mR :qT ] :rad
11 } \gw_pointed_arch exch def
```

[5] https://en.wikipedia.org/wiki/Doubly_connected_edge_list

Listing 4.4: Declaring a function that returns the parametric representation of a pointed arc with *GML* (Original from [1]).

```
1 function gw_pointed_arch(pL, pR, excess, offset, nrml)
2     mR = line_2pt(pR, pL, excess)
3     mL = line_2pt(pL, pR, excess)
4     rad = (dist(pL, pR) * excess) - offset
5     qT = intersect_circles(mL, rad, mR, rad, nrml)[end]
6     ([qT, mL, move_2pt(pL, pR, offset)],
7      [move_2pt(pR, pL, offset), mR, qT],
8      rad)
9 end
```

Listing 4.5: Declaring a function that returns the parametric representation of a pointed arc with *Khepri*

Examining the listings above, it becomes clear how *Khepri* can replicate the same procedures as those of *GML*, but following a much more compact, intelligible, and accessible fashion. Taking a closer look at specific fields further reinforces this observation:

Function definition and arguments: `line 1 and 11` of Listing 4.4, albeit separated, are intrinsically linked. When invoking this function (defined in `line 11`), its arguments `(line 1)` are retrieved from the *operand stack*—additional measures for syntactic legibility and stack-based conformance are required `(exch)`. This process can be made simpler by the syntax found in `line 1` of Listing 4.5, suppressing any stack-based awareness requirements and further simplifying its comprehensible qualities.

Function invocation: Comparing `line 4` of Listing 4.4 to `line 4` of Listing 4.5, it is clear how *Julia's* syntax provides a much clearer understanding of the order and flow of function invocation and arithmetic operations. Firstly, function invocation in *Julia* is clearly defined by visual boundaries (parenthesis); secondly, assigning values to a variable is better understood when replicating mathematical notation (i.e., prefix declaration with `'='`, instead of postfix declaration with `'!'`); and, lastly, the infix notation of arithmetic operations is much more familiar to a common user than that of postfix notation (i.e., the latter may contribute to overall confusion).

Stack-based paradigm avoidance: Invoking the function in `line 7` of Listing 4.4, much like the other procedures, requires previously computed values to abide by an order matching that of the arguments of this function. This limits the flexibility of the framework, in the sense that the user can no longer apply procedures in whichever order they desire—something that is mitigated by *Julia*. Fur-

thermore, we also have an additional stack operator (`pop`) to solely extract the last output returned by the function. However, despite employing a similar strategy in `line 5` of Listing 4.5, one can resort to keyword arguments to foster the expressive qualities of these processes—instead of returning a list where one must be aware of its order, one can return a list of named values that can later be referred to, individually, by their names; thus increasing the expressiveness of the framework.

4.4.2 Implementing Structural and Ornamental Representations with *Khepri*

Having explained how one can isolate structural and ornamental concepts resorting to fruitful programming paradigms—such that a structure is fixed and its ornamental qualities dynamic (allowing for the incorporation of various stylistic outputs in a single unmodified structure)—and having explored the limitations allied to *GML's* approach and how these can be mitigated with alternative solutions, we will now describe how we implemented these methodologies, resorting to *Khepri* as an AD tool, applied to the modeling of Gothic windows and cathedrals.

4.4.2.A Structural Representations of Gothic Windows

For the structure of a Gothic window, we consider a slightly modified version of the parametric representations found in Section 4.2.1. These parameters are the responsible factors that delineate the 2D shapes/boundaries of a window. Consequently, we have defined a set of procedures for their definition and retrieval. With regard to structural parametric definitions, we resort to a generalized *Julia struct* as defined in Listing 4.6.

```
1  mutable struct GothicWindow
2      bottom_left_corner
3      upper_right_corner
4      excess
5      vertical_distance_to_sub_arch
6      outer_offset
7      inner_offset
8      window_profile
9      rosette_style
10     left_sub_arch_style
11     right_sub_arch_style
12 end
```

Listing 4.6: Gothic window parametrization via a *Julia struct*.

The *struct* above is composed of structural and ornamental parameters. However, following the purpose of this section, we will merely explore the structural details therein. As the naming convention implies, to situate the window in a 3D space, we indicate its bottom left and upper right corners—in the form of *Khepri's* 3D point data type—, followed by spatial information concerning its constituents: excess, vertical distance to sub arches, and outer/inner offset (akin to those found in Section 4.2.1 of floating point and integer data types). This data structure will later be passed as an argument to a modeling function that outputs the window's 3D model.

Lastly, when structural parametric retrievals are concerned, we refer to additional functions later invoked, whenever necessary, within the aforementioned modeling function. These functions either return 2D shape data types, as defined by *Khepri*, or *Julia* number data types—both regarding the constituents of a Gothic window. Listing 4.7 showcases the required function definitions constituting these structural requirements.[6]

```
# == GOTHIC WINDOW STRUCTURAL GEOMETRY == #
# == ARCHES == #
# == BODIES == #
function get_left_sub_arch_body(...)
function get_right_sub_arch_body(...)
# == BODIES == #
# == TOPS == #
function get_offset_excess(...)
function get_arch_top_height(...)
function lancet_arch_top(...)
# == TOPS == #
# == ARCHES == #
# == ROSETTES == #
function compute_rosette(...)
# == ROSETTES == #
# == GOTHIC WINDOW STRUCTURAL GEOMETRY == #
```

Listing 4.7: Gothic window structural retrieval functions.

Following the order of the functions above, this listing is comprised of: (1) *getters* for the window's sub-arches, these being the sub-arches corner points for positioning purposes; (2) a *getter* for the sub-arches excess, that differs from the main arch; (3) a *getter* for the arches top portion height (the pointed arc's height); (4) a function that returns 2D arc *Khepri* representation data types for the left and right arcs

[6] A complete version of these functions, together with each ensuing listing, can be found in this project's public repository https://github.com/loumourao/ornamental_modeling_khepri

of the pointed arc; and (5) a function that returns all relevant parametric features of a Gothic window's rosette (its center and radius). All these definitions are then referred to, for intermediate calculations, within the function that generates the complete 3D model of a window. Finally, since the structure of a Gothic window is a fixed asset—whose design variations are only achieved by assigning different values to its parameters—, these are meant to remain static (as opposed to ornamental geometries).

4.4.2.B Ornamental Representations of Gothic Windows

Having explained our strategy when the implementation of structural representations of Gothic windows, via *Khepri*, is concerned, we will now explore the ornamental aspects to be incorporated by these structures.

First and foremost, with the parametric definition of a window being closely related to its structure (of 2D nature), we are now left with two additional steps pertaining to ornamentation: (1) develop a collection of generative functions that further refines/embellishes the window's structure and corresponding constituents, and (2) provide a set of operations that gives a 3D quality to these shapes' layout. However, in this section, we will merely cover the latter, showcased in Listing 4.8, and reserve the former for Chapter 5.

```
# == GOTHIC WINDOW ORNAMENTAL GEOMETRY == #
# == SOLID ORNAMENTATIONS == #
# == ARCHES == #
function three_dimensionalize_arch_top(left_arc, right_arc,
                                       offset_value, profile)
    half_offset = offset_value / 2
    profile = surface(scale(profile, abs(half_offset) /
                        DEFAULT_PROFILE_RADIUS, u0()))
    left_arc = offset_arc(left_arc, half_offset)
    right_arc = offset_arc(right_arc, half_offset)
    delete_shapes([left_arc, right_arc])
    union(sweep(left_arc, profile), sweep(right_arc, profile))
end

function three_dimensionalize_arch_body(...)
function three_dimensionalize_arch_middle(...)
# == ARCHES == #
# == ROSETTES == #
function three_dimensionalize_rosette(...)
```

```
20 # == ROSETTES == #
21 # == SOLID ORNAMENTATIONS == #
22 # == GOTHIC WINDOW ORNAMENTAL GEOMETRY == #
```

Listing 4.8: Gothic window ornamental 3D materialization.

As we can observe in the listing above, we incorporated 3D qualities in the ornamental section given that the 2D boundaries that define a window's structure can be filled by various shape definitions. Let us take, as an example, the generative function that outputs an arch's pointed arc (`lancet_arch_top` of Listing 4.7). This function outputs its left and right arcs which, in turn, serve as inputs to the `three_dimensionalize_arch_top` function above, together with two additional arguments responsible for a 2D shape (`profile`) to be *swept*[7] along these arcs (paths) and corresponding dimensions (`offset_value`)—the former being circumscribed, and the latter defining the diameter of that circumscription; hence serving as a shape scaling agent to the profile with a default radius. More specifically, this `offset_value` is no more no less than a value delimiting the 2D surface area of the window's structure (as previously shown in Figure 4.5) and represents a crucial parameter for 3D outputs. Let us take a closer look at Listing 4.8 and how to achieve the 3D model of a pointed arc:

`line 6:` A reduction in half to obtain the circumscription radius.

`line 7-8:` Provided a default radius of the profile to be swept, we resort to the `offset_value` as a scaling agent; thus conforming the profile's dimensions to those delineated by the offset.

`line 9-10:` We offset the left and right arcs to obtain the intermediate path sitting between the outer and inner boundaries of the window's surface area; thus allowing for this boundary to be completely filled.

`line 11:` Additional measures to remove 2D drawings.

`line 12:` A *Boolean* union that joins both 3D arc sweeps.

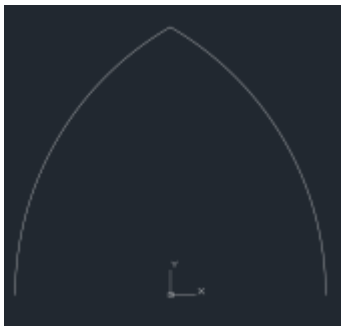
To illustrate this, Figure 4.8 showcases the structural representation of a pointed arc, followed by two different 3D models resulting from two distinct shape profiles; hence emphasizing the ornamental quality of 3D shapes.

Finally, with regard to the ornamental parameters in Listing 4.6, these, following the notions that we just described, serve as simple placeholders for additional information comprising the ornamental factors of the Gothic window:

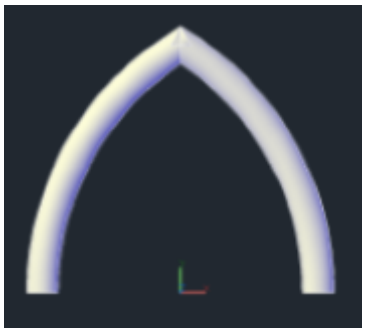
`window_profile:` A parameter, akin to that which we previously described, responsible for the propagation of shape through all entities of the main arch's boundary (i.e., its body and pointed arc).

`rosette_style:` A parameter that receives another *struct* comprised of structural and ornamental

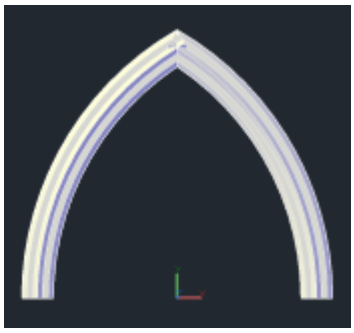
[7] In CAD workflows, a *sweep* is an operation that takes two arguments: a path and a shape. This is a useful operation to create 3D objects conforming to strict rules. In this case, a solid 2D shape can be continuously dragged along that path; thus resulting in a 3D output



(a) 2D pointed arc.



(b) 3D pointed arc with circle profile.



(c) 3D pointed arc with star profile.

Figure 4.8: Two outputs when interpreting the `three_dimensionalize_arch_top` function with different profiles: (a) depicts the structural representation of the pointed arc, (b) portrays a 3D output with a circle profile, and (c) displays an alternative profile derivation.

factors pertaining to the window's rosette (later explained in Chapter 5).

`left_sub_arch_style`: A parameter, of a recursive nature, that assigns an equal *struct* to that which it belongs (i.e., `GothicWindow`). This gives 3D shape to the left sub-arch and may hold different values, for stylistic variations, compared to its constituting main arch.

`right_sub_arch_style`: A parameter serving the same purpose as the parameter above, but aimed at giving shape to the right sub-arch.

Lastly, we defined two initializers that appropriately populate these *structs*, as showcased in Listing 4.9. These can then serve as constructors for Gothic windows and aid the derivation of modeling processes.

```
function Gothic_Window(bottom_left_corner, upper_right_corner,
                       excess, vertical_distance_to_sub_arch,
                       outer_offset, inner_offset, window_profile,
                       rosette_style, left_sub_arch_style,
                       right_sub_arch_style)
    main_arch = GothicWindow(bottom_left_corner, upper_right_corner,
                             excess, vertical_distance_to_sub_arch,
                             outer_offset, inner_offset, window_profile,
                             rosette_style, left_sub_arch_style,
                             right_sub_arch_style)
    gothic_window(main_arch)
end

function Sub_Gothic_Window(window_profile, rosette_style,
                           left_sub_arch_style, right_sub_arch_style)
    GothicWindow(nothing, nothing, nothing, nothing, nothing, nothing,
```

```
17                      window_profile, rosette_style, left_sub_arch_style,
18                      right_sub_arch_style)
19  end
```

Listing 4.9: Gothic window *struct* initializers.

We can see that, as opposed to structural representations, the ornamental characteristics of our implementation are the true dynamic factors that provide distinct styles depending on their different configurations. Consequently, not only can one add different shape configurations for expressive 3D models, but it is equally viable to add generative functions for ornamental refinements to the style parameters of the *struct* (this, together with examples on how to use the constructors in the listing above, will be covered in greater detail in Chapter 5).

Finally, for simplification's sake, the remaining functions in Listing 4.8 are contracted since they perform the same strategy as that which we just described. Furthermore, it is worth mentioning that we follow a slightly different approach than that found in *GML*, given that we apply a *sweep* through all 2D shapes, as opposed to performing surface area (window boundary) extrusions followed by a *sweep* that merely covers the outlines contouring these boundaries. To our understanding, our approach provides more expressive ornamental qualities than those achieved by following *GML's* alternative.

4.4.2.C Combining Structural and Ornamental Representations of Gothic Windows

To model a Gothic window in its entirety, our solution makes use of a single main modeling function `(gothic_window)` that receives as input a conveniently initialized *struct* (as those of Listing 4.6) and executes all necessary modeling steps conforming to auxiliary/intermediate calculations for *Cartesian* positioning of windows (e.g., invocations of functions such as `get_left_sub_arch_body` from Listing 4.7), together with structural (2D) and ornamental (3D) function combinations—akin to those which we previously described. Furthermore, it is worth mentioning that most of these procedures act upon the parametric values extracted from the function's input *struct*.

To describe this function, we refer back to the *structs* in Listing 4.9. It is clear that we have two distinct definitions depending on whether the asset being initialized is a main arch `(Gothic_Window)` or a sub-arch `(Sub_Gothic_Window)`. This goes to show that modeling a Gothic window is based upon a hierarchy comprised of the main arch (parent) and consequent sub-arches that may, or may not, contain additional sub-arches (children); thus forming an object relation that begs for recursive application. To that end, to effectively model a Gothic window, one must first perform a full initialization of the `Gothic_Window` *struct* and refer to the left and right sub-arches with a partial initialization of the `Sub_Gothic_Window` *struct*, as we can observe by the mandatory arguments of the former and the reduced parametric initialization of the latter (with many of its fields being marked as `nothing`). The reason why we designed these *structs*

following a different approach lies in the fact that, for coherent designs, most of the parametric values pertaining to positions in space require adjustments, via intermediate calculations, that conform to adequate positioning of the sub-arches and corresponding constituents. On that note, the `gothic_window` function is inherently recursive, in the sense that it receives a `Gothic_Window` *struct*, comprised of additional `Sub_Gothic_Window` *structs*, whose unassigned parametric values `(nothing)` are then populated with adequate *Cartesian* values by intermediate calculations that, once performed, apply the same procedure following a recursive invocation of the `gothic_window` function to the left and right sub-arches by as many times as necessary—depending on the number of total sub-arches of the window.

Following all the steps described so far, we can then obtain a complete 3D model of a Gothic window by invoking the `gothic_window` function with an appropriately initialized `Gothic_Window` *struct* (due to the lengthy nature of this function, its complete definition can be found in the previously mentioned public repository).

4.4.2.D Structural Representations of Gothic Cathedrals

Much like the previous representations, our approach to defining structural procedures for Gothic cathedrals is based on that found in Havemann's study [1]. However, our solution—in contrast to the static algorithmic description found in the aforementioned study—is appropriately defined in such a way that allows for different Gothic floor plans to be derived by tweaking the generative functions' parametric values. Furthermore, we adopt an additional mechanism to allow for directional axes conformance, across different backend conventions, by defining global variables (whose values' data types follow *Khepri's* vector implementation), as shown in Listing 4.10.

```
# Pillars are stored in an array and oriented S-N (row-wise)
# and W-E (column-wise)
# We will use the following convention in the xy-plane
# (which can later be switched to other planes
# through appropriate transformations)
#       N (y^)
#       |
# W -- O -- E (x>)
#       |
#       S
# where 'O' is the origin
# == WORLD DIRECTIONS == #
NORTH = vy(1)
SOUTH = -NORTH
```

```
WEST = vx(-1)
EAST = -WEST
GROWING_HEIGHT_DIRECTION = vz(1)
DECREASING_HEIGHT_DIRECTION = -GROWING_HEIGHT_DIRECTION
# == WORLD DIRECTIONS == #
```

Listing 4.10: The modeling environment's global variables.

Additionally, as shown in the initial comments of the listing above, when it comes to positioning the pillars of a cathedral in a *Cartesian* plane, we will store them in a matrix whose rows and columns follow south-north and west-east directions, respectively. On that note, we will now expand on how we implemented the structural units composing a cathedral.

The structural factors considered in our study are pillars and walls (each inherently related to one another). With that in mind, as showcased in Listing 4.11, we created two distinct data structures, holding all relevant parametric information that allows for object instantiation, for relationships to be established between each of them, and for ornamental changes in the final model.

```
mutable struct Pillar
    model
    row
    column
    center
    orientation
    width
    depth
    height
    north_wall
    south_wall
    west_wall
    east_wall
end

mutable struct Wall
    model
    left_pillar
    right_pillar
end
```

Listing 4.11: Pillar and wall parametrization via a *Julia* struct.

Taking a closer look at both data structures, we can see that each contains pointers to one another—the last four arguments of `Pillar`, and the last two arguments of `Wall`—, allowing for the previously mentioned relations; they also contain a parameter responsible for the ornamental quality of the asset (`model`); and additional parameters for intermediate calculations (all remaining parameters of `Pillar`). Having established this, it is now possible to refer to these *structs* as effective constructors for the structure of our cathedrals. To that end, we defined three relevant initializers: (1) an initializer concerning measurements (e.g., the distance between pillars) and delimiters (e.g., row and column range for each section of the cathedral), (2) a pillar info initializer, and (3) a wall info initializer. Once these initializers are invoked, they are responsible for storing the previously defined *structs*, together with additional information, in globally accessible dictionaries that serve all required modeling procedures, which are then automatically invoked; thus outputting a Gothic cathedral.

Concerning measurements and delimiters, this is the first dictionary to be initialized, given that the remaining initializers make use of this dictionary's content for proper execution. On that note, the values to be stored in this data structure pertain to (1) a default distance between each pillar, (2) the row and column range of each section within the cathedral, (3) additional values for the ambulatory's center, start angle, and angle increment (since, in contrast to the other sections, this follows a circular fashion where polar coordinates are applied); and (4) values for the indexes of the vertical and horizontal hallway jumps, both present in the middle of the cathedral's aisles and transept sections, respectively (where the equidistant properties between the pillars are no longer maintained).

With regard to the pillar initializer, we defined different pillars for each section of a Gothic cathedral. Succinctly speaking, three distinct pillars were defined: buttresses (pillars forming the exterior outlines of each section), outer pillars (pillars coming right after the buttresses), and inner pillars (all remaining pillars). All these pillars are related to one another in the dimensions that constitute them, in the sense that we defined a basic pillar type—whose parametric values (width, depth, and height) are indicated in the initializer's arguments—that serves as a default placeholder for a ratio-based system that appropriately manages the dimensions of all other pillar types. Furthermore, each pillar type, making use of values in the previously described dictionary, contains additional information for proper rotational placement based on the directional axes of Listing 4.10 or the polar angles of the ambulatory section. This initializer makes further use of the ranges present in the measurements and delimiters dictionary and stores these pillar types, attached to their corresponding sections, in the pillar info dictionary for subsequent modeling procedures. Lastly, with this distinction in pillar type, it is then possible to adapt different ornamental models for each pillar and propagate them in a congruent fashion.

Having explained the pillar initializer, there is not much left to expand on the wall initializer, as it

follows similar processes to those present in the pillar initializer. On that note, walls are equally related to the sections of a cathedral and stored in conjunction with the ranges that compose them within a wall dictionary. Additionally, the major difference in information between these functions is the fact that wall orientations are based on the pillars to which they are connected (i.e., these are established by connecting the centers of each pillar) from which the wall dimensions (width, depth, and height) are equally derived. Furthermore, we also established different types of walls: (1) flying buttresses (albeit not a wall, we consider it as such due to spatial positioning similarities to walls), (2) wall blocks, (3) arch blocks, and (4) window blocks. Finally—akin to the previously described pillar relations—, walls are equally related to one another, in the sense that we resort to wall blocks as an elementary unit to build upon the remaining wall types (flying buttresses being the exception), and, with these distinctions in place, it is also possible to adapt different ornamental models for each wall.

4.4.2.E Ornamental Representations of Gothic Cathedrals

For the ornamental representations of Gothic cathedrals, we follow a slightly different approach than that for Gothic windows: while we merely required a 2D path for *sweeping* purposes, together with recursive *struct* propagations for the ornamentation of windows, we now refer to each pillar and wall (by section) of a cathedral and assign the output of a generative modeling function to the `model` parameter of the *structs* in Listing 4.11. From this point forward, it is then possible to apply different modeling functions to each of the different types of pillars and walls. On that note, the functions that were defined for the ornamentation of pillars are showcased in Listing 4.12, where the previously described basic pillar serves as a default type for all remaining generative functions (contracted for simplification purposes), whose implementation also makes use of the dimensional parameters of the `basic_pillar` function.

```
function basic_pillar(center;
                      width = get_basic_pillar_width(),
                      depth = get_basic_pillar_depth(),
                      height = get_basic_pillar_height())
    half_width = width / 2
    half_depth = depth / 2
    bottom_left_corner = u0() - vxy(half_width, half_depth)
    upper_right_corner = u0() + vxy(half_width, half_depth)
    pillar_shape = surface_rectangle(bottom_left_corner, upper_right_corner)
    pillar_path = line(center, center + vz(height))
    basic_pillar = rotate(sweep(pillar_path, pillar_shape), pi/2, center)
    basic_pillar
end
```

```
14
15 function buttress(...)
16 function aisle_outer_pillar(...)
17 function aisle_inner_pillar(...)
18 function half_outer_crossing_buttress(...)
19 function outer_crossing_buttress(...)
20 function transept_pillar(...)
21 function crossing_pillar(...)
22 function ambulatory_buttress(...)
23 function ambulatory_outer_pillar(...)
24 function ambulatory_inner_pillar(...)
```

Listing 4.12: Pillar ornamental (3D) generative functions.

These notions are equally applied to walls, but with a slight difference concerning the relations among the different types. As previously stated, instead of resorting to dimensional relationships, here we refer to the basic wall block as an elementary unit to build upon much more refined wall types via *Boolean* operations. In other words, if one wishes to model an arch or a wall with a window, we resort to a solid block (the wall block) as the main entity from which *Boolean subtractions, unions, and intersections* are derived (with the remaining shape being either a solid arch, a window, or other kinds of shapes). Listing 4.13 showcases a contracted sub-list of ornamental functions for the different types of walls, together with expanded implementations of basic wall blocks and how one would perform *Boolean* operations in these basic units.

```
1  function standing_wall_block(left_pillar, right_pillar, height; offset = 0)
2      extrusion(wall_block_base(left_pillar, right_pillar, offset),
3                GROWING_HEIGHT_DIRECTION * height)
4  end
5
6  function standing_hollow_wall_block(left_pillar, right_pillar, height,
7                                      hole_height; offset = 0)
8      wall = standing_wall_block(left_pillar, right_pillar, height)
9      hole = standing_wall_block(left_pillar, right_pillar,
10                                hole_height; offset = offset)
11     subtraction(wall, hole)
12 end
13
14 function standing_arch_block(...)
```

```
15 function standing_window_wall_block(...)
```

Listing 4.13: Wall ornamental (3D) generative functions.

4.4.2.F Combining Structural and Ornamental Representations of Gothic Cathedrals

Similarly to the combination of structural and ornamental representations of Gothic windows, we follow a similar strategy to apply these combinations in a Gothic cathedral. However, instead of resorting to a single main function, we resort to two additional functions—one for the pillars and another for the walls—, each responsible for the instantiation of pillars and walls. This instantiation accounts for multiple factors, ranging from *Cartesian* positioning to linking an ornamental representation to its corresponding placeholder within the structural representations of both pillars and walls. Listing 4.14 showcases how these instantiators are defined.

```
1  function instantiate_all_pillars(pillars)
2      instantiate_aisles_buttresses(pillars)
3      instantiate_outer_crossing_buttresses(pillars)
4      instantiate_transept_buttresses(pillars)
5      instantiate_aisle_outer_pillars(pillars)
6      instantiate_aisle_inner_pillars(pillars)
7      instantiate_transept_pillars(pillars)
8      instantiate_crossing_pillars(pillars)
9      instantiate_ambulatory_buttresses(pillars)
10     instantiate_ambulatory_outer_pillars(pillars)
11     instantiate_ambulatory_inner_pillars(pillars)
12 end
13
14 function instantiate_all_walls(pillars)
```

Listing 4.14: Pillar and wall instantiators.

As we can observe, both functions receive as input an argument by the name of `pillars`. This argument is a 2D array, instantiated by the user, in which one specifies the upper range of rows and columns present in the desired cathedral (i.e., the upper range boundary totaling the ranges of each section of the cathedral). By invoking these instantiators, additional invocations concerning the different cathedral sections, to which they belong, are performed—as different sections (defined in the measurements and delimiters global dictionary) conform to different ornamental types. Following this notion, what each of these internal instantiators—found in the `instantiate_all_pillars` function—does is to traverse the

list of pillars, as derived in Section 4.4.2.D, and to populate each entity's parametric variables (except for the walls, which are assigned, at a later stage, by the wall instantiator) depicted in the `Pillar` *struct* of Listing 4.11. Consequently, the three most important parameters therein are (1) the `model`, (2) the `center`, and (3) the `orientation`.

Concerning the `model` variable, this is the sole responsible factor that provides ornamental qualities to a pillar. Therefore, as depicted in Listing 4.15, what the instantiator does is refer to a higher-order function that takes as input (1) the pillar matrix `(pillars)`, to store the end result in the globally accessible matrix; (2) the collection of pillars pertaining to the same section `(pillar_info_collection)`; and (3) a generative function responsible for the desired ornamental models `(pillar_type_instantiator)`.

```
function instantiate_aisles_buttresses(pillars)
    static_row_col_range_pillar_instantiator(pillars,
                                             get_aisles_buttresses(),
                                             buttress)
end

function static_row_static_col_pillar_instantiator(pillars,
                                                    pillar_info_collection,
                                                    pillar_type_instantiator)
    for pillar_info in pillar_info_collection
        row = get_pillars_row_info(pillar_info)
        col = get_pillars_col_info(pillar_info)
        orientation = get_pillars_orientation_info(pillar_info)
        center = get_pillar_coordinates(row, col)
        pillar = pillar_type_instantiator(center, orientation)
        pillar_model = pillar.model
        pillar_width = pillar.width
        pillar_depth = pillar.depth
        pillar_height = pillar.height
        pillars[row, col] = Pillar(pillar_model, row, col, center,
                                   orientation, pillar_width,
                                   pillar_depth, pillar_height)
    end
end
```

Listing 4.15: Dynamic instantiation of a pillar.

Following this implementation, if one wishes for a collection of congruent pillars to embody alternative

ornamental representations, one can simply change the generative function serving as input to the higher-order function of the listing above to a function of their liking.

With regard to the pillar's `center` and `orientation`, as observed in `lines 13` and `14` of Listing 4.15, the instantiator resorts to additional functions responsible for retrieving these values—with the retrieval of the pillar's `center`, shown in Listing 4.16, being much more convoluted than extracting its `orientation` (the latter merely referring back to a previously initialized value in Section 4.4.2.D and being responsible for rotating the pillar's model to fit its orientation).

```
1  function get_pillar_coordinates(row, column)
2      default_pillars_distance = get_default_pillars_distance()
3      horizontal_hallway_jump_prev_row = get_horizontal_hallway_jump_prev_row()
4      vertical_hallway_jump_prev_col = get_vertical_hallway_jump_prev_col()
5      ambulatory_first_column = first(get_ambulatory_col_range())
6
7      x = default_pillars_distance * column
8      y = default_pillars_distance * row
9
10     if row > horizontal_hallway_jump_prev_row
11         y += default_pillars_distance
12     end
13
14     if column > vertical_hallway_jump_prev_col &&
15        column < ambulatory_first_column
16         x += default_pillars_distance
17     elseif column >= ambulatory_first_column
18         get_ambulatory_pillar_coordinates(row, column)
19     else
20         xy(x, y)
21     end
22 end
```

Listing 4.16: A function responsible for retrieving the *Cartesian* coordinates of a given pillar.

As we can observe in the listing above, extracting the *Cartesian* coordinates of a pillar's center is not necessarily trivial. However, this goes to show the importance of delineating the measurements and delimiters described in Section 4.4.2.D. These values are then used for intermediate calculations following a sequence of conditional statements, for structural/positioning awareness purposes, which then conveniently translate the row and column ranges of the pillar to positions in space. To exemplify these

conditional checks, if these statements identify a pillar that is situated in the ambulatory section, they invoke a specialized function that follows an approach based on polar coordinates (since this section is circular; thus being conveniently described by polar coordinates).

Going back to the definitions of Listing 4.14, the `instantiate_all_walls` function is contracted since it follows these very principles (applied to the different types of walls). However, there is an additional factor that links the instantiation of walls to the pillars that support them, in the sense that each wall is instantiated following a left-to-right fashion on each of the sections within the cathedral; thus allowing to link its representation to its corresponding left and right pillars, and to assign a pointer to the last four parameters of the `Pillar` *struct* in Listing 4.11.

On a final note, the reason why we relate pillars to walls, via their parametric *structs*, lies in the fact that if one wishes to change a particular pillar, wall, or multiple combinations of these assets, it can be simply executed by referring to the pillar dictionary (stating its row and column range), switch its model to another, and apply the same procedure to its corresponding walls (e.g., referring to the north wall and changing that wall's model)—all this performed in the final model of a Gothic cathedral.

5

Evaluation

Contents

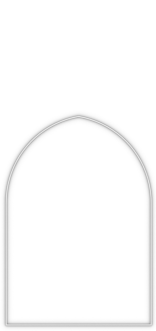
In this chapter, we will evaluate our solution by applying it to the modeling of Gothic windows and cathedrals. Before illustrating these models, we first showcase how to make proper use of our implementation with basic examples, followed by the modeling of more complex examples. On that note, in Section 5.1, we derive multiple Gothic window models—similar to those found in Havemann's study [1]—and showcase further models that incorporate additional ornamental examples to demonstrate our approach's expressiveness. Finally, in Section 5.2, we showcase a series of cathedral models following a generative process that conforms to different cathedral floor plans—via a set of parameters provided by the user—based on real-life examples of Gothic cathedrals found in the European landscape; thus, with this conformance to different floor plans in mind, this evaluation suggests a more dynamic, flexible, and expressive approach, to the modeling of these examples, than that of the previously mentioned study.

5.1 Modeling Gothic Windows

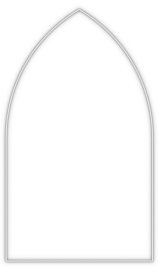
In order to model a Gothic window, one must resort to the constructors found in Listing 4.9. However, further measures are required if one wishes to incorporate all its constituents in a compact and organized manner. Taking this factor into account, we suggest that one implements a specific function that pertains to the overall style of the window, as demonstrated in Listing 5.1: a function that merely outputs a basic Gothic window frame, followed by multiple invocations with different parametric values (showcasing how one can effectively apply design variations in a window's structure by changing its *excess*). To illustrate the outputs of running such a function, we have Figure 5.1.

```
1  function Gothic_Window_Default(bottom_left_corner, upper_right_corner,
2                                 excess, vertical_distance_to_sub_arch,
3                                 outer_offset, inner_offset)
4      profile = surface_circle(u0(), DEFAULT_PROFILE_RADIUS)
5      main_arch = Gothic_Window(bottom_left_corner, upper_right_corner,
6                                excess, vertical_distance_to_sub_arch,
7                                outer_offset, inner_offset, profile,
8                                nothing, nothing, nothing)
9      main_arch
10 end
11
12 Gothic_Window_Default(xy(-50, -42), xy(50, 42), 0.6, 0, 1, 1)
13 Gothic_Window_Default(xy(-50, -42), xy(50, 42), 1, 0, 1, 1)
14 Gothic_Window_Default(xy(-50, -42), xy(50, 42), 2, 0, 1, 1)
```

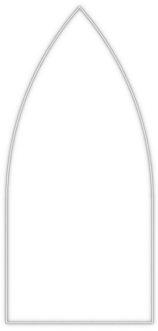
Listing 5.1: Gothic window *struct* initializers.



(a) Output of `line 12` (Listing 5.1).



(b) Output of `line 13` (Listing 5.1).



(c) Output of `line 14` (Listing 5.1).

Figure 5.1: Three Gothic window design variations by adjusting the *excess* parameter: (a) showcases a window with an *excess* of 0.6, (b) an *excess* of 1, and (c) an *excess* of 2.

In addition to illustrating what happens when running these functions, the figure above serves as an example of how one can perform variations in design by changing a single parameter. Naturally enough, incorporating ornamental refinements would allow for a higher degree of stylistic variations. We will now proceed with a brief explanation of how to expand a window's design with ornamental representations.

One important constituent that may encompass a wide range of ornamental embellishments is the rosette. With the main function that produces a Gothic window being responsible for the automatic derivation of its main structure, its sub-arches, its positioning in space, and the derivation of a rosette's parameters (i.e., its center and radius), we are now left with choices pertaining to its ornamental aspects. If one wishes to embellish the rosette, it is necessary to develop an additional *struct*, for initialization purposes, similar to the one found in Listing 4.7—this time focusing on the parametric schema of the rosette.

```
1 mutable struct Rosette
2     foils_instantiator
3     n_foils
4     starting_foil_orientation
5     displacement_ratio
6     rosette_profile
7 end
```

Listing 5.2: Rosette *struct* initializer.

This *struct* will then serve as the data type feeding the `rosette_style` variable of Listing 4.7—whose parameters will later be referred to, by the `gothic_window` function, for modeling purposes. Following this notion, the isolation of structural and ornamental aspects via higher-order functions, as previously demonstrated, requires passing the generative functions responsible for ornamental designs to an additional step that performs the integration of ornament in the structure; hence the parameter `foils_instantiator` in the *struct* above and the inclusion of Listing 5.3 in the code: the first two functions defining two different ways of filling a rosette, and the last code snippet being incorporated in the `gothic_window` function to specify the ornamentation functions, similar to those of Listing 4.8, to be invoked (as determined by the `rosette_style` parameter). Full code descriptions can be found in the previously mentioned public repository.

```
1 function rosette_rounded_foils(...)
2 function rosette_pointed_foils(...)
3
```

```
4  function Empty_Rosette_Style(rosette_profile)
5      Rosette(nothing, nothing, nothing, nothing, rosette_profile)
6  end
7
8  function Rosette_Pointed_Style(n_foils, starting_foil_orientation,
9                                 displacement_ratio, rosette_profile)
10     Rosette(rosette_pointed_foils, n_foils, starting_foil_orientation,
11             displacement_ratio, rosette_profile)
12 end
13
14 function Rosette_Rounded_Style(n_foils, starting_foil_orientation,
15                                 rosette_profile)
16     Rosette(rosette_rounded_foils, n_foils, starting_foil_orientation,
17             nothing, rosette_profile)
18 end
19
20 function gothic_window(window)
21 [...]
22 if rosette_foil_instantiator === rosette_rounded_foils
23 [...]
24 elseif rosette_foil_instantiator === rosette_pointed_foils
25 [...]
26 end
27 [...]
28 end
```

Listing 5.3: Rosette additions in the code.

Following these examples, one can then apply this methodology to as many additional constituents as desired—so long as these are appropriately identified in the window's structural parametrization and corresponding function (e.g., different arch tops other than the pointed arc). It is then possible to bring a higher degree of control to the ornamentation of these structures by (1) implementing *structs* regarding those constituents (Listing 5.2), (2) implementing further constructors for friendlier assignments (`lines 4-18` of Listing 5.3), and (3) adding conditional statements for different ornamental functions in the `gothic_window` function (`lines 22-26` of Listing 5.3).

With the implementations above, it is now possible to apply these constructors, and corresponding ornamental functions, to model a wide variety of Gothic windows. Listing 5.4 showcases how to implement a generative function taking into account all the constituents of a Gothic window.

```
1  function Gothic_Window_First_Style(bottom_left_corner, upper_right_corner,
2                                     excess, vertical_distance_to_sub_arch,
3                                     outer_offset, inner_offset)
4      profile = surface_circle(u0(), DEFAULT_PROFILE_RADIUS)
5      rosette_style = Empty_Rosette_Style(profile)
6      sub_sub_sub_arches_style = Sub_Gothic_Window(profile, nothing,
7                                                   nothing, nothing)
8      sub_sub_arches_style = Sub_Gothic_Window(profile, rosette_style,
9                                               sub_sub_sub_arches_style,
10                                              sub_sub_sub_arches_style)
11     sub_arches_style = Sub_Gothic_Window(profile, rosette_style,
12                                          sub_sub_arches_style,
13                                          sub_sub_arches_style)
14     main_arch = Gothic_Window(bottom_left_corner, upper_right_corner,
15                               excess, vertical_distance_to_sub_arch,
16                               outer_offset, inner_offset, profile,
17                               rosette_style, sub_arches_style,
18                               sub_arches_style)
19     main_arch
20 end
21 Gothic_Window_First_Style(xy(-50, -42), xy(50, 42), 1, 6, 2, 2)
```

Listing 5.4: The generative function of a Gothic window with a basic style.

To understand what the code above performs, we will expand on what each line is responsible for:

`line 4:` To give shape to the window, one must first define a profile. This can be individually applied to the constituents of the window (in this case, we apply a simple circle surface that gives shape to every element of the window).

`line 5:` To specify the rosette of a window, we refer back to the constructors in Listing 5.3 (here, we define an empty rosette).

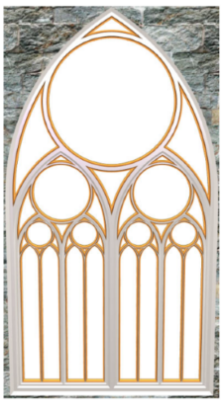
`line 6-7:` This definition must always be applied to the innermost sub-arches. This is due to the fact that no additional ornamentation is required, other than its outer frame; hence the `nothing` arguments for the rosette and sub-arches.

`line 8-13:` Similar to the previous definition, these constructors apply for the modeling of all remaining hierarchical sub-arches. It is, however, necessary to specify the previous to last sub-arch style (as it points to the innermost sub-arch) separately from the rest; hence the distinction between the `sub_sub_arches_style` and `sub_arches_style`. We can see that all sub-arches incorporate the

same rosette style, specific definitions of their corresponding sub-arches, and the profile of `line 4`.

`line 14-19`: This definition is responsible for the modeling of the main arch (the parent node in the hierarchical network of arches) and incorporates all previously established styles accessed throughout the recursive calls of `gothic_window`. The entire model constituting all the definitions herein is then returned.

Figure 5.3a illustrates the output of running the definition above and is followed by additional models with constituents that incorporate a refined set of ornamental representations (whose generative functions can be found in Listing A.1 and Listing A.2, respectively). This figure, compared to Figure 5.2, provides evidence that it is possible to achieve the outputs found in Havemann's study [1] with our approach. Finally, following the definition of the generative functions that pertain to a rosette, it is possible to control a set of parametric features concerning its orientation, the number of foils, and many other factors.



(a) Empty rosette Gothic window.

(b) Rounded rosette Gothic window.

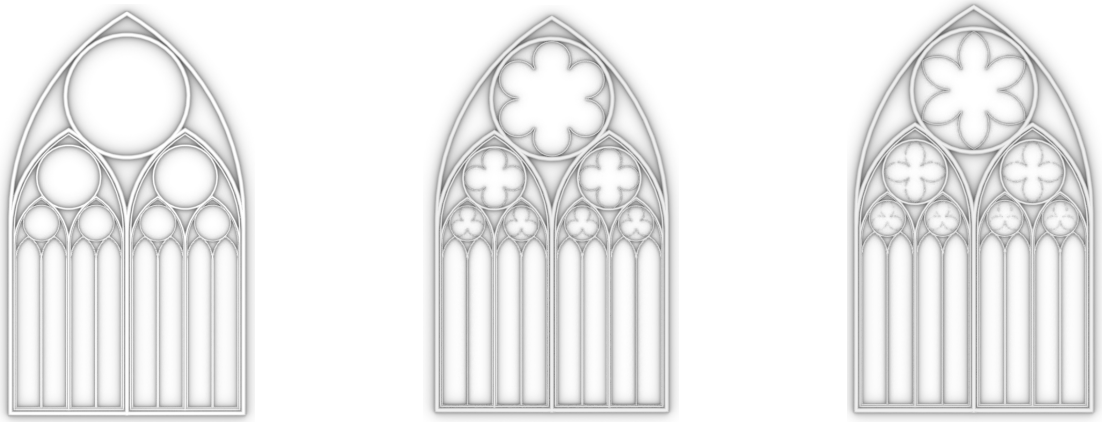
(c) Pointed rosette Gothic window.

Source: Gothic window style derivations [1].

Figure 5.2: Three Gothic window design variations via *GML*.

Lastly, as a final evaluation measure of our approach's expressiveness and level of control, Figure 5.4 illustrates the output of Listing A.3. This provides a modeling example encompassing multiple ornamental representations; thus evidencing that different stylistic elements (e.g., profiles and rosette styles) can be propagated to each constituent in isolation. Furthermore, a zoomed-in view of the left sub-arch, in Figure 5.4b, provides a clear demonstration of the effects that occur when applying different profiles to a shape, particularly in the pointed arc of the left sub-arch (star-shaped profile), its corresponding rosette (circular-shaped profile), and even the main arch's rosette depicted at the top-right corner (a hexagon-shaped profile).

Despite the slight deviation that this level of control may yield on coherent Gothic designs (i.e., applying different styles in each constituent, instead of forcing coherent designs), we have nonetheless preserved this possibility such that one can replicate the expressiveness of our approach when applied



(a) Output of Listing 5.4. **(b)** Output of Listing A.1. **(c)** Output of Listing A.2.

Figure 5.3: Three Gothic window design variations, via *Khepri*, by executing their corresponding functions.

(a) Output of Listing A.3. **(b)** A closer look at the profiles of (a).

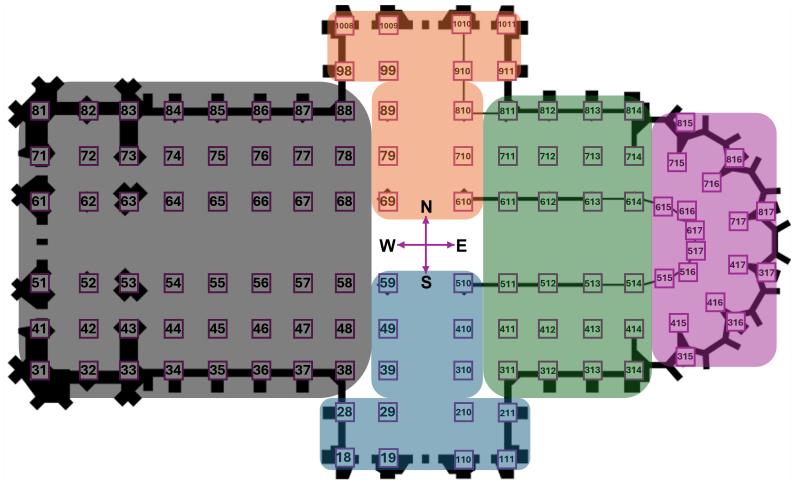
Figure 5.4: A Gothic window with each element encompassing different styles (a) side-by-side with a zoom-in of its profiles (b) (via *Khepri*).

to different case studies that require individual access to the constituents of a model.

5.2 Modeling Gothic Cathedrals

Having defined different Gothic window styles, one can later attach them to a Gothic cathedral—we will first expand on how to generate its structural model. Since our solution allows for the derivation of different floor plans that follow Gothic conventions, all that is left is to specify each section's row and column ranges via the `instantiate_measurements_and_delimiters` function, together with additional operations which we will proceed to explain.

Expanding on the illustration of Figure 4.6, Figure 5.5 provides a clearer explanation of how to delimit



Inspired by a floor plan illustration in [1].

Figure 5.5: Gothic cathedral floor plan: labeled by section and with indexed pillars.[1]

the cathedral's sections by row and column ranges (using the same color codes as those of the previous illustration). As we can observe, the pillars are indexed according to the corresponding row and column index in the pillar matrix, following a south-north and west-east direction, respectively. Looking at this very figure, we can then translate each section's ranges to the following:

West-end aisle: Following the pillars in the grey section, we have 3-8 and 1-8 as ranges for the rows and columns, respectively

East-end aisle: Following the pillars in the green section, we have 3-8 and 11-14 as ranges for the rows and columns, respectively

South-end transept: Following the pillars in the blue section, we have 1-5 and 8-11 as ranges for the rows and columns, respectively

North-end transept: Following the pillars in the orange section, we have 6-10 and 8-11 as ranges for the rows and columns, respectively

Ambulatory: Following the pillars in the purple section, we have 3-8 and 15-17 as ranges for the rows and columns, respectively

All we have to do now is feed these ranges into their corresponding sections and provide structural information for the dimensions of the pillars, as defined in Listing 5.5. In this description, we can find a simple reference to the `instantiate_measurements_and_delimiters` function and the parameters that propagate range information pertaining to the different sections of a cathedral. One additional factor worth mentioning is the `default_pillars_distance` parameter that determines the distance between two pillars—in equidistant sections—and the adaption measures for the vertical and horizontal jumps previously mentioned in Section 4.4.2.D, via intermediate calculations. Finally, following this, we have

[1] Original floor plan from https://commons.wikimedia.org/wiki/Category:Plans_of_Cologne_Cathedral

an equally required function for defining the dimensions for the basic pillars (i.e., width, depth, and height)—all of which, as previously mentioned in the same section, are then adapted to all other pillar types via a ratio-based system. With these instantiators in mind, one can then apply them to generate a cathedral (Listing 5.6 showcases how to use them).

```
function instantiate_measurements_and_delimiters(default_pillars_distance,
                                                  w_aisle_row_range,
                                                  w_aisle_col_range,
                                                  e_aisle_row_range,
                                                  e_aisle_col_range,
                                                  s_transept_row_range,
                                                  s_transept_col_range,
                                                  n_transept_row_range,
                                                  n_transept_col_range,
                                                  ambulatory_row_range,
                                                  ambulatory_col_range)

function instantiate_pillars_info(basic_pillar_width, basic_pillar_depth,
                                  basic_pillar_height)
```

Listing 5.5: Two required instantiators for the structure of a cathedral: the first responsible for all required information pertaining to the different sections of a cathedral, and the last delineating the dimensions of its pillars.

```
# == WINDOW STYLES == #
function Gothic_Window_First_Style(bottom_left_corner, upper_right_corner,
                                   excess, vertical_distance_to_sub_arch,
                                   outer_offset, inner_offset)
    window_profile = surface_circle(u0(), DEFAULT_PROFILE_RADIUS)
    rosette_style = Empty_Rosette_Style(window_profile)
    sub_arches_style = Sub_Gothic_Window(window_profile, nothing, nothing,
                                         nothing)
    main_arch = Gothic_Window(bottom_left_corner, upper_right_corner,
                              excess, vertical_distance_to_sub_arch,
                              outer_offset, inner_offset, window_profile,
                              rosette_style, sub_arches_style,
                              sub_arches_style)
```

```
14      main_arch
15  end
16  # == WINDOW STYLES == #
17  # == DEFAULT CATHEDRAL == #
18  instantiate_measurements_and_delimiters(7.53 * 2,
19                                          3:8, 1:8,
20                                          3:8, 11:14,
21                                          1:5, 9:10,
22                                          6:10, 9:10,
23                                          3:8, 15:17)
24  pillars = Array{Union{Pillar, Nothing}}(nothing, 10, 17)
25  instantiate_pillars_info(1, 1, 105)
26  instantiate_all_pillars(pillars)
27  instantiate_walls_info()
28  instantiate_all_walls(pillars)
29  # == DEFAULT CATHEDRAL == #
```

Listing 5.6: The set of generative procedures to model a Gothic cathedral.

Evaluating the code above generates the model of a Gothic cathedral whose floor plan matches that of Figure 5.5. To better understand these procedures, we will proceed with explaining each step:

`line 2-15:` A simple window style that will be applied to all sections of the cathedral.

`line 18-23:` This comprises the instantiator responsible for setting the structure of a cathedral as defined by the row and column ranges of its different sections. In this example, we can see that we incorporated the previously delineated ranges, according to their sections, together with a distance between pillars of approximately 15 units.

`line 24:` After instantiating the information that pertains to measurements and delimiters, an additional step is required—the instantiation of the matrix of pillars, whose row and column ranges must match the upper bounds of the previously defined ranges. This will serve as storage for the `Pillar` *struct* of Listing 4.11 which is then referred to by the forthcoming functions.

`line 25:` Here, we provide the dimensions of the basic pillars, where their width and depth are equal (1 unit), and their height is 105 units.

`line 26:` This is a simple function that refers to the pillar matrix and is responsible for the propagation of the previously defined information for all internal procedures that generate the pillars of a cathedral.

`line 27:` This function behaves in a similar manner to that of `line 24`. However, it does not require any additional information, since all required calculations are performed by internal procedures

that refer back to the globally accessible data structures generated by the previous functions. It is, nonetheless, still necessary to invoke it.

`line 28:` Similarly to `line 26`, this function refers back to the pillar matrix and is responsible for the propagation of the previously defined information for all internal procedures that generate the walls of a cathedral.

With these definitions in place, executing all functions until `line 26` results in the output of Figure 5.6: a top view of the pillars. As we can see, the positions of the pillars closely resemble the floor plan of Figure 5.5.

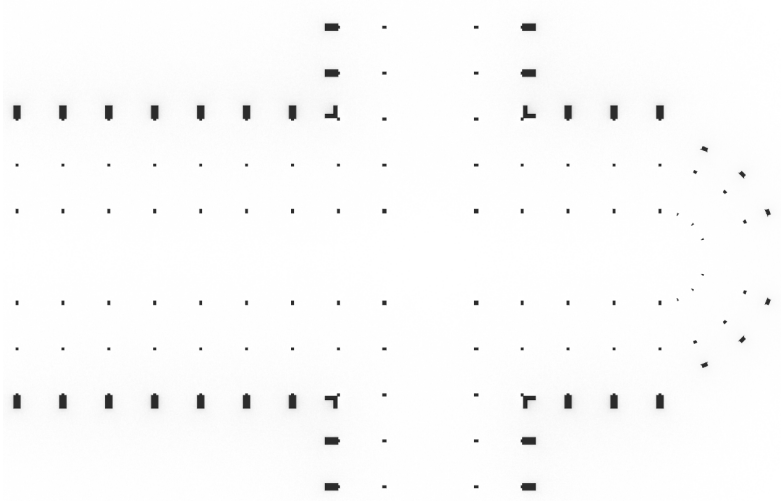


Figure 5.6: Gothic cathedral structure after executing all functions of Listing 5.6 until `line 26`.

Now, executing the remaining lines of code, we obtain the entire model of the Gothic cathedral that we have defined: Figure 5.7 shows the top view of the cathedral, Figure 5.8 illustrates two perspective views from the front and back, and Figure 5.9 showcases slightly angled front and back views of the cathedral.

When the windows of the cathedral are concerned, one must specify their style via the higher-order functions responsible for the generation of walls (i.e., the ornamental representations of the cathedral). Looking at Listing 5.7, we can see the generative function responsible for all window-containing walls of the transept sections. Here, one can specify which previously defined window style should be used by assigning the parameter `window_style_instantiator` with the function responsible for a given window style (in this case, the style defined in Listing 5.6). This same methodology for style propagation can be applied to all other window-containing wall generators (usually the outer walls of the cathedral) and allows for different window styles to be incorporated into the different sections of a cathedral.

```
1 function transept_inner_pillar_vertical_wall(left_pillar, right_pillar;
```

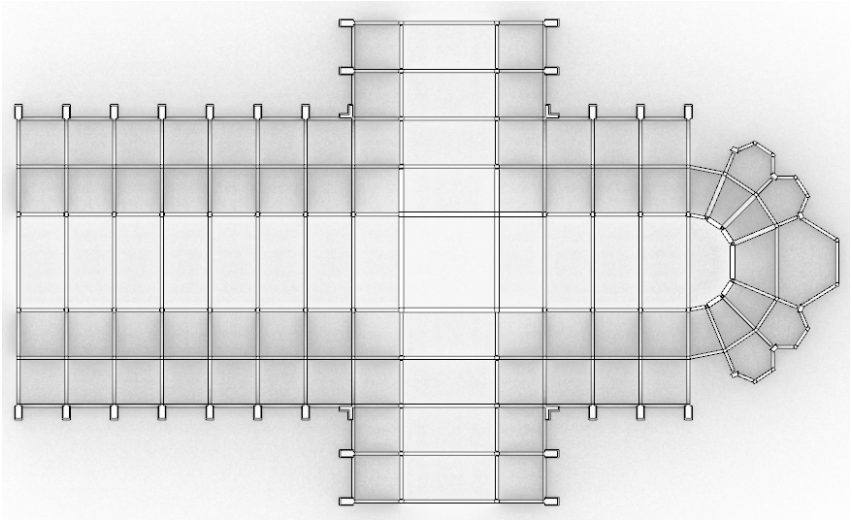


Figure 5.7: Top view of the Gothic cathedral of Listing 5.6.

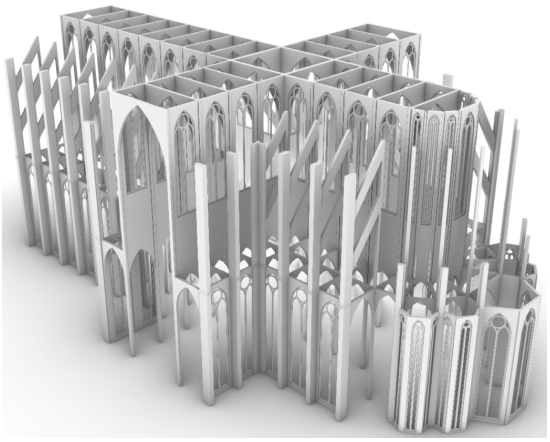
```
                                                    height = ...,
                                                    excess = ...,
                                                    vertical_distance_to_sub_arch =
                                                    ...,
                                                    window_style_instantiator =
                                                    Gothic_Window_First_Style,
                                                    offset = ...)
```

Listing 5.7: The set of generative procedures to model a Gothic cathedral.

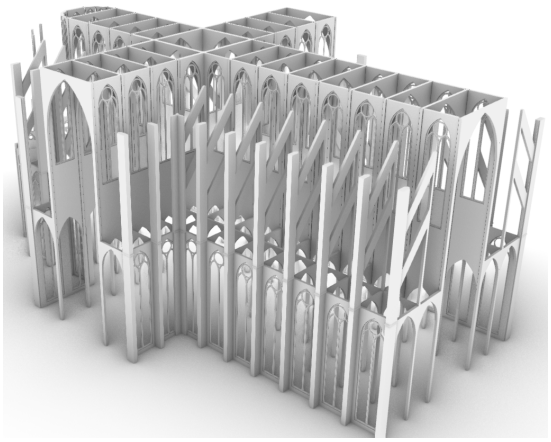
As a final test to evaluate our solution's adaptability to different floor plans, we resort to a real-life example: the *Cathédrale Notre-Dame de Reims*,[2], located in Reims France. Here, we refer to official floor plans, in Figure 5.10a, as a basis for range definition in Listing 5.8 and apply alternate style variations through the different window-containing walls of each section (following the principles of Listing 5.7). Figure 5.14 shows two different views that demonstrate the possibility of applying different window styles throughout the cathedral, as evidenced by the different windows in the walls of the transept (two leftmost windows of Figure 5.14a), the inner aisle (rightmost window of Figure 5.14a), and the outer aisle (all windows of Figure 5.14b).

```
# == WINDOW STYLES == #
function Gothic_Window_First_Style(...)
function Gothic_Window_Second_Style(...)
function Gothic_Window_Third_Style(...)
```

[2] https://en.wikipedia.org/wiki/Reims_Cathedral

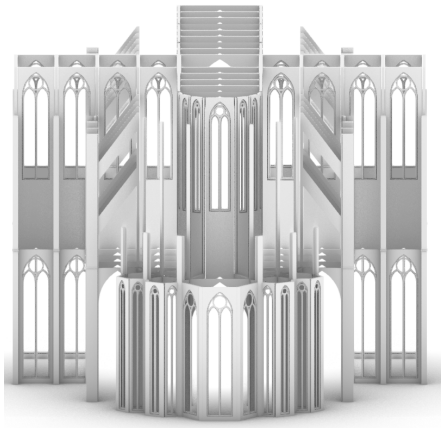


(a) Front perspective view of the Gothic cathedral of Listing 5.6.

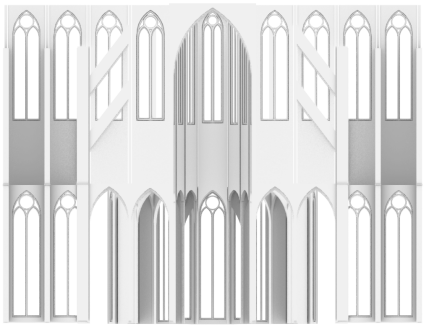


(b) Back perspective view of the Gothic cathedral of Listing 5.6.

Figure 5.8: Two perspective views of the Gothic cathedral of Listing 5.6.



(a) Front view of the Gothic cathedral of Listing 5.6.



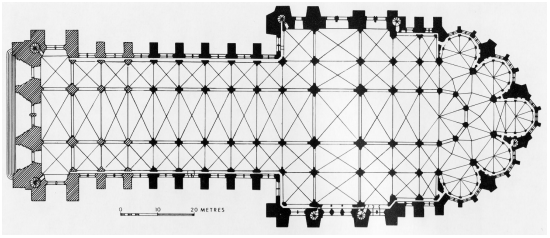
(b) Back view of the Gothic cathedral of Listing 5.6.

Figure 5.9: Two slightly-angled views of the front and back sides of the Gothic cathedral of Listing 5.6.

```
function Gothic_Window_Fourth_Style(...)
# == WINDOW STYLES == #
# == REIMS CATHEDRALE NOTRE-DAME == #
instantiate_measurements_and_delimiters(7.53 * 2,
                                        2:5, 1:9,
                                        1:6, 12:14,
                                        1:3, 10:11,
                                        4:6, 10:11,
                                        1:6, 15:16)
pillars = Array{Union{Pillar, Nothing}}(nothing, 6, 16)
instantiate_pillars_info(1, 1, 105)
instantiate_all_pillars(pillars)
```

```
instantiate_walls_info()
instantiate_all_walls(pillars)
# == REIMS CATHEDRALE NOTRE-DAME == #
```

Listing 5.8: The set of generative procedures to model the Reims cathedral.



(a) Reims cathedral floor plan.[3]

(b) Reims cathedral structure.

Figure 5.10: Side-by-side illustrations of the Reims cathedral floor plan (a) and corresponding structural derivation of Listing 5.8 (b).

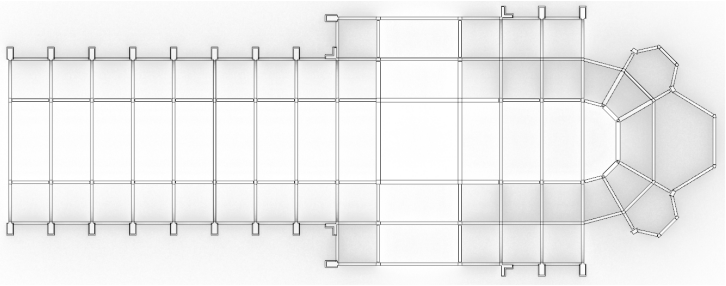
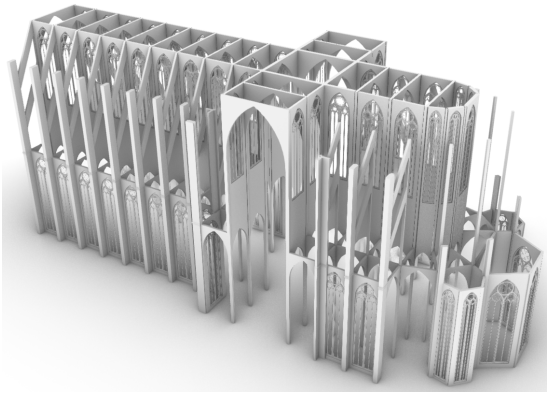


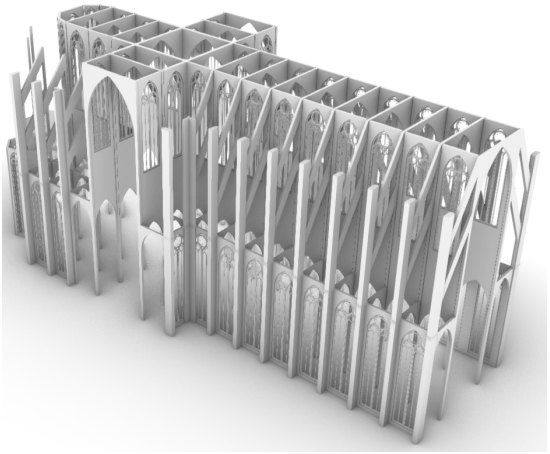
Figure 5.11: Top view of the Reims cathedral model of Listing 5.8.

Having tested our approach to the modeling of multiple Gothic windows and cathedral styles, based on real-life examples, this evaluation suggests the possibility of extending these scripting-based methodologies to different case studies where semi-automatic structural derivations, followed by ornamental integration—with little to no additional modeling efforts—may be desired, so long as one applies all necessary adaption measures to their corresponding modeling requirements; hence effectively reducing the workload associated with these modeling processes. Further examples comprising these qualities can be found in Appendix B, where two renowned Gothic cathedrals—*Cathédrale Notre-Dame d'Amiens* and *Cathédrale Notre-Dame de Paris*—have been derived via their corresponding generative procedures in Appendix A.

[3]Original from https://mcid.mcah.columbia.edu/art-atlas/mapping-gothic/reims-cathedrale-notre-dame

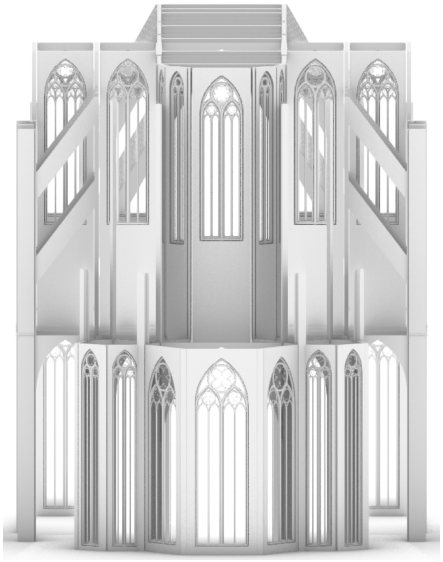


(a) Front perspective view of the Reims cathedral of Listing 5.8.

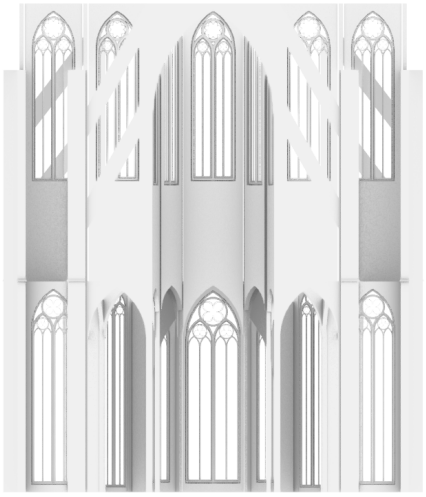


(b) Back perspective view of the Reims cathedral of Listing 5.8.

Figure 5.12: Two perspective views of the Reims cathedral of Listing 5.8.



(a) Front view of the Reims cathedral of Listing 5.8.



(b) Back view of the Reims cathedral of Listing 5.8.

Figure 5.13: Two slightly-angled views of the front (a) and back (b) sides of the Reims cathedral of Listing 5.8.

(a) Upper windows of the Reims cathedral of Listing 5.8.

(b) Lower windows of the Reims cathedral of Listing 5.8.

Figure 5.14: Two views depicting the different styles of the upper (a) and lower (b) windows of the right side of the Reims cathedral of Listing 5.8.

6

Conclusion

Contents

Despite the broad range of 3D operations that conventional CAD software offers, its interactive capabilities prove burdensome, time-consuming, and insufficient when scaling them to complex models. One such example pertains to architectural workflows where style variations in already-defined models are desired: something that requires changes in ornamental details without directly affecting their structural representations. On that note, limitations regarding the ornamental aspects of a shape are predominantly related to the time it takes a CAD user to perform shape refinement processes, mainly due to the complex interdependencies between structural and ornamental representations. Consequently, in order to mitigate this concern, one must facilitate the processes allied to already-established modeling conventions and contribute with appropriate abstractions capable of providing possible solutions that complement these requirements.

Our study addresses the above-stated limitations allied to the inclusion, variation, and combination of structural and ornamental details in built structures. Consequently, we consider convoluted architectural case studies where modeling procedures encompassing these requirements are heavily put to the test. For that purpose, we have chosen Gothic Architecture, and its most complex examples (cathedrals),

as our case study due to its intricate and ornate style; ultimately resulting in a set of requirements that paved the way for our solution, these being (1) provided a standardized set of Gothic floor plan conventions, our solution should be capable of adapting to particular designs (e.g., *Cathedrale Notre-Dame de Paris* or *Basilique Cathedrale Notre-Dame d'Amiens*), (2) provided a fixed set of structural Gothic window representations, our solution must be capable of changing its ornamental style without affecting its structure (e.g., a rosette can be either rounded or pointed foiled), among other requirements.

To tackle these requirements, we identified crucial CD approaches—PD, PM, and AD—, from which modeling processes are employed via algorithmic procedures, instead of direct geometric manipulation. These allow for much more flexible and scalable means of modeling when compared to conventional approaches. Furthermore, these are particularly useful for identifying and defining parameters that describe relations and interdependencies between various objects and are recurrently applied following the generative modeling paradigm: the application of modeling techniques where shape description relies upon a set of sequential operations that, when performed, conceive a shape's final form. On that note, with these three approaches in mind, one can effectively generate a variety of different designs stemming from a model's initial parametric representation; hence conforming to the requirements of our study. From here on in, we expanded upon useful and well-defined methodologies stemming from the approaches above. These methodologies follow rule-based and scripting-based notions and have been applied by various studies as a means of complying with architectural and virtual environment requirements.

Regarding rule-based approaches, we delved upon *CityEngine* and *CGA*: the first being a system that automatically generates virtual cities, and the latter being an extension aimed at avoiding architectural incoherences in the generation of buildings, façades, and windows of the former. Following this, rule-based approaches often resort to VPLs, due to their accessible and user-friendly qualities. However, these carry alongside them a set of disadvantages when modeling scalability and complex architectural projects are concerned: (1) the recurrent event of rule rewriting, (2) the unwanted impacts that these might have on later modeling stages, and (3) the lack of important, and much required, geometric features in the rules' vocabulary. Consequently, we identified these as best suited for subdivisions, derivations, and automatic generation purposes when preconceived models, stemming from much more flexible and complete shape modeling approaches, are guaranteed.

Concerning scripting-based approaches, these resort to TPLs for modeling purposes: much more flexible and extensible than VPLs. As such, these constitute a major advantage to our study's requirements compared to their rule-based counterparts, from which we explored additional studies that ultimately formed the backbone of our solution for the inclusion of complex ornamentation in structural representations. On that note, we explored AD tools such as *GML* and *Khepri*, where modeling procedures are executed based on the application of programming processes. Both these tools serve as

an API-like interface that abstracts away manual modeling processes via the application of functions. When executed, these functions are then converted to low-level modeling operations that output their corresponding model in a visual rendering backend.

To implement our solution, we chose *Khepri* due to its breadth and more perceptible syntax when compared to *GML*. This lies in the fact that *GML* resorts to an outdated, rigid, and convoluted programming language (*PostScript*) that is not friendly towards users with little to no programming background; in contrast to *Khepri*, that resorts to a programming language with much more perceptible syntax and smooth learning curve (*Julia*). Furthermore, *Khepri* constitutes additional benefits that pertain to versatility, portability, flexibility, and scalability: the first two regarding the generation of equivalent 3D models across different backends (stemming from a single set of algorithmic procedures), and the last two being a collateral effect of *Julia's* benefits over *PostScript*.

Following the observations above, we still refer back to *GML* as a useful source for the algorithmic descriptions of Gothic elements, which we then convert to *Khepri*, since these can be made simpler by *Julia*. On that note, our solution comprises algorithmic procedures that encompass an encapsulation strategy to isolate structural and ornamental representations from one another. Examples include (1) the tracery of a Gothic window (structure) and the shape that gives a 3D aspect to this tracery (ornament), and (2) the topological positioning of the walls of a Gothic cathedral (structure) and the elements that embellish them (ornament). For that purpose, we have defined optimized data structures (*structs*) that comprise parametric representations of the constituents of these Gothic elements, which are then referred to, for intermediate calculations, by structural definitions that can later embody ornamental embellishments via their corresponding definitions (isolated away from the former). This last step is performed by general functions that link these concepts together. With these implementations taken into consideration, a user can then refer to functions of their own to create Gothic windows and cathedrals of varied styles, following the procedures described in our evaluation; thus allowing for the derivation of different designs and reducing the time it takes for a designer to apply these modeling processes.

Finally, we evaluate our solution by applying its generative capabilities to the derivation of multiple Gothic cathedral floor plans (e.g., *Cathédrale Notre-Dame de Paris*, among others), incorporating various ornamental elements, and by modeling Gothic windows of varied styles stemming from the same structural representation. On that note, since Gothic Architecture is particularly complex, the results presented herein suggest a potential application to different case studies, so long as appropriate adaption measures are considered.

6.1 Solution Trade-Offs and Future Work

Despite the optimizations and benefits that our approach offers to structural and ornamental variations, paving the way for a wider inclusion of the latter in modeling environments, there are still trade-offs that must be taken into consideration. We will present brief hypotheses on how to mitigate them; thus serving as relevant considerations for future work. On that note, the following is a list of factors that should be taken into consideration:

Modeling approach: Most of the algorithmic descriptions contained within our implementation resort to analytic geometry, which may constitute additional overheads to users without a solid mathematical background. It is possible to simplify these by adapting them to constructive approaches closer to architectural and design thinking (e.g., approaches offered by *CGAL*[1]). Many studies cover these aspects, one example being Ventura's thesis [55];

Fixed floor plan conventions: Despite being adaptable to various Gothic floor plans, our implementation strictly conforms to a set of structural conventions restricted to the cross-shaped floor plan with an eastern ambulatory. Gothic examples from different periods might encompass additional designs to which our approach does not conform (e.g., *Cattedrale di Santa Maria del Fiore*[2]). One possible way of addressing these cases is to consider an alternative implementation comprising a set of procedures that allow for modularized attachment of the sections, or even an AI/ML approach for inverse algorithmic design [57], whilst preserving structural and ornamental encapsulations;

Conflicting Geometry: In our solution, attaching walls to pillars with different dimensions contributes to undesirable structural gaps. Furthermore, certain parametric modifications on windows contribute to undesirable outputs since we did not contemplate a mechanism that enforces geometric constraints. These can be applied to our implementation through the various mechanisms described in Ventura's thesis [55];

***Khepri's* Portability Issues:** Despite working in *AutoCAD*, our implementation seems to stress *Khepri's* portable capabilities when applied to different backend environments (e.g., Rhino, among others), due to what we suppose as being conflicting conversions on the representations of modeling operations from backend to backend. This constitutes an important aspect for future work aimed at amending these corner cases and further sustaining *Khepri's* portability paradigm.

[1] https://www.cgal.org/
[2] https://en.wikipedia.org/wiki/Florence_Cathedral

Bibliography

[1] S. Havemann, “Generative Mesh Modeling,” Ph.D. dissertation, Technische Universität Braunschweig, 2005.

[2] R. Castelo-Branco, I. Caetano, and A. Leitão, “Digital representation methods: The case of algorithmic design,” *Frontiers of Architectural Research*, 2022.

[3] I. Sutherland, “Sketchpad: A man-machine graphical communication system,” in *Proceedings of the May 21-23, 1963, Spring Joint Computer Conference*, 1963.

[4] F. Krull, “The origin of computer graphics within general motors,” *IEEE Annals of the History of Computing*, 1994.

[5] R. Fernandes, “Generative Design: A New stage in the Design Process,” Master’s thesis, Instituto Superior Técnico, Universidade de Lisboa, 2013.

[6] J. Whitehead, “Toward proccedural decorative ornamentation in games,” in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, 2010.

[7] R. Smelik, T. Tutenel, K. de Kraker, and R. Bidarra, “A declarative approach to procedural modeling of virtual worlds,” *Computers & Graphics*, 2011.

[8] S. Haegler, M. Pascal, and L. Van Gool, “Procedural modeling for digital cultural heritage,” *EURASIP Journal on Image and Video Processing*, 2009.

[9] I. Caetano and A. Leitão, “Architecture meets computation: An overview of the evolution of computational design approaches in architecture,” *Architectural Science Review*, 2020.

[10] J. McCormack, A. Dorin, and T. Innocent, “Generative design: A paradigm for design research,” in *DRS Conference Proceedings*, 2004.

[11] I. Caetano, L. Santos, and A. Leitão, “Computational design in architecture: Defining parametric, generative, and algorithmic design,” *Frontiers of Architectural Research*, 2020.

[12] A. Leitão, R. Fernandes, and L. Santos, “Pushing the envelope: Stretching the limits of generative design,” in *Knowledge-based Design: Proceedings of the 17th Conference of the Iberoamerican Society of Digital Graphics (SIGraDi)*, 2013.

[13] R. Oxman, “Theory and design in the first digital age,” *Design Studies*, 2006.

[14] R. Aish and R. Woodbury, “Multi-level interaction in parametric design,” in *Smart Graphics*, 2005.

[15] J. Monedero, “Parametric design: A review and some experiences,” *Automation in Construction*, 2000.

[16] C. Hernandez, “Thinking parametric design: Introducing parametric gaudi,” *Design Studies*, 2006.

[17] E. Whiting, J. Ochsendorf, and F. Durand, “Procedural modeling of structurally-sound masonry buildings,” *ACM Transactions on Graphics*, 2009.

[18] Y. Parish and P. Müller, “Procedural modeling of cities,” in *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, 2001.

[19] P. Janssen and R. Stouffs, “Types of parametric modelling,” in *Emerging Experiences of the Past, Present and Future of Digital Architecture*, 2015.

[20] U. Krispel, C. Schinko, and T. Ullrich, “A survey of algorithmic shapes,” *Remote Sensing*, 2015.

[21] R. Smelik, T. Tutenel, R. Bidarra, and B. Benes, “A survey on procedural modelling for virtual worlds,” *Computer Graphics Forum*, 2014.

[22] G. Stiny and J. Gips, “Shape grammars and the generative specification of painting and sculpture,” in *Information Processing, Proceedings of IFIP Congress*, 1971.

[23] M. Özkar and S. Kotsopoulos, “Introduction to shape grammars,” in *ACM SIGGRAPH 2008 Classes*, 2008.

[24] N. Chomsky, “Three models for the description of language,” *IRE Transactions on Information Theory*, 1956.

[25] G. Stiny, “Introduction to shape and shape grammars,” *Environment and Planning B: Planning and Design*, 1980.

[26] H. Konig and J. Eizenberg, “The language of the prairie: Frank lloyd wright’s prairie houses,” *Environment and Planning B: Planning and Design*, 1981.

[27] P. Prusinkiewicz and A. Lindenmayer, *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990.

[28] O. Deussen and B. Lintermann, *Digital Design of Nature: Computer Generated Plants and Organics*. Springer, 2005.

[29] G. Rozenberg and A. Salomaa, *The Mathematical Theory of L Systems*. Springer, 1980.

[30] H. Abelson and diSessa Andrea, *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*. The MIT Press, 1981.

[31] R. Goldman, S. Schaefer, and T. Ju, "Turtle geometry in computer graphics and computer-aided design," *Computer-Aided Design*, 2004.

[32] P. Prusinkiewicz and L. Kari, "Subapical bracketed l-systems," in *International Workshop on Graph Grammars and Their Application to Computer Science*, 1994.

[33] M. Lipp, P. Wonka, and M. Wimmer, "Interactive visual editing of grammars for procedural architecture," in *ACM SIGGRAPH 2008 Papers*, 2008.

[34] C. Schinko, M. Strobl, T. Ullrich, and D. Fellner, "Scripting technology for generative modeling," *International Journal On Advances in Software*, 2011.

[35] F. Zhiwen, Z. Lingjie, L. Honghua, C. Xiaohao, Z. Siyu, and T. Ping, "Floorplancad: A large-scale cad drawing dataset for panoptic symbol spotting," in *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021.

[36] K. Steinfeld, "Dreams may come," in *ACADIA 2017: DISCIPLINES & DISRUPTION (Proceedings of the 37th Annual Conference of the Association for Computer Aided Design in Architecture)*, 2017.

[37] L. Gatys, A. Ecker, and M. Bethge, "A neural algorithm of artistic style," *arXiv*, 2015.

[38] A. S. Incorporated, *PostScript Language Reference Manual*. Addison-Wesley Longman Publishing Company, 1990.

[39] M. Wegener, "Operational urban models state of the art," *Journal of the American Planning Association*, 1994.

[40] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky, "Instant architecture," *ACM Transactions on Graphics*, 2003.

[41] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool, "Procedural modeling of buildings," *ACM Transactions on Graphics*, 2006.

[42] M. Saldana, "An integrated approach to the procedural modeling of ancient cities and buildings," *Digital Scholarship in the Humanities*, 2015.

[43] P. Müller, T. Vereenooghe, P. Wonka, I. Paap, and L. Van Gool, "Procedural 3d reconstruction of puuc buildings in xkipché," in *Proceedings of the 7th International Conference on Virtual Reality, Archaeology and Intelligent Cultural Heritage*, 2006.

[44] J. Halatsch, A. Kunze, and G. Schmitt, "Using shape grammars for master planning," in *Design Computing and Cognition '08*, 2008.

[45] L. Shanglin, W. Manli, O. Zi, H. Qingwen, and X. Juan, "Rapid modelling method and application using cga," *Journal of Physics: Conference Series*, 2021.

[46] C. Barrios, "The search of form, the search of order: Gaudí and the sagrada familia," in *ACM SIGGRAPH 2008 Art Gallery*, 2008.

[47] M. Kramer and E. Akleman, "A procedural approach to creating american second empire houses," *Journal on Computing and Cultural Heritage*, 2020.

[48] E. Catmull and C. Jim, "Recursively generated b-spline surfaces on arbitrary topological meshes," *Computer-Aided Design*, 1978.

[49] I. Stroud, *Boundary Representation Modelling Techniques*. Springer London, 2006.

[50] C. Eastman and K. Weiler, "Geometric modeling using the euler operators," *Pittsburgh: Institute of Physical Planning, Carnegie Mellon University*, 1979.

[51] R. Castelo-Branco and A. Leitão, "Visual meets textual: A hybrid programming environment for algorithmic design," in *RE: Anthropocene - Design in the Age of Humans: Proceedings of the 25th International Conference of the Association for Computer-Aided Architectural Design Research in Asia (CAADRIA)*, 2020.

[52] M. J. Sammer, A. Leitão, and I. Caetano, "From visual input to visual output in textual programming," in *Intelligent & Informed: Proceedings of the 24th International Conference of the Association for Computer-Aided Architectural Design Research in Asia (CAADRIA)*, 2019.

[53] J. Bezanson, A. Edelman, S. Karpinski, and V. Shah, "Julia: A fresh approach to numerical computing," *SIAM Review*, 2017.

[54] C. Reas and B. Fry, *Processing: A Programming Handbook for Visual Designers and Artists*. MIT Press, 2007.

[55] R. Ventura, "Geometric Constraints in Algorithmic Design," Master's thesis, Instituto Superior Técnico, Universidade de Lisboa, 2021.

[56] S. Havemann and D. Fellner, “Generative parametric design of gothic window tracery,” in *Proceedings Shape Modeling Applications*, 2004.

[57] J. David, “Inverse Algorithmic Design: Automatic Conversion of Floor Plan Images into Computer Programs,” Master’s thesis, Instituto Superior Técnico, Universidade de Lisboa, 2022.

A

Project Code

```
function Gothic_Window_Second_Style(bottom_left_corner, upper_right_corner,
                                    excess, vertical_distance_to_sub_arch,
                                    outer_offset, inner_offset)
    profile = surface_circle(u0(), DEFAULT_PROFILE_RADIUS)
    main_rosette_style = Rosette_Rounded_Style(6, pi/2, profile)
    sub_rosette_style = Rosette_Rounded_Style(4, 0, profile)
    sub_sub_rosette_style = Rosette_Rounded_Style(3, pi/2, profile)
    sub_sub_sub_arches_style = Sub_Gothic_Window(profile, nothing, nothing,
                                                 nothing)
    sub_sub_arches_style = Sub_Gothic_Window(profile, sub_sub_rosette_style,
                                             sub_sub_sub_arches_style,
                                             sub_sub_sub_arches_style)
    sub_arches_style = Sub_Gothic_Window(profile, sub_rosette_style,
                                         sub_sub_arches_style,
                                         sub_sub_arches_style)
    main_arch = Gothic_Window(bottom_left_corner, upper_right_corner,
                              excess, vertical_distance_to_sub_arch,
```

```
                                outer_offset, inner_offset, profile,
                                main_rosette_style, sub_arches_style,
                                sub_arches_style)
    return main_arch
end
Gothic_Window_Second_Style(xy(-50, -42), xy(50, 42), 1, 6, 2, 2)
```

Listing A.1: The generative function of a Gothic window with a rounded foiled rosette.

```
function Gothic_Window_Third_Style(bottom_left_corner, upper_right_corner,
                                   excess, vertical_distance_to_sub_arch,
                                   outer_offset, inner_offset)
    profile = surface_circle(u0(), DEFAULT_PROFILE_RADIUS)
    main_rosette_style = Rosette_Pointed_Style(6, pi/2, 2, profile)
    sub_rosette_style = Rosette_Pointed_Style(4, 0, 2, profile)
    sub_sub_rosette_style = Rosette_Pointed_Style(3, pi/2, 2, profile)
    sub_sub_sub_arches_style = Sub_Gothic_Window(profile, nothing, nothing,
                                                 nothing)
    sub_sub_arches_style = Sub_Gothic_Window(profile, sub_sub_rosette_style,
                                             sub_sub_sub_arches_style,
                                             sub_sub_sub_arches_style)
    sub_arches_style = Sub_Gothic_Window(profile, sub_rosette_style,
                                         sub_sub_arches_style,
                                         sub_sub_arches_style)
    main_arch = Gothic_Window(bottom_left_corner, upper_right_corner,
                              excess, vertical_distance_to_sub_arch,
                              outer_offset, inner_offset, profile,
                              main_rosette_style, sub_arches_style,
                              sub_arches_style)
    return main_arch
end
Gothic_Window_Third_Style(xy(-50, -42), xy(50, 42), 1, 6, 2, 2)
```

Listing A.2: The generative function of a Gothic window with a pointed foiled rosette.

```
function Gothic_Window_Fourth_Style(bottom_left_corner, upper_right_corner,
                                    excess, vertical_distance_to_sub_arch,
```

```
                                    outer_offset, inner_offset)
    circle_profile = surface_circle(u0(), DEFAULT_PROFILE_RADIUS)
    quad_star_profile = union(surface(regular_polygon(4, u0(),
                                      DEFAULT_PROFILE_RADIUS, 0)),
                              surface(regular_polygon(4, u0(),
                                      DEFAULT_PROFILE_RADIUS, pi/4)))
    hexagon_profile = surface(regular_polygon(6))
    main_rosette = Rosette_Pointed_Style(3, pi/2, 2, hexagon_profile)
    sub_left_rosette = Rosette_Rounded_Style(6, 0, circle_profile)
    sub_right_rosette = Rosette_Pointed_Style(9, pi/4, 5, quad_star_profile)
    sub_sub_arches_style = Sub_Gothic_Window(circle_profile, nothing,
                                             nothing, nothing)
    left_sub_arch_style = Sub_Gothic_Window(hexagon_profile,
                                            sub_left_rosette,
                                            sub_sub_arches_style,
                                            sub_sub_arches_style)
    right_sub_arch_style = Sub_Gothic_Window(circle_profile,
                                             sub_right_rosette,
                                             sub_sub_arches_style,
                                             sub_sub_arches_style)
    main_arch = Gothic_Window(bottom_left_corner, upper_right_corner,
                              excess, vertical_distance_to_sub_arch,
                              outer_offset, inner_offset, quad_star_profile,
                              main_rosette, left_sub_arch_style,
                              right_sub_arch_style)
    return main_arch
end
Gothic_Window_Fourth_Style(xy(-50, -42), xy(50, 42), 1, 6, 2, 2)
```

Listing A.3: The generative function of a Gothic window with each constituent incorporating a different style.

```
# == WINDOW STYLES == #
function Gothic_Window_First_Style(...)
function Gothic_Window_Second_Style(...)
function Gothic_Window_Third_Style(...)
function Gothic_Window_Fourth_Style(...)
# == WINDOW STYLES == #
# == AMIENS, CATHEDRALE NOTRE-DAME == #
```

```
instantiate_measurements_and_delimiters(7.53 * 2,
                                        3:6, 1:7,
                                        2:7, 10:13,
                                        1:4, 8:9,
                                        5:8, 8:9,
                                        2:7, 14:16)

pillars = Array{Union{Pillar, Nothing}}(nothing, 8, 16)
instantiate_pillars_info(1, 1, 105)
instantiate_all_pillars(pillars)
instantiate_walls_info()
instantiate_all_walls(pillars)
# == AMIENS, CATHEDRALE NOTRE-DAME == #
```

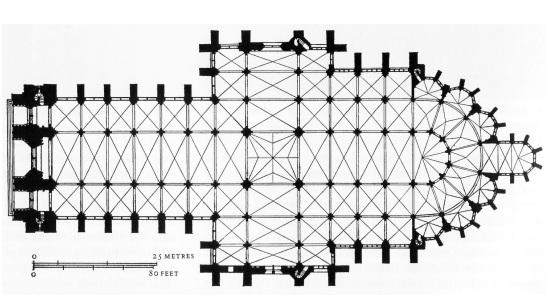
Listing A.4: The set of generative procedures to model the Amiens cathedral.

```
# == WINDOW STYLES == #
function Gothic_Window_First_Style(...)
function Gothic_Window_Second_Style(...)
function Gothic_Window_Third_Style(...)
function Gothic_Window_Fourth_Style(...)
# == WINDOW STYLES == #
# == PARIS CATHEDRALE NOTRE-DAME == #
instantiate_measurements_and_delimiters(7.53 * 2,
                                        1:6, 1:8,
                                        1:6, 11:15,
                                        1:3, 9:10,
                                        4:6, 9:10,
                                        1:6, 16:17)
pillars = Array{Union{Pillar, Nothing}}(nothing, 6, 17)
instantiate_pillars_info(1, 1, 105)
instantiate_all_pillars(pillars)
instantiate_walls_info()
instantiate_all_walls(pillars)
# == PARIS CATHEDRALE NOTRE-DAME == #
```

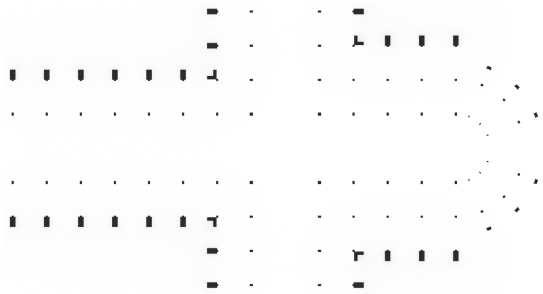
Listing A.5: The set of generative procedures to model the Paris cathedral.

B

Real-Life Gothic Cathedral Models



(a) Amiens cathedral floor plan.[1]



(b) Amiens cathedral structure.

Figure B.1: Side-by-side illustrations of the Amiens cathedral floor plan (a) and corresponding structural derivation of Listing A.4 (b).

[1] Original from https://mcid.mcah.columbia.edu/art-atlas/mapping-gothic/amiens-cathedrale-notre-dame
[2] Original from https://mcid.mcah.columbia.edu/art-atlas/mapping-gothic/paris-cathedrale-notre-dame

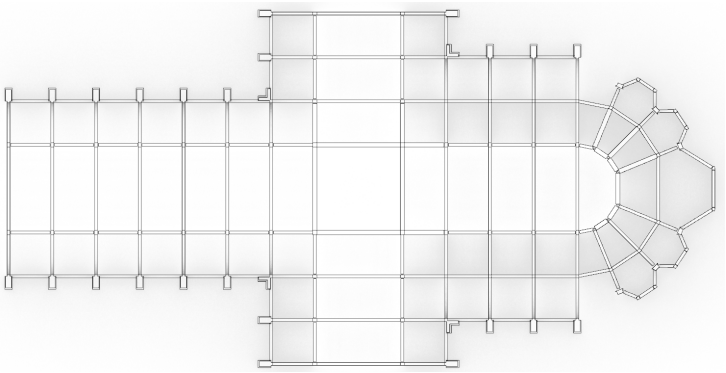
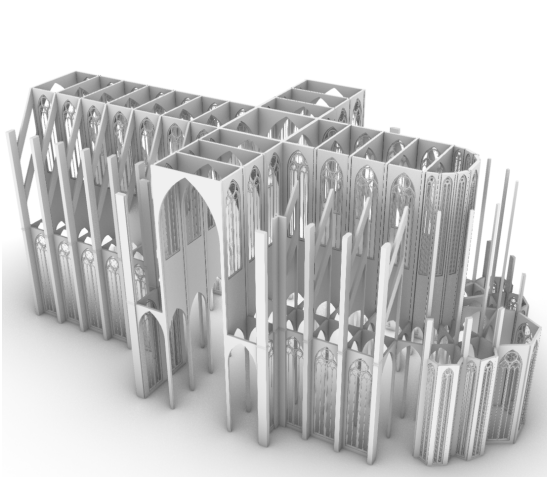
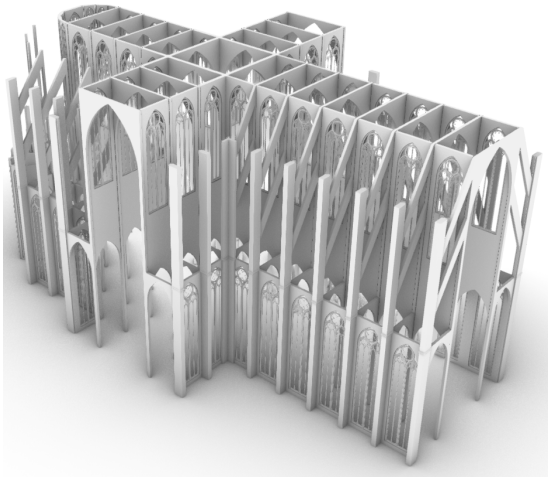


Figure B.2: Top view of the Amiens cathedral model of Listing A.4.

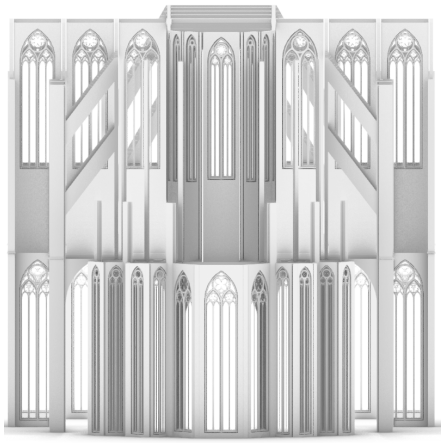


(a) Front perspective view of the Amiens cathedral of Listing A.4.

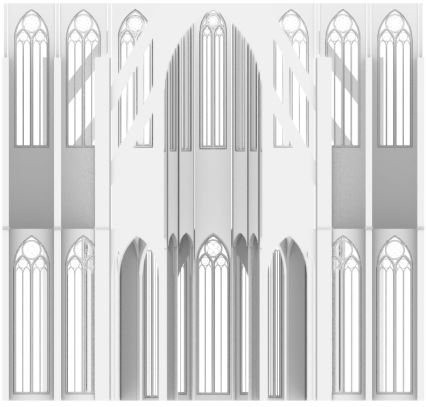


(b) Back perspective view of the Amiens cathedral of Listing A.4.

Figure B.3: Two perspective views of the Amiens cathedral of Listing A.4

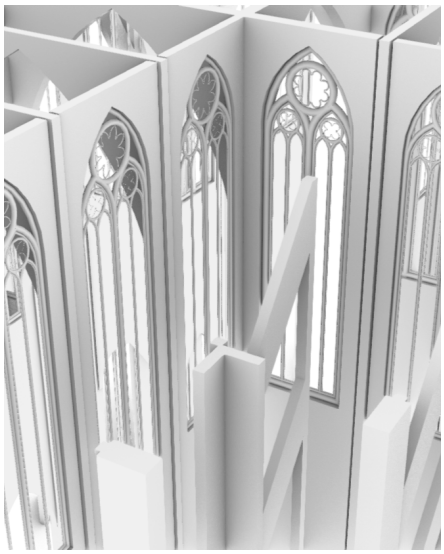


(a) Front view of the Amiens cathedral of Listing A.4.



(b) Back view of the Amiens cathedral of Listing A.4.

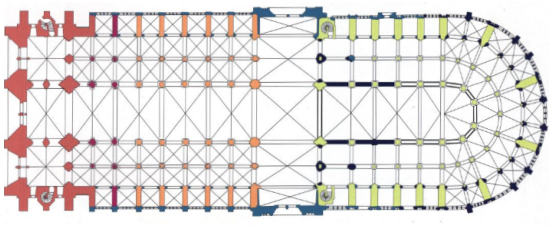
Figure B.4: Two slightly-angled views of the front (a) and back (b) sides of the Amiens cathedral of Listing A.4.



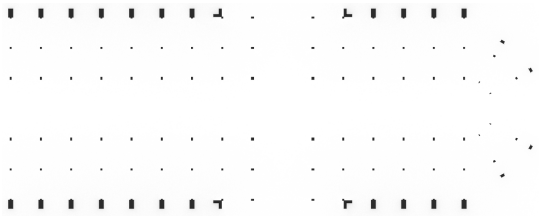
(a) Upper windows of the Amiens cathedral of Listing A.4.

(b) Lower windows of the Amiens cathedral of Listing A.4.

Figure B.5: Two views depicting the different styles of the upper (a) and lower (b) windows of the right side of the Amiens cathedral of Listing A.4.



(a) Paris cathedral floor plan.[2]



(b) Paris cathedral structure.

Figure B.6: Side-by-side illustrations of the Paris cathedral floor plan (a) and corresponding structural derivation of Listing A.5 (b).

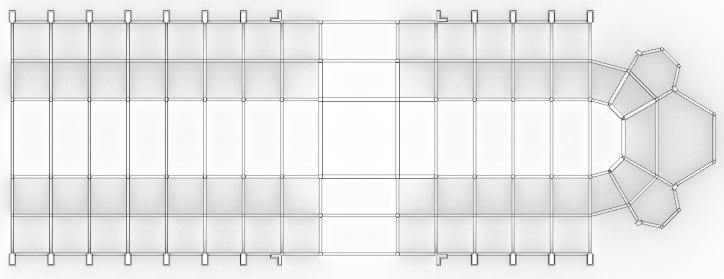
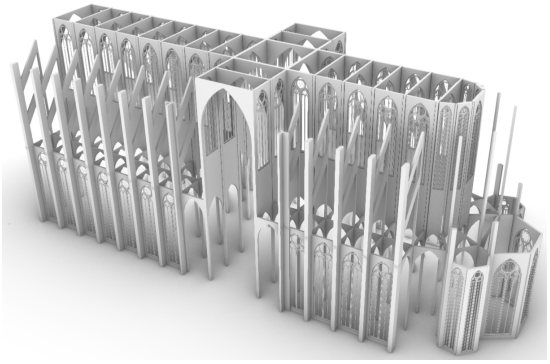
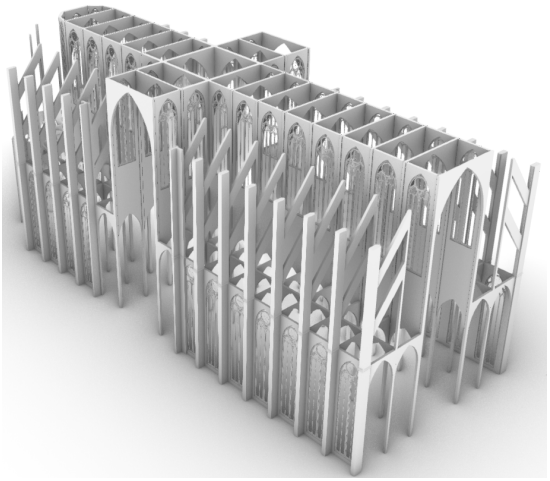


Figure B.7: Top view of the Paris cathedral model of Listing A.5.

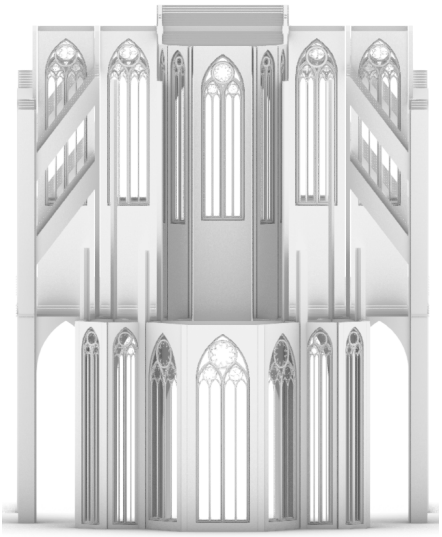


(a) Front perspective view of the Paris cathedral of Listing A.5.

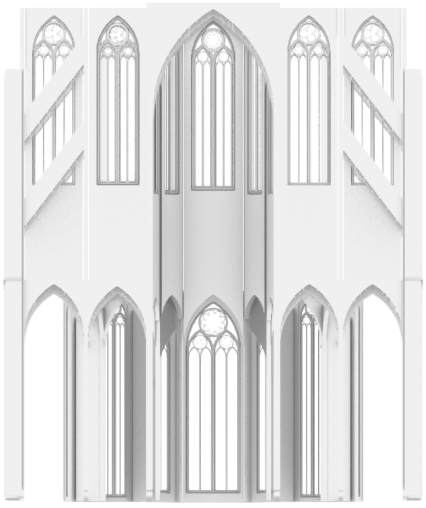


(b) Back perspective view of the Paris cathedral of Listing A.5.

Figure B.8: Two perspective views of the Paris cathedral of Listing A.5.



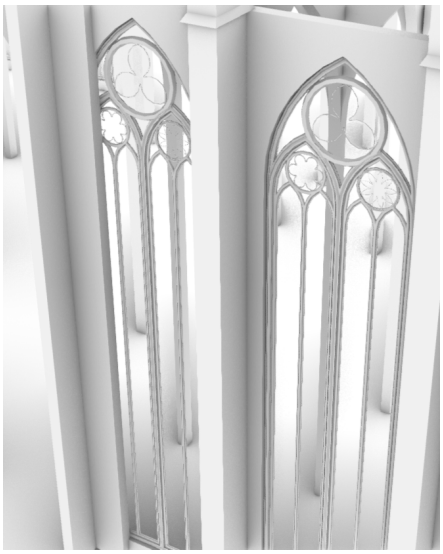
(a) Front view of the Paris cathedral of Listing A.5.



(b) Back view of the Paris cathedral of Listing A.5.

Figure B.9: Two slightly-angled views of the front (a) and back (b) sides of the Paris cathedral of Listing A.5.

(a) Upper windows of the Paris cathedral of Listing A.5.



(b) Lower windows of the Paris cathedral of Listing A.5.

Figure B.10: Two views depicting the different styles of the upper (a) and lower (b) windows of the right side of the Paris cathedral of Listing A.5.