Utrecht University

MASTER THESIS

Procedural generation of man-made objects: a case study in Gothic window traceries

---

Author:
R. V. VERMEULEN
IC-6339034

First examiner:
J. L. VERMEULEN Msc
Second examiner:
Dr. F. STAALS

A thesis submitted in fulfillment of the Master's programme in

Game and Media Technology

at the

Department of Information and Computing Sciences

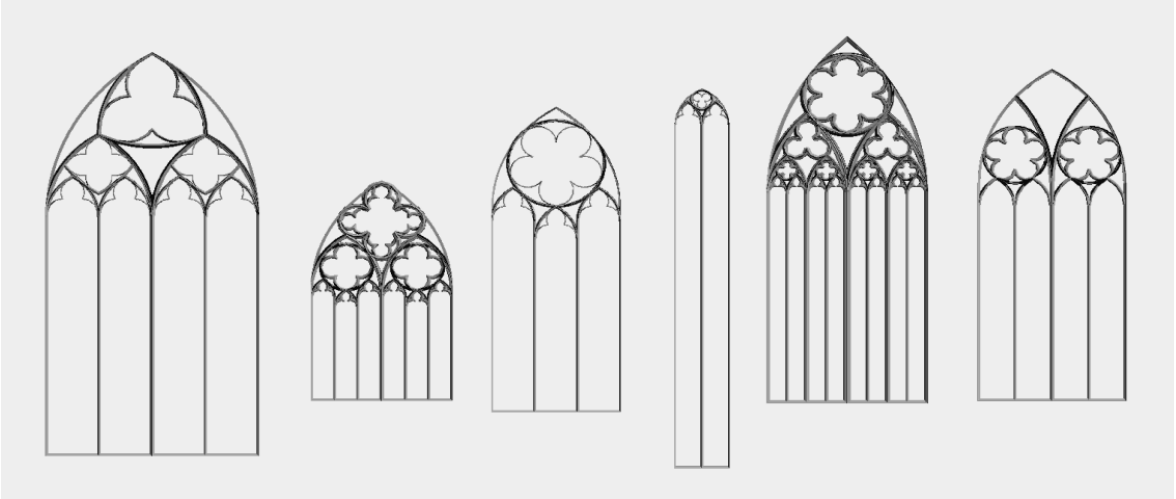


FIGURE 1: Some examples of our procedurally generated Gothic window traceries taken from our test set (picture numbers from left to right: 61, 54, 15, 55, 75, 22).

1 Introduction

In today's world, computer games and computer-created animation play a part in the daily lives for hundreds of millions of people around the globe and the video game industry has been prospering for many years [13]. To make virtual worlds interesting and keep gamers entertained, many assets still need to be crafted manually by specialist artists. This makes interesting, large-scale environments very costly to produce and hence unattainable for most small developers. A possible manner in which to reduce these costs is to make use of a technology named *procedural content generation*, usually abbreviated as PCG [42, 44].

PCG is the field of creating game content automatically through algorithmic means [42]. Content in this case includes any type of content imaginable, such as 3D-assets, materials, textures, sounds, NPCs, story-lines, speech, game levels, game mechanics, in-game objects such as weapons or collectibles, cities, world maps, particle systems, lightening of the scene and camera angles [25, 42]. PCG has been a popular topic of research in the past decade, especially the algorithmic generation of game levels and game worlds [25]. However, despite a call for research in PCG for decorative ornamentation made in 2010 [44], this field has remained relatively unexplored [13, 25]. Meanwhile decorative ornamentation remains one of the costliest aspects in creating diverse, interesting, and realistic large-scale environments, as each ornamental object in a virtual world still needs to by hand-crafted by a 3D-artist [44].

1.1 Contribution

In this paper, we wish to explore this field by proposing an approach towards creating and evaluating a PCG system for decorative man-made objects. The contribution of this paper will be two-fold. First, we will propose a general approach towards creating a PCG system that can automatically generate aesthetic objects that appear to have been designed by a human. Secondly, we propose a simple, inexpensive method to objectively test the quality of such a generator. To put both our general approach and our evaluation method to the test, we will create a PCG system for Gothic window traceries following our approach and evaluate it following our method. This way we will show that it is possible to automatically generate designs that are indistinguishable from man-made designs but that still provide noticeable variety.

1.2 Scope

Due to time and resource constraints, we chose to limit our project to the procedural generation of Gothic window tracery designs and the evaluation of this generator. We chose to focus on Gothic window traceries specifically because these type of windows display a great variety in design [5, 40] while being purely geometric in nature [5, 11, 40]. In fact, most of these designs can be faithfully modeled using only circle parts and straight lines because the medieval architects that created the original designs likely did so using only a pass and a straight edge [5, 11, 40]. The geometric nature of these designs enables the purely algorithmic generation of 3D meshes without the need of any type of exemplary data, such as pre-existing 3D-models or pictures. This allows us to focus purely on the creative generation of ornamental designs rather than on how to attain, select and process such data. Also, the absence of movable parts in Gothic windows, such as shutters or sliding frames, allows us to ignore the question of how to faithfully display the various possible states of these parts. In other words, because Gothic windows typically cannot be opened, we do not need to generate multiple meshes of the same window being fully opened, partly opened or closed, nor do we need to generate any type of armature for our windows, again allowing us to focus purely on the decorative aspects.

Additionally, Gothic windows are easy to evaluate qualitatively in a fair manner. As we did not have a suitable set of manually created meshes available to compare our generated meshes with, we needed to create this set ourselves. To do this in a fair manner, we needed a real-world, well-defined class of objects to generate that the general public is familiar with and already has some preconceived notion of. In other words, most people already have a preconceived notion of what constitutes as a 'medieval Gothic window' and 'medieval Gothic windows' is an already well-defined, uncontested set of real-world objects (it can reliably be confirmed from which church or cathedral a specific design is originated and whether this building was built in the Gothic period). Hence, we can fairly argue that our generated designs are qualitatively good if our test subjects cannot accurately distinguish our generated designs from real world designs.

In the section 'Relevant work' we will discuss previous research done on shape grammars and the procedural generation of architecture, Gothic window traceries and other man-made objects and patterns that belong to a specific style or design tradition. Then we will introduce our general approach, discuss why we think this approach is suitable for the generation of man-made objects and discuss some challenges and considerations that are inherent to our approach. In the section 'System' we will describe in more detail how we generated the Gothic window traceries, and in the section 'Evaluation method' we will lay out and discuss our proposed method to objectively test aesthetic content and give details on our implementation of this method. In 'Results and discussion' we will present and discuss our survey results, in 'Conclusion' we will summarize our conclusion and we will suggest some possible avenues for further research in 'Future work'.

2 Relevant work

To the authors' knowledge, there has been no previous research towards a general PCG algorithm for man-made objects, nor has there been any previous research towards *shape grammars* (introduced in the section 'Shape Grammars') fit for automatic generation. In contrast, there has been a lot of research focused on creating particular PCG algorithms and shape grammars for very specific sets of man-made objects, including Gothic window traceries [11, 40]. In this section we will discuss literature on the following subjects: shape grammars, PCG algorithms for architecture, PCG algorithms for Gothic window traceries and finally

we will give a very brief overview of some other PCG algorithms based on specific style rules or style conventions.

2.1 Shape grammars

Shape grammars were first introduced by Stiny and Gips in 1971 [39]. A shape grammar is defined as a formal, exact manner to represent a set of geometrical shapes that can be generated by recursively applying rules selected from a single, finite set onto a predefined starting shape. Previously, formal grammar systems only supported textual representations rather than geometrical shapes. Stiny and Gips proposed a set of grammars with which geometric, non-representational paintings and sculptures could be defined and created.

Their original grammar model was very simple, consisting of a conclusive set of terminal shapes that are part of the end result and cannot be erased, a set of non-terminal shapes that cannot be part of the end result and thus need to be erased before an end result can be found, shape rules that replace one set of terminal and non-terminal shapes with another such set and one predetermined initial shape. Recursively, a rule would be matched with a sub-shape of the starting shape and the matching sub-shape would be altered in a way that is defined by that rule. This would be done until no rules applied to the shape. Various different outcomes were realized by allowing multiple shape rules to apply to the same input shapes. No algorithm is given to select the most suitable rule out of those applicable as this grammar was not used for the fully automated creation of art, but rather as a tool to be used by the artist. In 1980 Stiny summarized later elaborations on this simple model [38], such as the addition of labeled points with which it was easier to uniquely represent a subshape and parametric shape grammars with which it was easier to define rules that could be matched to a wider range of subshapes.

Later work on shape grammars mostly focused on their application in CAD systems to enable designers to efficiently create prototypes and improve their communication with consumers [15, 32], and on using shape grammars to achieve consistency in brand adherence in designs [6]. Additionally, a multitude of shape grammars was created to represent sets of objects in specific styles or design traditions, an overview of which was given by Chau et al. [6].

2.2 Architecture

While we did not find any previous work on PCG for decorative man-made designs or objects in general, we did find various PCG algorithms for the creation of individual buildings. None of these algorithms were directly suitable for creating objects with a more complex design such as Gothic window traceries, but many ideas on these algorithms concerning design logic are applicable to man-made objects in general. For this reason we found it worthwhile to briefly go over the development in PCG algorithms for buildings and architecture.

One of the most commonly used methods for building generation are split grammars, introduced by Wonka et al. in 2003 [45]. They observed that previously existing PCG methods, such as L-systems, were unfit for the generation of architecture because the structure of buildings does not reflect a growth process, unlike elements like plants and road networks. Furthermore, using these methods it is difficult to set spatial constraints for the generated geometry which may lead to buildings growing into each other. Therefore, Wonka et al. devised a grammar in which a building façade is recursively described as a partitioning of either axis-aligned subspaces or so-called *terminal shapes*, until there are no more subspaces to be divided. Which partitioning or terminal shape is selected to fill an open subspace is determined by attributes assigned to the subspaces that describe the semantic meaning of that

space, such as 'first floor', 'wall' or 'door'. As subspaces may be defined in arbitrary detail this algorithm is be able to generate windows together with the rest of a building's façade, but there is a major limit to the expressiveness of this algorithm as the design is bound to the same axis-aligned grid on every level in the split hierarchy [45].

In 2006, Müller et al. created *CGA shape* [30], a shape grammar for the procedural generation of buildings which combines split-grammar-based façades with volumetric models based on sets of arbitrarily translated and rotated primitives that were used earlier as elementary building models in the work on procedural city generation by Parish and Müller [33]. *CGA shape* grammars describe coherent façades for these more complex mass models that are aware of the relative positioning and intersections of the volumetric primitives. This way things like illogical positioning of doors or walls intersecting windows can be prevented in building generation. *CGA shape* is utilized in a popular commercial city generation tool named *CityEngine* [43].

In later years, a number of extensions to split grammars have been proposed to increase the expressiveness of the algorithm. In 2013, Thaller et al. [41] extend split grammars by allowing split operations not only on axis-aligned planes, but on any plane in a 3D-space, effectively replacing the hierarchical axis-aligned box structure enforced by a split grammar with a hierarchical structure of convex polyhedra. They show that any operation that can be preformed on the original split grammar structure can be generalized to their convex-polyhedron-based structure [41].

A year later, Zmugg et al. [46] extend on convex-polyhedron-based split grammars by integrating free-form deformations into the grammar to allow curved geometry. As deformations can be applied at any level in the hierarchy, subsequent operations are aware of the deformations and can be made to adapt to them in an intelligent manner. In 2016, Jesus et al. [16] introduce layered split grammars, in which multiple subdivision hierarchies are defined for the same space of which the geometry can be merged to create one object. An advantage of this addition is that layers can represent design aspects that can be easily defined independently of other aspects, even when they intersect. Another addition that is presented in this paper is the usage of 2D vector shapes to represent complex geometry on planar surfaces [16].

While these various extensions to split grammars greatly increased the complexity of designs that could be generated and could potentially be used to generate Gothic window tracings, we found that the restrictions inherent to these grammars still made them impractical both for more general design algorithms and for the generation of Gothic window traceries. Specifically, many human-made designs include non-convex patterns or elements of which the design process cannot be replicated as a split grammar in a natural manner. Furthermore, we found it not necessary to strictly adhere to this restriction in order to generate realistic Gothic window tracery models, even though we did adhere to a type of recursive partitioning of 2D space to create the designs for our traceries (this is explained in more detail in the 'System' section).

In 2011, Kelly and Wonka [19] presented an entirely novel shape grammar that is able to model a building based on a collection of 2D-curves tracing the shapes of the building's features. Their method is able to model complex architectural features that cannot be easily defined using basic split grammars, such as overhanging rooftops, dormer windows and chimneys. However, as some input curves are needed, this method is less suited for procedural generation than split-grammar-based approaches.

2.3 Gothic windows

Havemann and Fellner [11] were the first to attempt to generate Gothic church windows using a functional model representation method they called *Generative Modeling Language* (GML) [12]. However, as their generation method is based around a specific prototype window template it falls short on representing the wide variety of Gothic window tracery patterns that exist in the real world.

Charbonneau et al. [5] presented a shape-grammar for the creation of Gothic rose windows as an historical case study to explore the possibility of utilizing shape-grammars to more efficiently but faithfully construct 3D-models of archaeological artifacts. While their approach resembles our proposed approach, the focus of this paper was to provide evidence of the archaeological validity of this approach rather than to explore the system or the approach itself. The system they presented was very simple and only meant as an aid for manual modeling of historic architectural features. They did not touch on the possibilities and challenges of a more complex system or on its usefulness in procedural content generation.

Takayama [40] created a more flexible method, that could potentially recreate many real world Gothic traceries. It is based on a limited number of basic motifs commonly found in Gothic window tracery, such as arches and foils, which are defined as compositions of circle parts. These motifs are placed into a field, which initially is the area of the window, until there is too little space left in the field to place another motif. This is done without overlap of the motifs and with regards to bilateral or rotational symmetry. Then the inner areas of the motifs will be used as fields to be filled in a recursive manner until the end criteria are met. As the exact shapes of these motifs are variable and the composition of motifs in a field is variable as well, this algorithm is quite expressive, but through the definition of the basic motifs and the space-filling nature of the algorithm, the results still fit within the bounds of the Gothic style.

While Takayama [40] did succeed in generating complex Gothic window traceries with sufficient expressiveness, his algorithm is specifically designed to generate Gothic window traceries and he did not touch on other styles, other architectural elements or on the generation of human-made objects in general. His approach is unsuitable for most domains outside Gothic window tracery.

2.4 Style-based PCG

Similarly to how Takayama's system is based on the particularities of (a subset of) Gothic window tracery designs, there have been many other PCG systems that exclusively generate decorative objects or patterns within a given style.

Some of these systems exemplify the practicality of shape-grammar-based systems for the reconstruction of archaeological sites, examples include systems that reconstruct ancient Mayan [29, 34] and Roman [28, 31, 35] settlements, systems that create Greek Ionic-style columns [20] and systems that reconstruct Doric temples [27]. Other style-based PCG systems have been built with the aim of preserving traditional building styles. This is done by digitizing the specific rules and elements by which these traditional buildings are designed. Examples include a system that automatically generates traditional Balinese settlements [9], a system that faithfully constructs traditional East-Asian roof styles [23] and systems that create traditional Chinese houses [14, 24].

Other work focuses on the generation of certain traditional design patterns. For example, Lee et al. created a system to generate designs in a Korean traditional pattern called *bosang-whamun* [22], Dalvandi et al. created a system to generate faithful designs for a specific type

of traditional Persian rugs [8] and there have been multiple papers on the generation of a specific type of irregular Chinese lattices named 'ice-ray' lattices [10, 37].

A particular design tradition that has been widely studied are Islamic patterns, as these patterns are based on an advanced underlying geometric and mathematical logic which for a large part has been lost in history. Kaplan [18] was one of the first to present a method to algorithmically recreate these patterns by first generating the star or rosette patterns that are typical for this style and then filling the space in between. Kaplan and Salesin [17] later built further on this work by extending the method to generate Islamic patterns on a sphere and on a hyperbolic plane. Cromwell [7] later devised a tessellation-based method for Islamic pattern PCG that was based on historic findings by Lu and Steinhardt [26].

3 General approach

In this section we will present the main contribution of this paper, namely a general approach towards the development of PCG systems for ornamental man-made objects. This section is built up as follows: first we will introduce our approach in the section 'Modules', then we will discuss some advantages of the control inherent to our approach in the section 'Control' and in the section 'Variation' we will discuss how variable output can be realized with our approach. In the section 'General module design' we will discuss some general guidelines in the design of modules and in the section 'Module order' we will discuss how the order in which modules are executed can influence the output of the system. Finally, in the section 'Module parameters' we will discuss how modules may influence the behaviors of other modules using parameters and in the section 'Context tree' we will present the concept of a *context tree* and discuss its benefits.

3.1 Modules

The main challenge in creating a general PCG system for man-made objects is the inherent inability of a machine to invent new solutions for new problems. In most PCG research, the subject matter is either something that is created in a natural process that can be digitized in an algorithm (such as terrains, road networks or vegetation), or something of which the quality can be evaluated by a machine so that the generated content can be optimized automatically (such as maps and levels, that can be tested on difficulty and playability) [25, 42]. In contrast, man-made objects are created through a series of creative design decisions that cannot be uniformly digitized and the generated content is of purely aesthetic value, meaning that it cannot be reliably evaluated without some form of human input. It is for this reason that the previously discussed systems still either need a large amount of human input, such as the manually designed rules of shape and split grammars, or are restricted to a very specific subset of objects so that they can be generated through uniform rules, such as is the case with the style-based PCG systems. However, the existence of these style-based PCG systems shows us that man-made objects that are stylistically similar to each other often can be generated using uniform algorithms. The reason for this is that, while the creativity of a human designer is theoretically limitless, the elements that constitute a unique design are rarely unique themselves as designers are usually taught and inspired by other designers in their environment. This results in the existence of many real-world design patterns that reoccur across different unique designs.

Our approach takes advantage of this repetitiveness by linking these reoccurring design elements to reoccurring pieces of code. We will call these reoccurring pieces of code *modules*. We propose that if a PCG system consists out of a set of manually created modules that accurately reflect reoccurring steps of a possible design process and that can be combined to

form a variety of different complete design procedures, the resulting designs will be realistic and the system will be able to create a large variation of unique designs that would be costly to create otherwise.

A module may represent any algorithm that is relevant to a design process, such as different algorithms that fill a 2D space, algorithms that create different types of symmetry or repetition, algorithms that create a specific shape conforming to specific requirements, but also algorithms that do not directly generate design elements but instead preform other tasks such as analyzing geometry and its context or making a design decision. The main prerequisite for a module is that it reflects a part of a human design process that is sufficiently independent of the rest of the process so that it can be resolved in a similar manner across multiple unique design processes.

A complete PCG system consists of a large collection of modules and each generation procedure is the execution of a starting module, which represents the object being generated and which determines how this is done. Usually, the starting module will call on other modules in the system, which may represent sub-elements of the object or which may preform some other task for the starting module. These modules in turn may or may not call other modules, and so it continues until all modules return and thus the desired object has been generated by the starting module. A system may be non-deterministic if one or more of its modules is non-deterministic and in most cases there are modules that call other modules in a non-deterministic manner, creating a non-deterministic combination of modules that together form a complete, non-deterministic design process.

3.2 Control

As each module is created manually, the creator of the PCG system has a lot of control over the output. This makes it easy to ensure that the final output is realistic and aesthetically pleasing. Additionally, the manual creation of modules allows for the integration of other, more specific PCG algorithms.

Another advantage of a system created following our approach is that it lends itself well to giving control to non-technical users and enabling them to take advantage of PCG. The connection of individual modules to parts of a real-world design process makes it easier to understand how the system works and it is easy to combine the system with externally created content, as modules that would otherwise generate this content can be replaced.

3.3 Variation

While a manually created system is incapable of creating truly original design ideas, there are two ways in which variation can still be created. The first way is by using random numbers as exact values in a generation algorithm. This can be done for values such as the dimensions and positioning of geometry but also values like frequencies, angles and ratios. These values will usually be generated within a predetermined range and with a predetermined distribution so that quality output can be ensured.

An advantage of this type of variation is that a very large number of unique values can be created with very little effort, even if the range of this variation is limited. A disadvantage is that it is non-trivial to create a variety of content in this manner that is structurally variant rather than only variant in the exact values of certain variables, which might make the generated objects noticeably similar. However, there might still be a lot of visual variation that can be generated even with this type of variety alone.

The second type of variation is the variation that is gained by giving the system a random choice between a limited number of alternative solutions to the same problem. In many cases, each solution is implemented as a separate module so that a calling module

has a number of optional modules to call from, but this is not necessarily the case as such a choice might also be implemented as an if-else statement, among other options. The second type of variation is inherently different from the first because the number of unique options for the second type of variation is limited, since every option needs individual consideration. Meanwhile, the number of unique options for the first type of variation is practically unlimited.

A large advantage of having the second type of variation is that it allows for easy implementation of any type of variation that is needed, including structural variation and variation between different algorithms that might be used. A successful system will likely use both types of variation to avoid unnatural repetitiveness in the design space.

Even though a PCG system following our approach will likely never be able to fully capture the expressiveness of real-world designs, it might actually be desirable to have some repetitive patterns across our design space. If the repetitive use of modules by a system reflects the repetitive use of design elements in the real world, this repetitiveness might add to the coherence of the different objects. This can enable the system to generate objects in specific styles, which can enhance the immersion of a virtual world and increase the visual distinction between its different locations.

3.4 General module design

The content of a module can vary wildly, but generally a few key rules should be upheld when designing modules to create a well-functioning system. These rules are, in order of importance:

1. Each module must be reliable.
2. Each module must fulfill or represent a complete and meaningful task or element.
3. Each module must be as flexible as is viable within the restrictions of the other rules.

The first rule is to ensure that larger, more complex systems with many interdependent modules remain manageable, while the second rule ensures that the system can accurately resemble a human design process. An additional advantage of the second rule is that it also makes the system more easy to understand and thus more easy to maintain and extend. The third rule minimizes the number of unnecessary duplicates of modules and maximizes each module's re-usability.

More concretely, the flexibility that is referred to in the third rule means a flexibility in parameters: each module must take as many parameters as is feasible within the context of the other rules (meaning that, following the second rule, each parameter must represent a meaningful and relevant value so that it is easy to understand how the parameter should be used). However, if the calling module cannot provide meaningful values for all the available parameters, new values should be generated from within the module in which they are used as it is from within the context of this module that it is best understood what the best default values are for each parameter.

In general, modules should be designed in such a way that the probability that the module needs to be altered at a later stage is minimal. The reason for this is that there will often be a lot of interdependency between the modules: if an existing module is changed, this change might affect any of the dependent modules as it is unknown how the dependent modules utilize the altered module. This means that the effect of the change needs to be reconsidered for each of the dependent modules. Furthermore, if the original module is correct and the alteration changes the meaning of the module, it is possible that the altered module is no longer usable for one or more dependent modules. For this reason,

it is often better to create a new module instead of modifying the existing module when the existing module is correct. In case a module does need to be changed, for example because an existing system is being extended, dependencies between modules should always be documented.

If these rules are adhered to, the variety of the PCG system will grow exponentially with the number of modules that are created and new PCG systems may reuse modules of older PCG systems, increasing the overall effectiveness of this approach the more modules are created.

3.5 Module order

It is relatively trivial to ensure that an individual, independent module produces the desired type of output. However, it is more challenging to create modules in such a way that the outputs reliably fit well together while still being re-usable by different calling modules. Let us consider a general case in which two modules A and B are called from a third module C that ensures that the outputs of A and B fit together within its context. Theoretically, A, B and C can be implemented in four different ways:

- C determines the dependent variables between A and B and A and B can both be fitted to C.
- A is an independent module, C extracts the necessary variables from the output of A and B can be fitted to A.
- B is an independent module, C extracts the necessary variables from the output of B and A can be fitted to B.
- A and B are both independent modules, C repeatedly generates A and B until their outputs fit together and C can produce an output.

All four approaches may be best in certain cases, but no approach is best in all cases and often a combination of these four approaches can be used. For example, while it is generally inadvisable to use the fourth approach as it can be far more expensive than the other three, it might still be useful in cases where the outputs of A and B can usually be successfully combined and A and B only need to be repeated by exception.

In some cases, several of these four approaches are viable. While often these different approaches may theoretically lead to the same set of results, there are usually differences in the frequency of certain types of outputs and thus the effective variety because A and B may or may not be restricted by predetermined variables. In these cases, it is best to implement multiple versions of module C representing all these different approaches, so that variety is maximized. Furthermore, by having several optional modules providing different solutions to the same problem, we can make the simulated design procedures less machine-like and more closely resembling the possible design procedures of one or several different human designers. This makes it less likely that the group of generated objects feels unnaturally repetitive to a human observer.

An example of this is in the generation of *window sketches* in our PCG system (the concept of *sketches* will be introduced in the 'System' section). In many Gothic windows a pattern occurs where two or more smaller subwindows are placed inside a larger window in such a way that there is space between the arches of the subwindows and the arch of the larger window (see figure 2). This space is usually filled with a circle, or another motif that is typical for the Gothic period, in such a way that the motif touches both the upper arch and the lower arches so that the tracery can support itself. Windows that follow this pattern

FIGURE 2: Some examples of real-world windows that follow a pattern where a decorative motif is placed between an upper arch and lower subarches. In the three leftmost pictures, the motif is fitted to the geometry of the arches, in the other pictures the lower arches are fitted to the motifs. Note that where the motif becomes smaller as the arches become smaller in the second and third picture from the left, in the rightmost picture the three circles have the same size, which would be hard to recreate if the arches were generated before the circles. From left to right, these photos are taken from the following churches: Cathédrale Saint-Pierre of Beauvais, Notre-Dame of Rouen, Notre-Dame of Paris, Dom of Magdeburg, Cathédrale Saint-Étienne of Sens, Notre-Dame of Amiens.

often fall into one of two categories: either the bottom of the lower arches aligns with the bottom of the upper arch, or it does not. If it does not, it can occur that instead the motif is placed in a meaningful location. In most cases, it is impossible to locate both the lower arches and the motif in a meaningful way as they need to touch, therefore we build separate modules for each subtype of this option.

Keep in mind that these modules do not necessarily represent non-overlapping geometry, but that they may represent other types of deconstructions as well. For example, the shape of a line and the decoration along a line may also be represented by two different modules that can be combined and of which the output may or may not fit well together. Other examples are the content of a text and its font or the overall shape of a decorative knot and its internal structure.

3.6 Module parameters

All communication between modules happens through a global data-structure that maps unique sets of labels to parameter values. Various types of information can be written to and read from this data-structure by modules on any depth in the dependency tree, lending the system flexibility but also possibly making the system more complex. Entries in this data-structure may describe general attributes of the object being generated, such as the style or the material of (part of) the object, or design decisions that have previously been made. This information can be used in later modules to determine other values or to decide between different algorithms, in a similar way as how decisions of a human designer often depend on the context in which the design was made and on other previously made design decisions that often apply to the design in a more global manner.

For example, an early module may decide that the design is ‘very detailed’ and has ‘sharp edges’, while later modules may retrieve these labels to determine the local exact geometry. Usually, the key of a parameter value consists out of a parameter name and labels that pertain the scope for which this value is valid, in our system possible labels are “all”, “default” or labels identifying a specific branch in the module tree, a specific module or a specific call to a module (default values are only used if the system is deterministic, which is indicated by the parameter “is_random”, pertaining to “all”).

Alternatively, this global data-structure may be used to set or restrict the value of a specified parameter in one or more later modules so that it can be ensured that the output fits the purpose of a dependent module. Every value that may be non-deterministically generated by a module is a parameter and every parameter may or may not be set in advance. Hence, when all parameters for all module calls are set in advance by the user, the system is no longer non-deterministic. This way it is easier to debug the system and it enables a user to control the system through the input.

3.7 Context tree

To be able to set parameters for specific module calls, each path in the dependency tree that the algorithm has taken is identified by a unique *context string*. The context string is passed as a local parameter to every call to another module and every new module appends a unique string to it identifying the module (usually the module's name) and occasionally additional data for different module calls from that module, so that the context string is unique for every instance where a module is executed.

By combining the context string of a specific function call with the name of the parameter, the value of a single parameter in a single call can be set. Additionally, the value of a parameter for all calls of a module within a specific branch of the module execution tree can be set by combining the context string of the branch with the parameter name, the module identifier and the label "all". If a parameter has been set for a module or a module call, the module is forced to use the preset value, but a module may never assume a parameter is set in advance and should always contain a default method of determining its value (this method may be deterministic and non-deterministic, this method may be a module on its own and this method may also involve the calling of other modules).

Every parameter value for every module execution is stored in the global data-structure with the parameter name and the context string. The complete collection of this data we will refer to as the *context tree*. The context tree forms a complete alternative representation of the generated content, because it includes every non-deterministically determined value that the system has generated. Additionally, if it is used as input of a new iteration of the system the exact same object will be generated as the system is deterministic if all parameter values are set in advance. This makes the context tree very useful for debugging the system as it can be used to analyze the path that a system has taken and it can be used to reconstruct a bug in a deterministic manner. Furthermore, this representation may be preferable input for a machine learning system producing new inputs for the PCG system, as this representation reflects the choices that were made to arrive at a design rather than a raw mesh or geometric data.

Ideally, we would have developed the parameter system more fully to include conditional preset parameters (such as, 'the color is green if the color of the previous module is red') and allow parameters to be restricted to ranges rather than set to a single value, but due to time constraints we were not able to explore these and other options further.

4 System

In this section we will present the Gothic window tracery PCG system that we have built following our approach. First we will give an overview of the system architecture in the section 'Architecture', then we will introduce the concept of an *sketch* and discuss how we use this in our system in the section 'Sketches', then we will give an overview of all *sketch* modules used in our system in the section 'Sketch modules' and finally we will present the

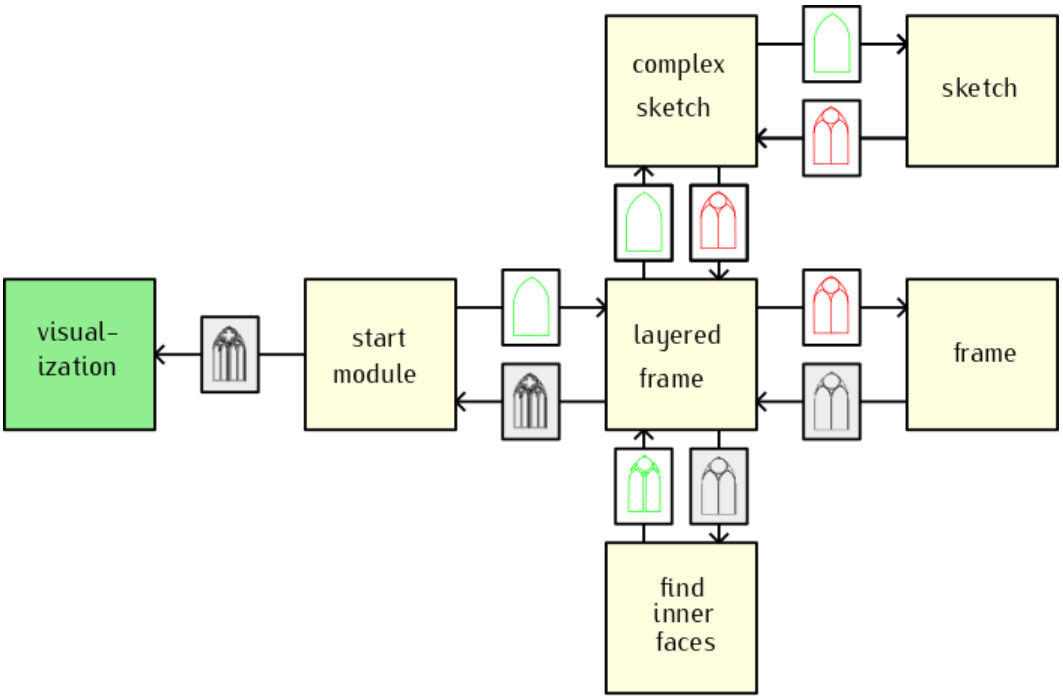


FIGURE 3: The architecture of our PCG system. The yellow boxes represent individual modules, the green box represents the calling program, which is not considered part of the PCG system. This figure displays only the uppermost modules in the module tree, there are many deeper modules that are used but that are not displayed here.

layered frame algorithm in the section 'Layered frame' and we will explain how the object is converted to a mesh in the section 'Mesh'.

4.1 Architecture

Summarily, our system is built up as is displayed in figure 3. The start module, or *Gothic window* module, generates a 3D-model of a Gothic window by first setting some global parameters and generating the outer 2D-shape of the window (as seen from the front view). Afterwards, the start module calls the *layered frame* module to produce a 3D-model of a window frame based on the predetermined outer shape and parameters. The *layered frame* module first determines the "layered" geometry of the cross-sections of the tracery and then for each layer generates all 3D-geometry associated with that layer, creating the complete window and returning it to the start module, which directly returns it to the visualization program that initialized the system. To be able to display the output using OpenGL, the visualization program converts the exact 3D-geometry used by our PCG system to a mesh after it is generated.

For each layer in the geometry, the *layered frame* module first calls the *complex sketch* module to get a *sketch*, then calls the *frame* module using the sketch and a layer of the cross-section to create a layer of the 3D-geometry and lastly calls the *find inner faces* module to find the 2D-polygons formed by the inner boundaries of the previously created 3D-geometry, which will be used as input for the next iteration. The *complex sketch* module can create a more complex sketch by iteratively calling the *sketch* module, using the output of the previous iteration as the input for the next, and the *sketch* module divides a given input polygon into smaller faces using many other, more specific sketch modules, forming a *sketch* of a tracery design.

In the following sections we will first describe what a *sketch* is and how these are generated in our system, then we will explain in more detail how the *frame* and *layered frame* modules work and finally we will briefly lay out how our models are visualized.

4.2 Sketches

In our system, most of the design decisions are made in the *sketch* module. A *sketch* is a simplification of a more complex design that directly describes the geometry that a human might see in that design, even though that geometry is often not directly represented in the data. For example, if the shape of a heart is traced by smaller circles, people will automatically discern the heart shape even though this shape is not directly part of the geometry. Similarly, if a picture of a heart is represented as a bitmap, no part of the data actually contains the geometry that forms the heart shape yet a human viewer will easily be able to see this shape. In both cases, the geometry that forms the heart shape is what we will call the sketch of the design.

In the design of 3D-objects, there are often 2D-sketches used that either form the outline of (part of) an object from a particular view point or that are visualized by decorations on the surface of the object. In our case, the sketches directly determine the shapes that are visible in our tracery from the front view. As a sketch of a design is usually independent of how that sketch is visualized, it should be generated by a separate module or set of modules. This way it is easier to combine different sketches with different visualizations and it is easier for a system to interpret a design in the same way as a human would.

In the case of Gothic windows, there are usually multiple, layered 2D-sketches to be found that are distinguished by the differing shapes of the cross-sections of the bars. Often, the inner parts of the bars forming a more global sketch seamlessly split of into bars with a different, smaller cross-section that trace a more local sketch that is entirely contained in a face of the global sketch. Because of this containment, we have chosen to use a variation of the split grammar algorithm [45] to generate these sketches. Where the original split grammar algorithm is based on axis-aligned split configurations, our algorithm is based on a predefined set of shapes that can be subdivided in predefined ways. Each subdivision is implemented as an individual module.

Where the original split grammar algorithm is based on axis-aligned split configurations, our algorithm is based on a predefined set of shapes that consist out of straight lines and circle parts. This set of shapes was largely inspired by the 'motifs' that were defined and used by Takayama [40] to generate a wide range of unique designs. Each input shape has one or multiple possible configurations of output shapes (possibly including the input shape itself) that fit into and completely fill the input shape. Similarly to a split grammar, both these configurations and the (possibly non-deterministic) manner in which a configuration is selected are predefined in the system, in our case as individual modules. In fact, any PCG system that is based on a split grammar is a system that follows our approach, as in a split grammar all rules and all shapes are manually created and the variation that a split-grammar-based system can create relies on the non-deterministic combination of various rules.

Most of the sketch modules are flexible, allowing variation in the exact input geometry as long as certain structural properties of the shapes could be guaranteed, and some of these modules are variable, generating new variation in the exact geometry or even in the configuration of shapes. The shape of each polygon and other structural information is directly communicated between modules as attributes of a *face object*. This information can be used to reliably extract the correct geometrical information from a shape while allowing flexibility in the exact geometry. Each sketch module receives one *face object* as input, which

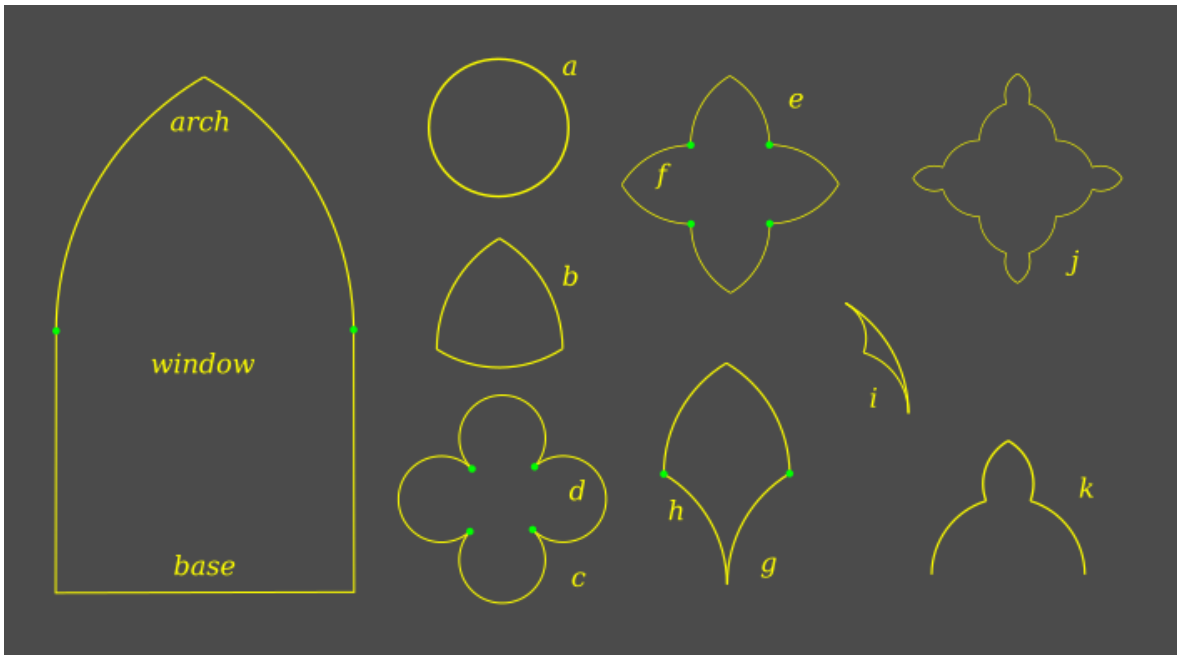


FIGURE 4: All shapes included in our system except *raster_face* and *rest*. Green dots are placed where a shape is divided into subshapes. a: *circle*, b: *convex_curved_polygon*, c: *lobe_polygon*, d: *lobe*, e: *lancet_polygon*, f: *lancet*, g: *dagger*, h: *tail*, i: *cusp*, j: *complex_polygon*, k: *complex_arch*

defines the outer bounds of the configuration, and returns the generated configuration as a vector of *face objects* that exactly fill the input shape. A *face object* may or may not consist of other *face objects* that define sub-shapes of the shape. Whether a *face object* consists of sub-shapes depends on its *shape* attribute.

4.3 Sketch Modules

In our system, the following shapes are defined: *window*, *arch*, *base*, *circle*, *convex_curved_polygon*, *lobe_polygon*, *lobe*, *lancet_polygon*, *lancet*, *dagger*, *tail*, *complex_polygon*, *raster_face*, *cusp*, *complex_arch* and *rest* (see figure 4). A *window* consists out of two sub-shapes, of which one is a *base*, which is a topless rectangle, and of which the other functions as the top of the base to create a complete polygon, this sub-shape is often but not necessarily an *arch*. An *arch* always consists of two circle parts and forms either a rounded or a pointed arch. A *convex_curved_polygon* is a convex polygon of which all edges are outwardly curved circle-parts, a *lobe_polygon* is a flower-like motif with round petals (also known as *foils* in Gothic architecture), which are the *lobes*, and a *lancet_polygon* is a similar motif with pointed petals, which are of the type *lancet*. A *dagger* consists out of a *tail* and either a *lancet* or a *lobe* and a *cusp* is a typical, thorn-shaped element. Of these shapes, *base* and *arch* only occur as sub-shapes of a *window*, *tail* only occurs as a sub-shape of *dagger* and *lobe* and *lancet* only occur as sub-shapes of either a *dagger* or their respective polygon. *complex_polygon*, *cusp*, *complex_arch* and *rest* cannot be further divided, either because such division we found unlikely to occur in real life (*cusp*), because we found it too complex (*rest*) or both (*complex_polygon*, *complex_arch*). Hence, *window*, *circle*, *convex_curved_polygon*, *lobe_polygon*, *lancet_polygon*, *dagger* and *raster_face* are the shape types that have modules to further divide that shape into a sketch.

A *circle* will always be filled with a *lobe_polygon*, which is created through repeated use of a separate module that creates a cusp on a circle-part. The number of lobes is determined by *circle_n* and ranges between 3 and 8 and depends on the radius of the circle and on random

chance. Other relevant variables are *point_up*, which determines whether the motif has a single highest point (as opposed to two) and which has a probability of 7/8 to be true, and *rosette_dr*, which equals the distance between two centers of two neighboring *lobes* in the motif divided by the radius of the *lobes*. *Rosette_dr* can have a value 2, meaning that the circles that the *lobes* are based on touch each other but do not overlap, or a value of $1 + 2/3$, meaning that there is an overlap between each pair of circles of a third of the radius. These values are generated in separate modules, and the latter two values are also used to generate *lobe_polygons* of either 3 or 4 *lobes* (known as respectively *trefoils* and *quatrefoils* in the context of Gothic architecture [40]).

A *convex_curved_polygon* will be either filled with a fitting *lancet_polygon* or with a *complex_polygon* which exists out of a combination of three cusps for each edge of the polygon. Both of these sketches are generated by separate modules and deterministic in structure. A *lancet_polygon* will either add a cusp for each circle part via iterative use of a separate *lancet* module, which in its turn reuses the *cusp* module, or it will return the input polygon. A *lobe_polygon* similarity may do nothing or split each *lobe* into two symmetric circle-parts and use the *lancet* module to decorate the motif with cusps. If altered, the inner face will be a *complex_polygon* in both cases. The *dagger* module similarly will either use the same *lancet* module or return the input.

The window module first determines how many subwindows the current sketch should produce based on the window attribute *base_n*, which determines the total number of times that the input window is vertically divided near the bottom. For the initial window shape this value is determined by the start module in such a way that the glass panels have a realistic width, favoring values for *base_n* that are easily divisible (the width of the glass panels ranges between 30 and 180 centimeters, with a preference for widths between 40 to 100 centimeters). From the value of *base_n*, *cur_sketch_base_n* is derived, which indicates the *base_n* of the sketch that will be generated by this module. Each subwindow of the current sketch will have a *base_n* that equals the *base_n* of the input window divided by *cur_sketch_base_n*. The value of *cur_sketch_base_n* is either 2 or 3, unless *base_n* cannot be divided by these numbers in which case *cur_sketch_base_n* equals *base_n*. After *cur_sketch_base_n* is determined, the window module selects one out of six other modules representing subtypes of window sketches (see figure 5). Each of these six modules will be shortly described in the following paragraphs.

The first subtype is a window without any subwindows. The returned sketch has a probability of 1 out of 7 of being the same as the input window and otherwise cusps are added to the arch of the window using the *lancet* module.

The second subtype is a window sketch that has exactly two subwindows, of which the arches are adjacent with the arch of the input window. The pointedness of the sub-arches is variable, and the face between the larger input arch and the lower sub-arches is decorated by one of five modules representing the following decorations: a third sub-arch connects the top of the input arch with the top of the two arches of the subwindows to create a *dagger* (only possible when the pointedness of all arches is equal or near equal), the largest possible circle is found and placed in the polygon (this circle will be wedged between the three arches and form the typical sketch for a window from the *high-Gothic* period), a Reuleaux-triangle (which is labeled as a *convex_curved_polygon*) connects the top of the input arch with the top of the two arches of the subwindows (only possible when the pointedness of all arches is equal or near equal), an outwardly curved diamond-shape is formed by part of the input arch and two circle-parts that are wedged between the sub-arches or a *lobe_polygon* or *lancet_polygon* is generated and placed in such a way so that the motif can be as large as possible (using the same module that is used to place the circle). The last module reuses the modules to generate the *point_up* parameter and the *rosette_dr*

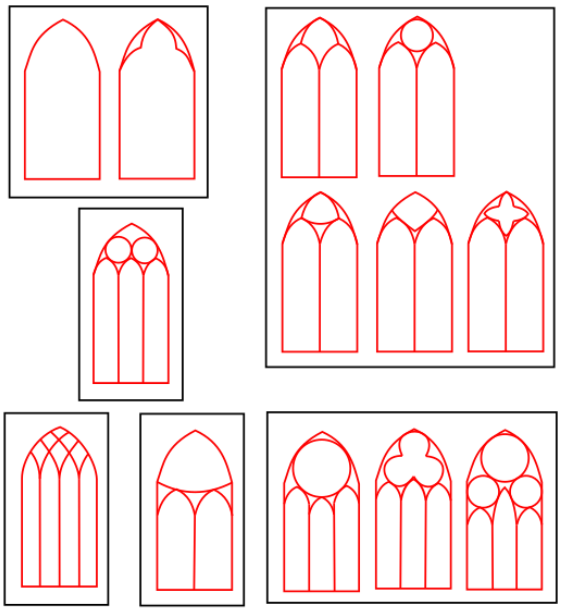


FIGURE 5: The *window sketch* modules. Each black box represents a module that is called by the initial *window sketch* module based on *cur_sketch_base_n* and each red sketch represents a module that produces that type of sketch. If a black box contains multiple sketches, the module called by the initial *window sketch* module does not directly generate the sketch, but chooses between more specific sketch modules.

parameter, but uses a different parameter named *shape_n* to determine the number of petals which equals either 3 or 4.

The third type of subwindow is similar to the second but contains three subwindows instead of two and only has one way of decoration, which is creating two maximally sized circles in the polygon between the input window and the subwindows. In this module, the shape of the arches of the subwindows is also variable.

The fourth type of subwindow creates a "raster" to decorate the window and divide it up in subwindows (see figure 6). This raster is created by moving the two circles on which the arch of the input window is based horizontally away from the window in steps that equal the width of the subwindows. A raster pattern and the arches of the subwindows are defined by the ensuing circles. The diamond-shaped faces that are not subwindows are labeled with the shape *raster_face* and can be filled with either circles or *lobe_polygons*, which are generated, placed and sized using the same modules that are used in the second *window* module. The fourth *window* module is preferred for windows that are divided in 5 or 7 subwindows, as few of the other modules can be applied to this type of window, but is also commonly used when a window is divided in 2 or 3 subwindows to provide a variation in the division of space as opposed to respectively the second and third *window* modules.

The fifth and sixth modules are differentiated from the other modules because the tops of the subwindows are usually not adjacent to the top of the input window. Instead, another module is used by both modules to determine the highest possible positioning for each arch after the decoration is placed. Similarly to the fourth window sketch module, these modules are not bound by a set number of subwindows, but in contrast to the fourth module these modules also allow the creation of exactly one subwindow. In this case the arch of the subwindow lays below the arch of the input window and the space between the two arches is decorated, making these modules especially suitable for very tall windows.

The fifth *window* module transforms the arch of the input window into two edges of a

FIGURE 6: Two examples of the "raster" pattern in Gothic window tracery. The left photo displays a plain raster pattern decorated with cusps that divides the window into three subwindows and three raster faces. In the right photo the raster pattern is used three times to divide a (sub)window into two subwindows and one raster face. The resulting subwindows and raster faces are further divided using a different pattern. The left photo is taken from the Cathedral of Norwich, the right photo is taken from the Saint Bavo's Cathedral of Ghent.

Reuleaux-triangle, after which the arches of the subwindows are placed so that they touch the bottom edge of this Reuleaux-triangle. The sixth *window* module places either a circle, a *lobe_polygon* (defined using *shape_n*) or three circles with variable sizes in the highest possible location within the window, after which the arches of the subwindows are placed. This sketch module can generate non-deterministic variation in structure, as the generated configuration is dependent on the exact geometry of the circle, the polygon or the circles. The sixth *window* module is also the only *window* module that is able to process windows of which the top is not an *arch*, though in our system these windows do not occur.

Structurally non-deterministic modules such as the sixth *window sketch* module can potentially offer far more variation than our system is capable of. However, in our implementation they are also far more likely to create bugs and therefore we had to severely limit their use to a select few cases. The reason for this is that if decorative geometry is generated and positioned inside the window shape independently of the structure of the window shape, the structural information of the sketch can no longer be derived from the input shape and thus needs to be derived for the geometry. In our implementation this often lead to errors or erroneous output where the *face objects* were ill-defined and subsequent modules were ill-applied. These errors were often caused by precision errors in either the module that positioned the geometry or the module that analyzed the geometry for its structure. We did use these modules in some of our sketch modules where the output was still fairly reliable, namely in the variations of the second window sketch module where a circle, *lobe_polygon* or *lancet_polygon* is placed between the input arch and the subarches, the third and sixth window sketch modules and the module that fills a *raster_face* with either a circle or a *lobe_polygon*. Examples of features we omitted because the were too error prone are: filling a *raster_face* with a *lancet_polygon*, further dividing the *rest* shapes and deriving structural information for the *rest* shapes.

Additionally, there are still many real-world Gothic window designs that our system is unable to generate because we did not have the time to implement the required composition modules but that potentially could have been generated if we had the time to do so. This includes many windows from the late Gothic period, as windows from this period show a

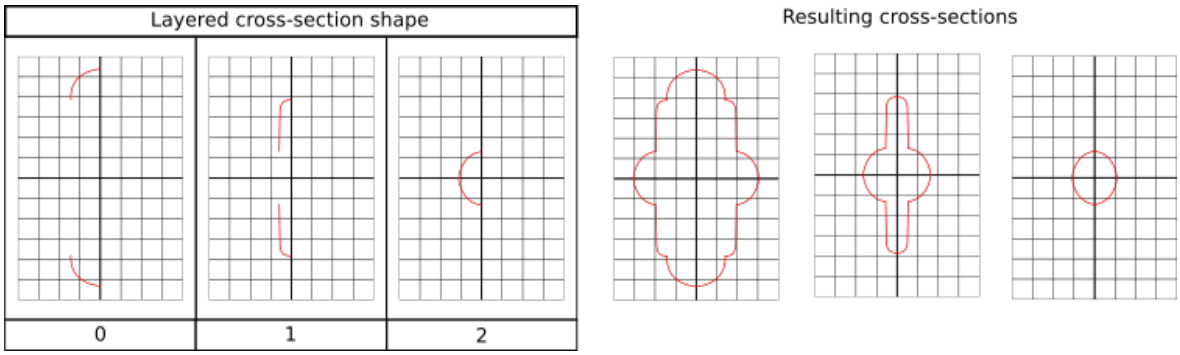


FIGURE 7: On the left hand side a "layered cross-section shape" is displayed, which consists of a vector of 2D-geometry. On the right side the three resulting types of cross-section are displayed. (The thicker lines in the grids represent the x and y-axes of the 2D-space.)

lot more structural variation where windows from the high Gothic period are structurally often very similar. We also restricted or system to only generate *lancet* windows (windows with a pointed arch) to simplify our evaluation, but as Gothic windows in other shapes often contain the same motifs our system could quite easily be expanded to include windows of other shapes as well, including rosette windows. Furthermore, because we separated the sketch of the windows as an independent module, this module could be reused in the generation of any Gothic style architectural element or object as the shapes and configurations that are included in this set of modules are defining for the Gothic architectural style is a whole.

4.4 Layered frame

The *layered frame* module combines the sketches in the design of a single window by creating the layered frame. It receives a 2D-polygon determining the outer shape of the window and a number of attributes describing the window from the starting module. The algorithm first determines the number of layers, taking into consideration *base_n*. The number of layers equals ${}^2log(base\_n)$ plus possibly an additional layer for subwindows that do not divide the window vertically and another possible additional layer for a different visualization of the cusps (the decorative "thorns" that are typical of Gothic window tracery). Based on the number of layers, an appropriate vector of 2D-shapes is selected that determines the cross-section of every part of the tracery in the manner as is displayed in figure 7. The number of shapes in the vector equals the number of layers and each shape describes a part of the left half of the most global cross-section, so that the last shape in the vector on its own forms the smallest cross-section. The cross-section can also be generated non-deterministically, but we chose not to do so for our system because the windows would be evaluated from the front view only, hence we prioritized types of variety that would be most visible from that angle.

Then the algorithm does the following for each layer, starting with the most global:

1. Generate a sketch for all of the available input polygons.
2. Call the *frame* module to create the 3D-geometry for each face of each sketch for this layer, using the cross-section piece that corresponds to the current layer.
3. Create a *face object* for the inner boundary polygon for each face of each sketch using the offset created by the cross-section piece (and the attributes of the original polygon), and use these as the input polygons for the next iteration.

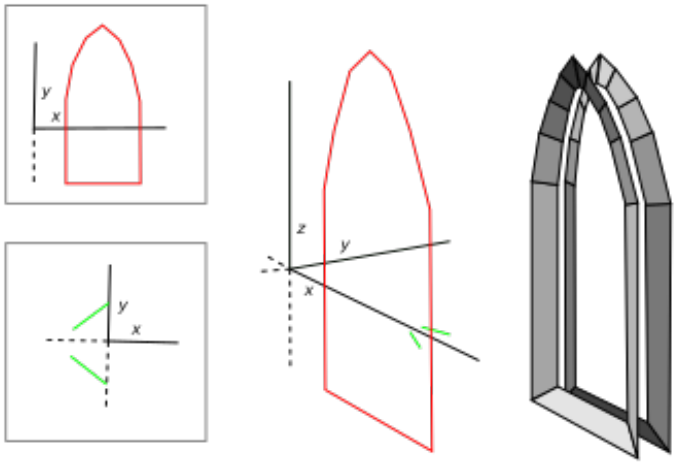


FIGURE 8: The frame module takes a 2D-sketch (red) and the 2D-geometry of a cross-section (green) as input and produces 3D-geometry (here visualized as a mesh).)

The 3D-geometry is created by the *frame* module, which takes 2D-geometry for the sketch and for the cross-section as input (see figure 8). The origin of the cross-section piece will follow the edges of each face while the positive y-axis of the cross-section piece will be aligned with the away-axis in the 3D-space. We use a right-handed axis system where the positive x-axis corresponds to the right direction, the positive y-axis corresponds to the away direction and the positive z-axis corresponds to the up-direction (as seen from the initial viewpoint). This means that the axes of the cross-section geometry are parallel with the x and y-axes in the 3D space if the direction of the sketch edge is parallel to the positive z-axes. The positive y-axis of the sketch corresponds with the positive z-axis of the 3D-space, making it so that the 2D-sketch is projected on a plane parallel to the initial view plane and thus that we view the generated window frame from the front.

As every face in the sketch is defined in a counter-clockwise manner, each inner edge of the sketch is defined twice: one time in each direction. As a consequence, each inner edge will be used twice to generate 3D-geometry and the cross-section geometry will be effectively mirrored along the y-axis. Therefore, the cross-section geometry given to the *frame* module should only display the left half of a cross-section layer. This way the cross-section is automatically completed as it is mirrored, and the inner layers seamlessly align with the outer layers as only the inner half of a bar is generated for the boundary edges of each sketch.

While the y-values of the cross-section directly correspond to the y-values of the 3D-space, the x-values of the cross-section correspond to the normal of the counter-clockwise directed edges. This means that a straight edge will be translated by its normal multiplied with the x-value of the cross-section and for a circle-part it means that the x-value will be added to its radius. As each edge is individually offset in this manner, it often occurs that the endpoints of these edges no longer align after these transformations. To fix this, the endpoints of the relevant edges are moved across their geometry accordingly.

4.5 Mesh

In our PCG system, the 3D-geometry is defined as quads of which the surface is defined by (the combined interpolation of) its edges. Adjacency data is not stored, instead it is assumed that edges that are geometrically near each other can be merged into one edge. This is done to simplify the separate generation of the 3D-geometry for the different layers and for the

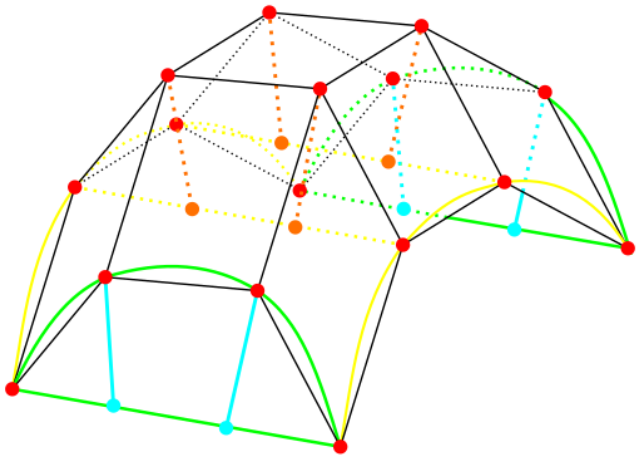


FIGURE 9: First the offset vectors for each inner vertex on two opposed sides are calculated (blue), then they are linearly interpolated to create the inner offset vectors (orange) that are subsequently used to create the inner vertices of the surface. (Dashed lines indicate lines behind the surface.)

different faces of each sketch, as in this manner 3D-patches of different layers or different sketch faces can be positioned next to each other without needing to adjust the adjacency data. Furthermore, the exact nature of the 3D-geometry makes it more easy to interpret and more efficient to process during the generation phase.

After the window is generated, this representation is converted into one that can be visualized with *OpenGL*. First, each patch is individually discretized into a quad-mesh of adequate precision (see figure 9). Each vertex along an edge can be defined by a value t that equals 0 for the starting point of the edge and 1 for the end point of the edge. For a straight edge, t equals the distance from the starting point divided by the length of the edge, and for a circle or ellipse t equals the difference between the angle of the vertex and the angle of the starting point, divided by the angle difference between the starting point and the endpoint of the edge. For each vertex on two opposing edges of the squad, a corresponding point is defined on a straight line between the two endpoints of the edge using t. The offset vector from this point to the vertex is calculated and then rotated in such a way that the average normal of the two endpoints aligns with the positive z-axis. Then, the inner vertices on the surface of the patch are defined using a linear interpolation between the two corresponding offsets, rotating the resulting offset in accordance to the interpolated edge normal and adding the offset to the corresponding point on the straight line between two vertices on the other pair of opposing edges.

After a mesh is generated for each patch in the model, the separate meshes are joined together in a single mesh in a process that we call “stitching”. First adjacent patches are found by comparing the alignment of the edges in the exact geometry, then the vertices of the different patches are ordered according to their place on the exact edge. Finally, the vertices of other patches are added to the corresponding mesh faces of the current patch, so that there are no holes between adjacent patches. Vertices that are too close together are merged. After the patches are stitched together, the edges between the original patches are only noticeable in the sudden non-alignment of the inner edges between patches, but they are not noticeable in the rendered surfaces.

We implemented our system in C++ for Windows 10 using the standard library, the *GLFW 3* [1] library for visualization and the *Eigen* [2] library for matrix calculations.

5 Evaluation method

We found that in most previous work concerning PCG methods either the evaluation of these algorithms was very subjective, the test group evaluating the algorithm was very small or the algorithm was not evaluated at all. The algorithms that were objectively evaluated were often tested on concrete qualities like the difficulty or playability of levels so that the evaluation could be done through simulation. Other PCG methods were evaluated through user studies in which content designers learned and then evaluated the system. These evaluations often took considerable time of the test subjects, who often were specialists in the use of design software, hence these evaluations were often conducted by only a small number of participants and were of a qualitative rather than a quantitative nature. Finally, there were also systems that were evaluated by the public in real-time through the distribution of a fully developed game. While this might be the most reliable way to evaluate a real-time PCG system, it is relatively costly as the game needs to be developed and it might not be equally applicable for different types of content.

As our content is aesthetic, it cannot be reliably evaluated by a machine and we did not have the time and resources to either hire experts to evaluate our system or to develop and distribute a real-time game. Therefore, we created a different, relatively easy and inexpensive evaluation method so that we can feasibly test our generator within our time and resource constraints. As we did not find a similarly objective evaluation method in previous work that was suitable for aesthetic content, we will present this evaluation method as our secondary research contribution.

Our evaluation method objectively scores a generator along two different measures, namely the quality of the individual models and the overall perceived variation of the generated output. The quality is tested by presenting the generated models next to models from a golden standard and asking human observers to identify the models belonging to the golden standard. The variation is tested by asking observers to identify previously seen models among new models in a memory task. Both tests are preformed on the same data set so that an evaluation along both measures can be done for each generated object. This enables us to analyze which pictures preform better or worse along which measure which might give us an insight in what type of features influences the performance of the designs.

As the most interesting parts of our design could be captured from the front view, we decided to let the participants evaluate a pre-generated data set of 75 pictures of our models instead of showing the models directly through a 3D interface. This data set is included in appendix A. To maximize our data per picture, we let each participant evaluate the entire data set divided over the two tests. Each participant did both tests, but the variety test needed to be done first because it requires the participant to memorize our pictures and the quality test cannot use pictures that are previously shown. Therefore, before the start of each survey the generated data set was randomly divided into a set of 52 models for the variety test and 23 models for the quality test.

A large advantage of our method is that it is relatively simple and the test can thus be distributed via an online survey (implemented in HTML and PHP and published on a personal web-space), enabling us to gather sufficient data despite corona restrictions. Also, we made sure that our data set was small enough so that the survey did not take much more than five minutes to complete. We considered this essential for our evaluation method, as people are much less likely to voluntary participate if doing so takes a lot of their time.

To ensure that our data-set is a fair reflection of the output of the system, we included all generated objects in our test set until we had 75 examples, with the exception of cases where the output was erroneous. We considered a model to be erroneous if it included features that our system should not be able to generate. This was the case for 7 out of 82 pictures

that we generated to create the 75 pictures for our test set. We believe that most of our erroneous output was caused by the precision errors occurring in the sketch modules where the structure of the output needs to be derived from the geometry. The reason for this is that we have had many of such errors while developing our system, and this believe is further substantiated by the fact that four out of the seven erroneous modules display misplaced cusps, that should have been placed on the inner edges of *lobe_polygons* but that are instead placed on surrounding edges. All our erroneous models are included in appendix B.

5.1 Quality

The quality test consisted out of 23 questions for which 2 pictures were shown per question, one from the generated set and one from a golden standard. The participant was asked to identify the picture from the golden standard. In many real-world cases, the golden standard will be a set of objects designed by an artist to be used in a specific virtual world and the aim of the generator will be to generate objects that fit the aesthetics of that same virtual world. This way it can be tested whether the generated objects confirm to the intended style of the generator.

As we did not have a virtual world or an experienced designer for our research, we chose to manually model a selected set of real-world Gothic windows using our generator as an engine. This way we can ensure that our generated models fit the style of modeling of the golden standard and that any differences we measure originate in differences in the design of the traceries. Hence, to generate good quality output in our experiment, the design of the Gothic window traceries should be indistinguishable from a selected set of real-world Gothic window designs. We used *Google Street View* [3] to find real-world examples of Gothic windows from many different Gothic churches and cathedrals across Europe. This way we could easily collect a large sample of real-world designs that we can reliably certify to be created during the Gothic time-period. From this collection, we selected the designs that did not contain any features that our generator was unable to generate and that reflected the expressiveness of our generator as fully as possible, so that we did not have to artificially restrict the variety of our system in order for our generated output to resemble the golden standard. All golden standard pictures with their real-world sources are included in appendix C.

The participant is repeatedly presented with two pictures that are displayed next to each other (see figure 10). One picture is part of the golden standard and one picture displays a generated model. The participant then has to pick the picture of which they believe that it belongs to the golden standard. In our research, we did that by asking them the following question: “Which window is modeled after a real world design?”. Both pictures must be new, meaning that the participant has not seen them before, and the objects in all pictures must be displayed in a similar way, meaning that the scales and perspectives shown must be the same for all pictures. In our experiment, this was ensured by picturing all windows from the initial viewpoint, including the entire view. For each question, both pictures are randomly chosen from the sets of not previously seen pictures and which picture is shown in which place is randomly chosen as well.

5.2 Variety

As the true variety that our system can produce is excessively large and as most people will be able to recognize these differences if two pictures are shown next to each other, we decided to evaluate the variety of our system through a memory test. This also better reflects how the generated objects would be experienced if they were used in an actual game or

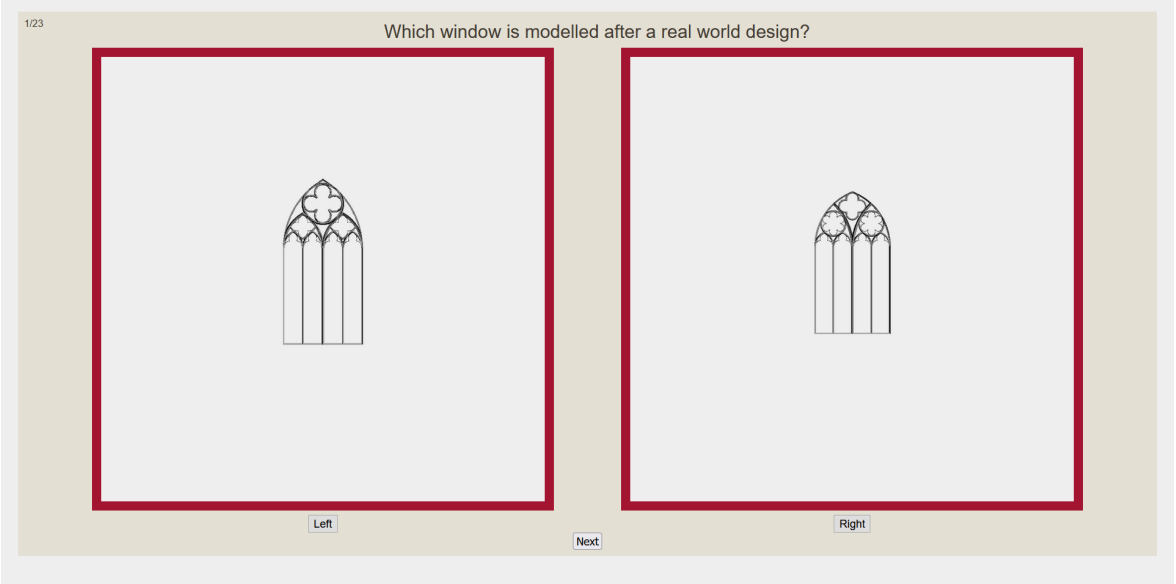


FIGURE 10: Screenshot of the quality test. Participants are forced to select either "Left" or "Right" before they are able to go to the next question.

other virtual setting, as objects belonging to the same category are generally used throughout a virtual world and not in just one location. We will randomly show our participants either a new picture or a picture that they have seen before and we will ask them whether they believe if they have previously seen this picture. If we can statistically prove that the participants are more likely to identify a new picture as new than they are to identify a picture they have seen before as new, we can conclude that the output of our system provides more perceived variety than copies of previously seen objects.

The variety test is as follows: in 75 questions, 52 unique pictures are shown and 23 copies of a previously seen picture. One picture is shown per question, and the observer needs to decided whether they have previously seen that picture or not (see figure 11). The more similar the different pictures are, the higher the probability that an observer will falsely assume that a new picture is a copy of a picture that they had previously seen.

For most questions, there is a random probability of 1/3 that a copy will be shown. This ratio was chosen to maximize the number of pictures being evaluated per observer while keeping the survey short enough that our participants would be encouraged to complete both tests. There are two exceptions for this ratio. The first two pictures are always unique because a copy never occurs immediately after the original picture has first been shown. The other exception occurred near the end of each test, as occasionally the system was forced to either show only unique pictures or copies as there were always exactly 52 unique pictures shown in 75 questions. After a new unique picture is shown, it is removed from the set of new pictures and placed in a set of previously seen pictures. When it has been decided whether a unique picture or a copy is shown, a picture is selected at random from the appropriate set. It is possible that a picture is shown more often than twice, this is more likely at the beginning of the test as the set of previously seen pictures is still relatively small. As this becomes less likely as more unique pictures are shown and as all pictures are picked at random, we do not think the possible repeated display of copies significantly affected our results.

Existing literature on visual recognition memory [4, 21, 36] showed us that most people should be easily able to recognize a picture they have previously seen. In one experiment in which people when forced to choose between a previously seen and a new picture where

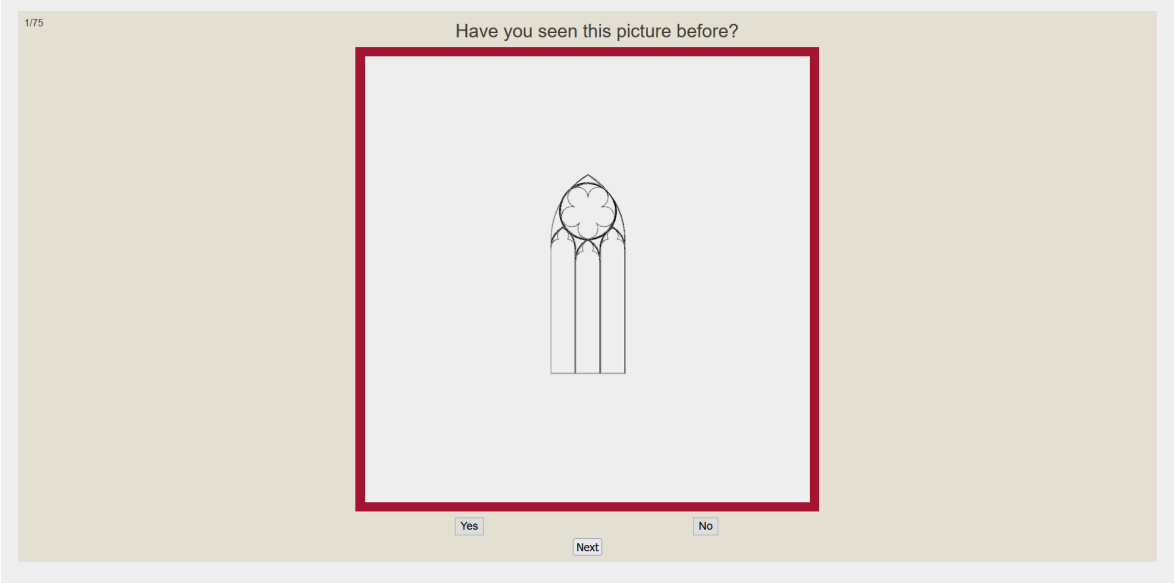


FIGURE 11: Screenshot of the variety test. Participants are forced to select either "Yes" or "No" before they are able to go to the next question.

the different pictures showed 'topical similarity', people were able to recognize up to 2500 similar pictures with 88% accuracy [4]. Other experiments on this topic concerned similarly large numbers of pictures, and visual recognition memory was thus described as 'virtually limitless' in capacity. [21, 36].

For this reason we expect the participants to usually be able to identify the copies correctly and that mistakes that are made in the recognition of the unique pictures are due to a similarity in the pictures and not due to the memory of the participant. We will compare each participants performance for the unique pictures with their performance for the copies to deduce their true ability to identify the unique pictures.

6 Results and discussion

We approached people via mail to ask them to fill in our survey and we also requested them to forward the mail to friends and family. Therefore, we do not know for certain who our participants were, but we consider it likely that most of our participants were Dutch and that our participants were on average younger and higher educated than the general public. However, we do not think these biases significantly influenced our results so we assume that our participants are representative of the general populace.

There were 77 people who completely filled in the variety test, out of which three people assumed that every picture was a unique picture (the number of questions and the number of the current question remained visible during the test) and out of which seven people assumed that the first picture was a copy (in the text explaining the test it was explicitly mentioned that the first picture is always unique). We considered the remaining 67 people for our results. There were 75 people who completely filled in the quality test, who we all considered in our results. Session identifiers were renewed between the two tests, so we were unable to trace back the people whose results were deemed unsuitable in the first test. However, as the task of the second test was different and perhaps easier, people's understanding of the first task might not necessarily be reflected in their understanding of the second task.

Confusion matrix of the real against the perceived picture category for all participants combined					
$n_o = 67$, $n_q = 75$	Perceived →	Unique	Copy	Prob. seen as unique	Recall, Specificity
Real ↓	$n_o \times n_q = 5025$	$P_U = 2864$	$P_C = 2161$		
Unique	U = 3484	$U_U = 2287$	$U_C = 1197$	$U_U/U = 0{,}656$	
Copy	C = 1541	$C_U = 577$	$C_C = 964$	$C_U/C = 0{,}374$	$C_C/C = 0{,}626$

TABLE 1

6.1 Variety

Table 1 shows a summary of all the data gathered with the memorization task (with exclusion of the data from the previously mentioned omitted participants). The variable n_o is the number of observers and n_q is the total number of questions each observer answered. U is the number of "unique" pictures, which are pictures that have not previously been seen before, and C is the number of "copies", or pictures that the observer has seen before. U_U, C_U and P_U are respectively the numbers of unique, copied and total pictures that were perceived to be unique and similarly U_C, C_C and PC are the numbers of pictures that were perceived to be copies. Additionally, the probability of either type of picture to be perceived correctly is shown (U_U/U and C_C/C) as well as the probability of pictures from either set to be seen as unique (U_U/U and C_U/C).

On average, a unique picture that was generated by our system had a probability of 0,66 to be identified as unique. This is the *subjective variety* of our system. While the subjective variety might be influenced by factors of the output other than the actual variety, such as visual complexity of the object, it can be argued that these other factors should actually be taken into account when discussing the variety of aesthetic output. This is because the main value of aesthetic content is its directly apparent value, so if the generated content seems more variable to the average observer than it actually is this apparent variety might be equally or more valuable than actual variety that helps the observer to recognize unique objects but that is perhaps not as noticeable. However, this subjective variety is also more susceptible to factors outside of the evaluated objects, such as possible expectations of the participants on the frequency that copies will occur or tendencies towards one of the two options created by the culture of the observers or the design of the test.

The *objective variety* of our system is calculated by subtracting the probability that a copy is falsely identified as unique from the probability that a unique picture is identified as unique, following the formula $U_U/U - C_U/C$. As C_U plus C_C equals C, the resulting value is equal to the *informedness*, which is the *sensitivity* plus the *specificity* minus 1. The *sensitivity* is also known as the *recall* and is calculated as U_U/U, while the *specificity* is the recall of the negatives which in our case is calculated as C_C/C. In figure 12 the sensitivity and specificity are plotted against each other. The chance line shown in this figure equals the equation $U_U/U + C_C/C = 1$. This equation is true for an observer when they perceive a picture to be unique with the same frequency for the copies as for the the unique pictures. As can be seen in the figure, there is no data that is placed significantly below the chance line. This is expected, because it means that no observer preformed significantly worse than they would by chance.

The informedness ($Sensitivity + Specificity - 1$) is thus an estimate of the probability that the observer identified a unique picture as unique because he recognized the picture as such, instead of identifying it as unique as a random guess or identifying it as a copy. This value can range from 1, in which case all pictures were correctly identified, to -1, in which case all pictures were incorrectly identified. If the value is 0, the unique pictures were

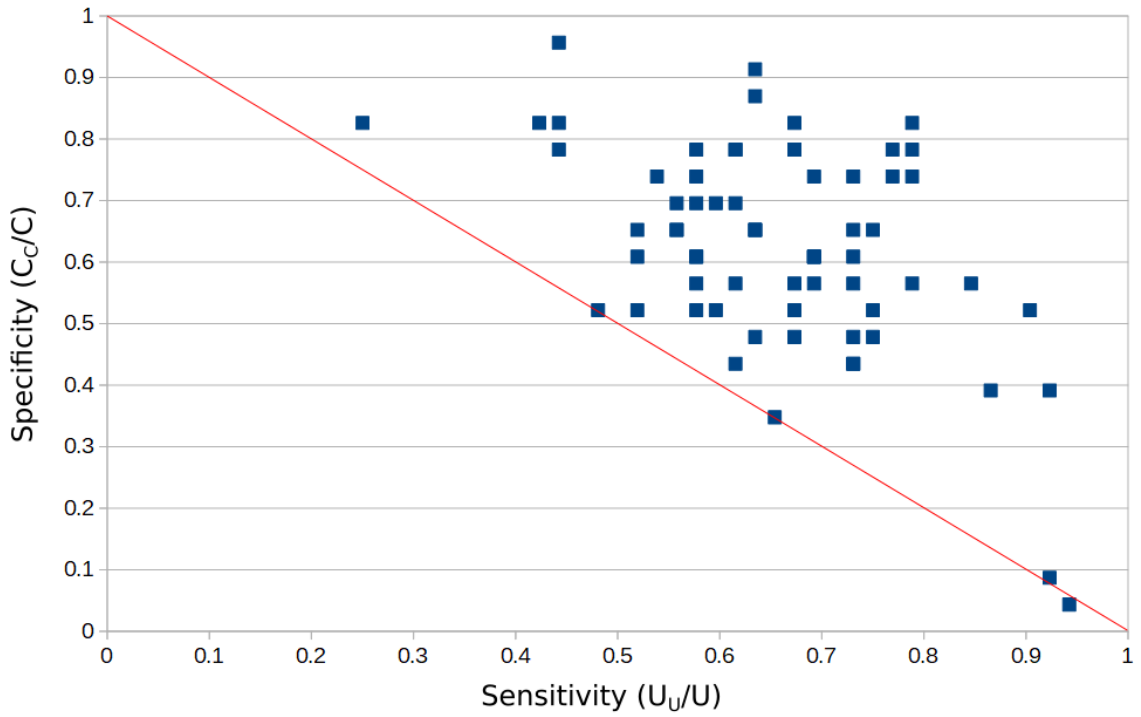


FIGURE 12: Each dot represents an observer. The closer a dot is to the top-right corner, the better the observer was at discerning the unique pictures from the copies. No observer is placed significantly below the chance line, which equals $U_U/U + C_C/C = 1$.

equally likely to be identified as unique as the copies. The objective variety of our system is 0,28.

Other statistics commonly used to a score a discriminator based on a confusion matrix, such as the F-score, the phi coefficient and the accuracy, were less suitable to measure the objective variety. Unless the number of copies is large enough so that the precision of U ($U_U/(U_U + C_U)$) can approach 0 while the recall of U is 1, the F-score will be (often significantly) higher for a test in which nearly all pictures are identified as unique than for a test in which nearly all pictures were identified as a copy. Accuracy has a similar problem as it divides the total number of correct predictions by the total number of predictions, therefore over and underestimation of the number of unique pictures is only punished equally when the number of copies and unique pictures are equal. The phi coefficient does not have this problem as it summarizes all relations in a confusion matrix symmetrically and returns a correlation score that is independent of the sizes of the different categories. However, this correlation score is more abstract than the informedness and thus harder to interpret, while the two scores seem to strongly correlate for our data (see figure 13).

Both the subjective variety and the objective variety can be calculated per participant, per picture or over the system as a whole. The first two methods respectively give more insight in how different people might experience the variety that the system generates and what type of pictures adds more or adds less to the overall variety of the system. For the subjective variety the average score per observer is 0,656 with a standard deviation of 0,128 and a t-value of 9,99 (measured against the chance value of 0,5), and the average score per picture is 0,646 (the average per picture is different because not every picture is used for the variety test equally often) with a standard deviation of 0,199 and a t-value of 6,35. For the objective variety the average score per observer is 0,282 with a standard deviation of 0,151 and a t-value of 15,3 (measured against the chance value of 0), and the average score

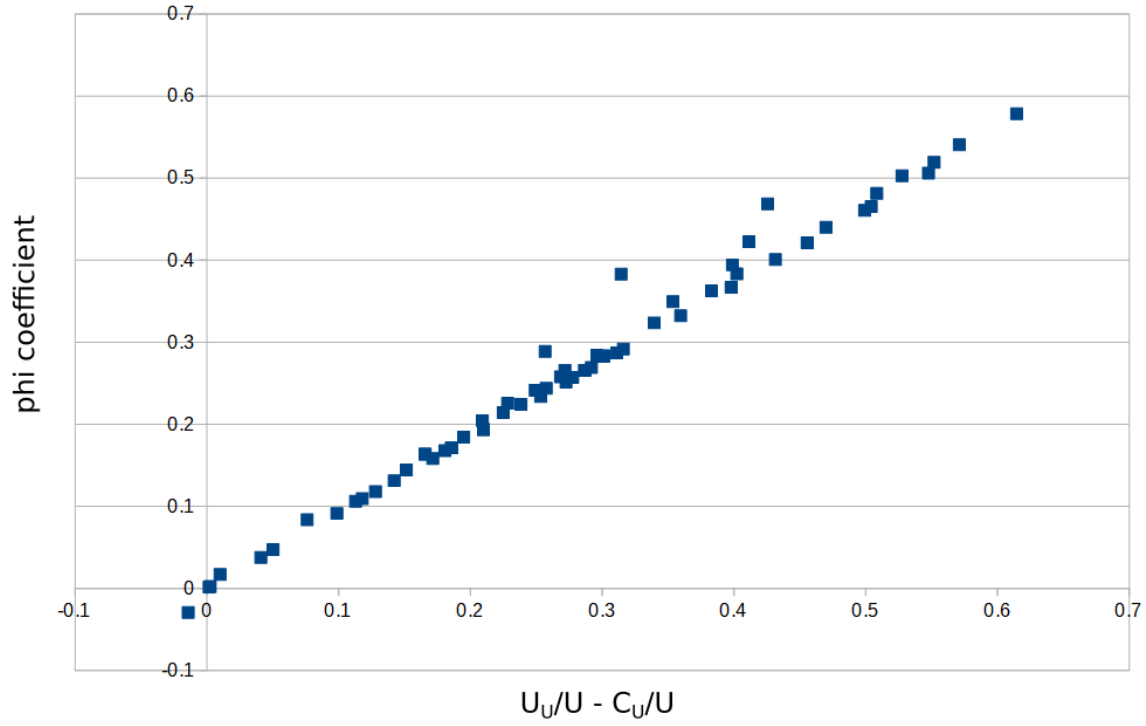


FIGURE 13: Scatterplot of the phi coefficient and the informedness of *U* calculated for each person.

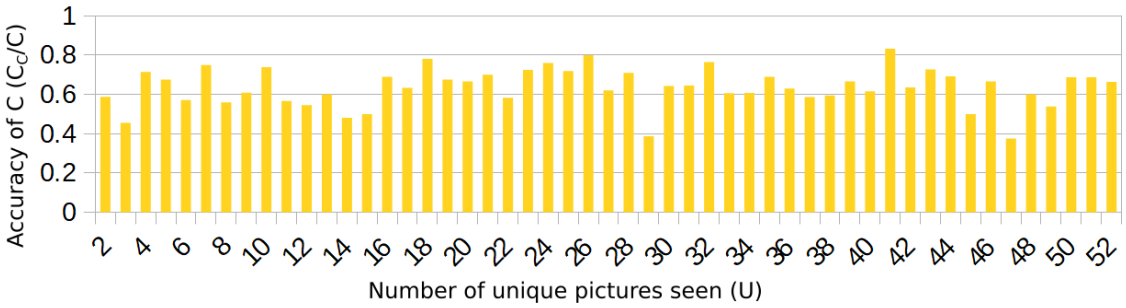


FIGURE 14: Average accuracy of C for each number of unique pictures that the participant had seen at the moment they identified the picture. No trend could be found.

per picture is 0,291 with a standard deviation of 0,142 and a t-value of 17,8. As there were 67 participants included in this test, the probability that these results were due to chance is negligible for both subjective and objective variety per participant (less than 0.0005), proving that our generator creates significantly more observed variety than copies of previously seen pictures.

A notable detail is that people were significantly worse at identifying the copies than we expected. However, when we look at the average accuracy for *C* for each number of pictures that the participants needed to keep in memory at the time they identified the picture (see figure 14), we see that this accuracy remains constant as the capacity grows. This makes it unlikely that this difficulty to correctly identify copies can be explained by the number of pictures that the participants needed to remember or the time between the moment that a picture was last shown and the moment that the copy is shown, as a downward trend would be expected if either of those causes were true.

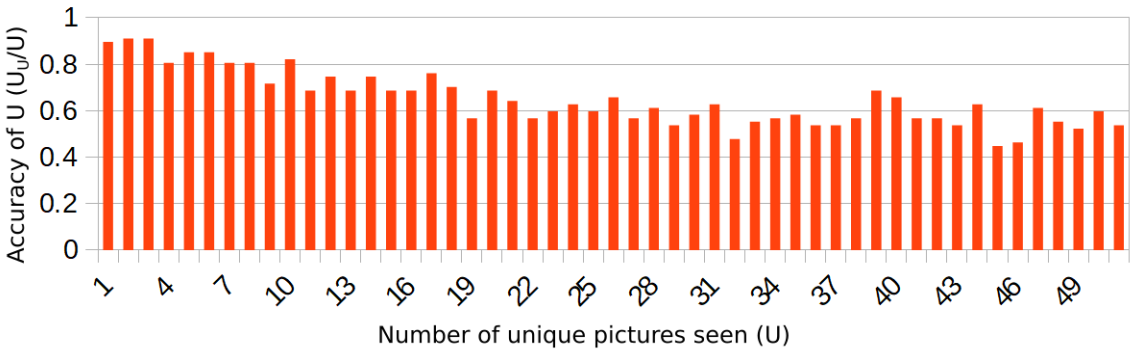


FIGURE 15: Average accuracy of U for each number of unique pictures that the participant has seen at the moment they identified the picture. The visible downward trend seems to be asymptotic.

Instead, it is possible that people did correctly recognize repeated pictures as possibly being copies but underestimated their own ability to accurately identify the copies and therefore incorrectly identified some copies as unique. A reason for this confusion could be the visual complexity of the pictures themselves: people might have assumed that they missed differences in details when they did not. Another explanation could be that the participants expected there to be more copies than there actually were as no information was provided on the frequency or the number of copies. Furthermore, in the experiment conducted by Brady et al. [4] (in which the participants showed an accuracy of 88% in a visual memorization and recognition test of topically similar pictures), participants were forced to identify the previously seen picture out of a pair of a previously seen picture and a new picture. Contrarily, we showed only one picture per question and asked our participants whether they had previously seen the picture. This might have made our test more difficult than the test of Brady et al. [4], as a participant might feel uncertain about whether they have seen a picture or not. It might be more difficult to judge whether a picture is a copy than it is to judge which picture out of two is most likely to be a copy. On the other hand, one might argue that the results of our approach better reflect how the variety of the windows would be judged if they were used in a virtual world, as the approach of Brady et al. [4] would be less suited to measure the subjective variety.

The development of the accuracy of C is an interesting contrast with the development of the accuracy of U, which starts higher but decreases as the number of previously seen unique pictures grows in a seemingly asymptotic manner (see figure 15). An explanation for this could be that our participants felt more certain that a unique picture was a unique picture than that a copy was a copy as long as the new picture was sufficiently dissimilar to any previously seen pictures. Hence, as the number of previously seen pictures grows, the probability of seeing a unique picture that is dissimilar to all the previously seen unique pictures becomes smaller and the probability of a unique picture being correctly identified decreases. As it becomes less likely that a unique picture is sufficiently dissimilar to the previously seen unique pictures as more true unique pictures are shown, the probability that the picture is added to the set of dissimilar pictures with which each new picture is compared decreases as well. This could explain why the downwards trend appears to be asymptotic as opposed to linear.

Regardless of the cause of this downward trend, it shows us that our evaluation method is strict enough to measure the limits of our system. It also shows us that our average variety values would have been higher if we included less than 52 unique pictures in our variety test. This means that the results of different variety tests with different absolute numbers for U cannot be reliably compared.

6.2 Quality

In contrast to the variety test, during the quality test two pictures were compared and identified in one question, one picture out of both categories. Therefore, the data cannot be compared in a confusion matrix as it is identical for both categories: if a generated model was wrongly identified as being a real-world Gothic design, a real-world design was simultaneously identified as being a generated model and vice versa. The quality score is calculated as G_M/G, in which G is the total number of times a generated model occurred in the quality test and G_M is the number of times a generated model was perceived to be a manual model. Hence the quality score is the frequency in which a generated model is viewed as manual. The (rounded) overall quality score of our system is 0,5061, in which G_M equals 873 and G equals 1725 (which equals the 75 observers who completed the quality test times the 23 questions for each observer). We will use a normal distribution to estimate the probability that the observed quality score is significantly different from 0,5 (the score that is approximated if each question is answered at random).

Similarly to the variety scores, the quality score can be evaluated per participant and per picture. When evaluated over each participant participant, the average probability of a picture being identified correctly (G_G/G) was 0,4939 with a standard deviation of 0,0936. The distance between the observed average and chance (0,5) is 0,0061, which equals 0,065 standard deviation. If we round this value up to 0,07, the corresponding z-value is 0,5279. This means that the probability that the distance from the average to 0,5 is significant is $(0,5279 - 0,5) \times 2$, which equals 0,0558 or 5,58%.

Likewise, when calculated per picture, the average probability of that picture being identified correctly was 0,4942 with a standard deviation of 0,1632. Here, the distance between the average and chance is 0,0058 or 0,0355 standard deviation, which rounds up to 0,04 and corresponds to a z-value of 0,5160. The probability that the difference we found is significant is thus 0,032 or 3,2%.

The likelihood of the observed average scores being significantly different from 0,5 is quite small. Additionally, our generated pictures were on average less likely to be identified as being generated than our manually created pictured. Therefore, we conclude that our participants were unable to distinguish between our generated pictures and our golden standard, and thus that our generated pictures were of sufficient quality.

6.3 Pictures

In the tables 2, 3 and 4 the six highest and lowest scoring pictures are shown for each of the measures of subjective variety (U_U/U), objective variety ($U_U/U - C_U/C$) and quality (G_M/G) respectively. Beneath each picture first the picture number is presented followed by the score. For the subjective variety and quality scores the score is both presented as the fraction of U_U or G_M divided by U or G respectively and as a single value.

As can be seen from table 2 and 3, the lowest scoring pictures of both subjective and objective variety belong to the same group of structurally similar windows. The structure that this group is based on is very common for windows from the high-Gothic period and was therefore one of the first structures that we implemented, many of the other templates we many added to increase variation.

On the other hand, the highest scoring windows for subjective and objective variety are very different and, as we can see in figure 16, for most windows there is no strong correlation between the two scores. From the first table you can see that the windows scoring highest at subjective variety, which are the windows that were most frequently identified as unique regardless of whether they were used as unique pictures or as copies, are generally quite large windows with a lot of detail, while the windows that scored highest for objective

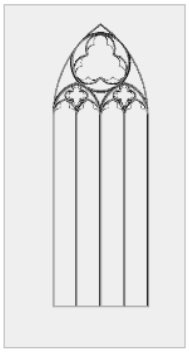
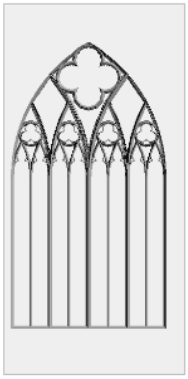
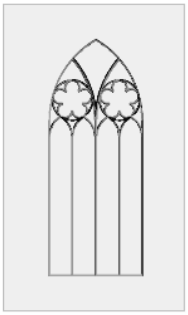
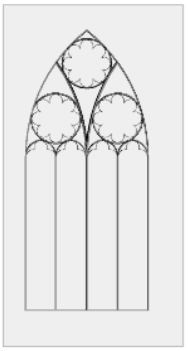
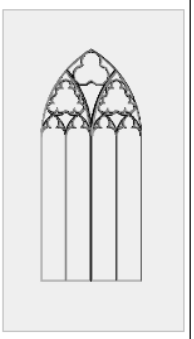
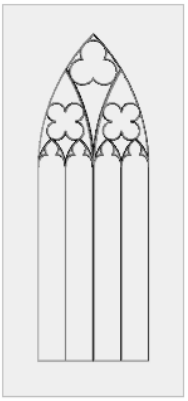
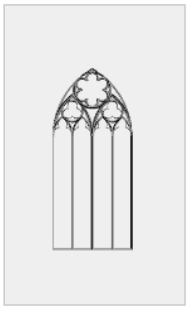
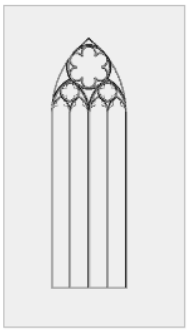
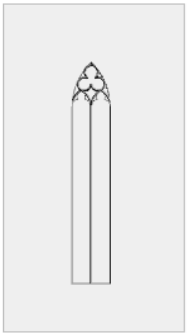
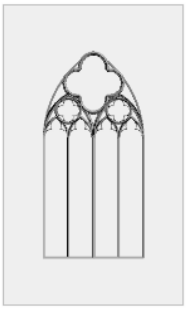
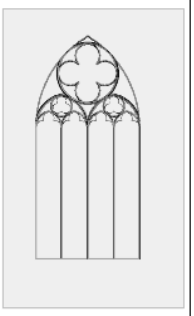
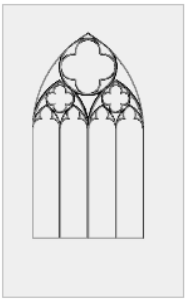
Subjective variety												
											
63		12		22		8		35		11	
52/55	0,945	40/43	0,930	40/43	0,930	39/42	0,928	39/42	0,928	41/45	0,911
											
51		49		45		38		19		53	
17/45	0,378	12/41	0,293	13/45	0,289	12/43	0,279	10/43	0,233	9/44	0,205

TABLE 2: Below each picture, from top to bottom and from left to right the following values are displayed: picture number, number of times the picture was correctly identified as unique (U_U) divided by the number of times the picture occurred as unique in total (U) and the subjective variety score as a single value (U_U/U). The pictures in the top row are the six pictures with the highest score, the pictures in the bottom row are the six pictures with the lowest score. The pictures are ordered from left to right by the decrease in the score. There are no differences between scores that have been lost due to rounding.

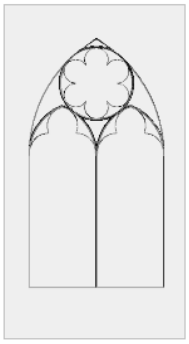
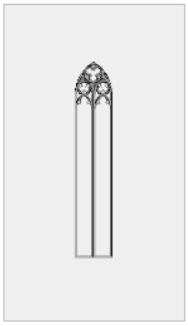
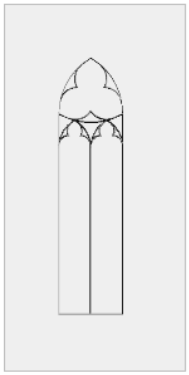
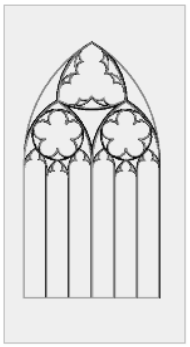
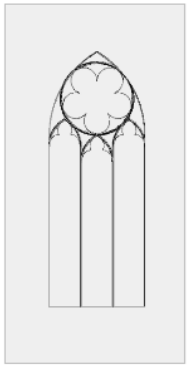
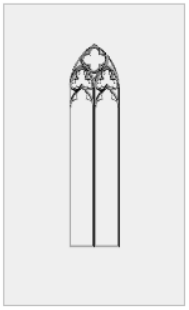
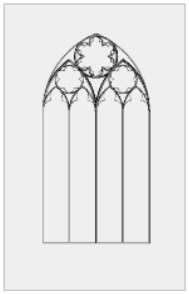
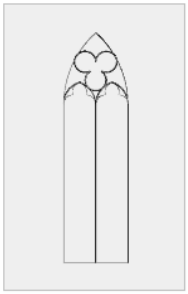
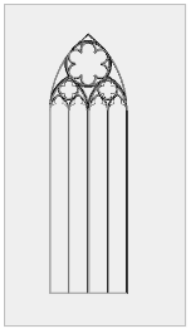
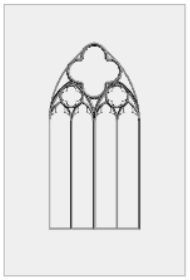
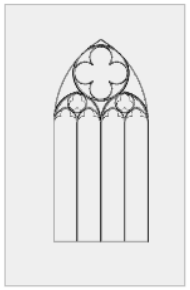
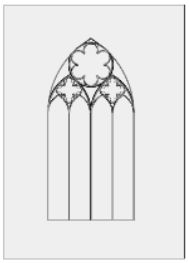
Objective variety					
					
21	65	18	24	15	36
0,570	0,567	0,515	0,511	0,510	0,489
					
46	33	49	38	19	13
0,092	0,086	0,056	0,056	0,033	-0,248

TABLE 3: Below each picture, from top to bottom the following two values are displayed: picture number and the objective variety score (calculated as $U_U/U - C_U/C$). The pictures in the top row are the six pictures with the highest score, the pictures in the bottom row are the six pictures with the lowest score. The pictures are ordered from left to right by the decrease in the score. There are no differences between scores that have been lost due to rounding.

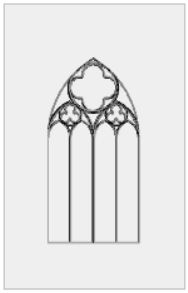
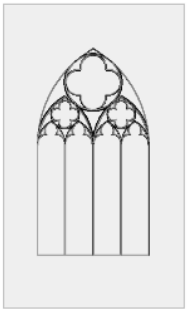
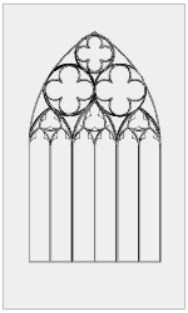
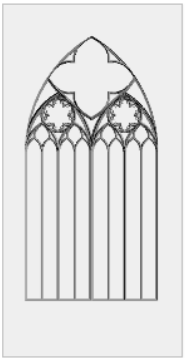
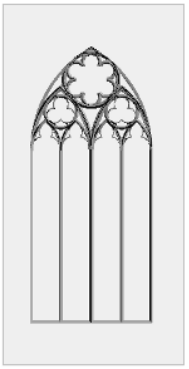
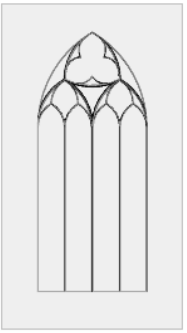
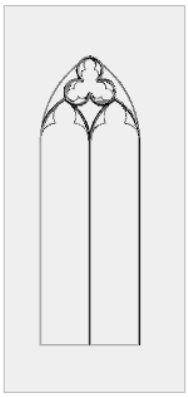
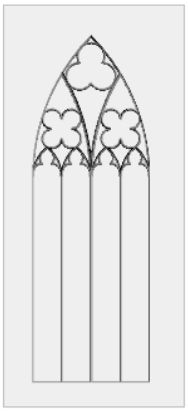
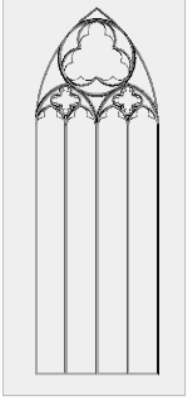
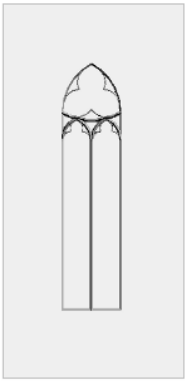
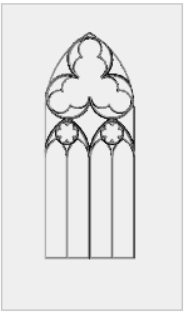
Quality												
											
48		53		56		32		51		9	
15/20	0,75	21/28	0,75	20/27	0,741	14/19	0,737	16/22	0,727	13/18	0,722
											
20		11		63		35		5		6	
7/30	0,233	5/24	0,208	3/15	0,2	4/25	0,16	3/19	0,158	3/20	0,15

TABLE 4: Below each generated model, from top to bottom and from left to right the following values are displayed: picture number, number of times the model was falsely identified as manually created (G_M) divided by the number of times the picture occurred in the quality test (G) and the quality score as a single value (G_M/G). The pictures in the top row are the six pictures with the highest score, the pictures in the bottom row are the six pictures with the lowest score. The pictures are ordered from left to right by the decrease in the score. There are no differences between scores that have been lost due to rounding.

variety are more often very simple. This supports our theory that the unexpectedly low recall for C can be partly explained by our participants possibly believing that they might have missed differences in the details of more visually complex models.

An interesting outlier is picture 13, which is the only picture with a negative score for the objective variety (see table 3). This picture occurred 45 times as a unique picture in the memory test out of which it was identified as unique 21 times ($U_U/U = 0,467$) and it occurred 14 times as a copy out of which it was 10 times identified as unique ($C_U/C = 0,714$), meaning that is was more frequently identified as a new picture when it was a copy than when it was a new picture. As we could not find anything anomalous about the picture itself, we believe that this picture is an outlier by chance.

We found that there was no correlation between the quality of the pictures and the variety of either kind, as can be seen in figures 17 and 18. Some of the windows of the lowest scoring group in variety are among the highest scoring pictures in quality, which is not unexpected as this type of window is very common in real life and thus is relatively likely to be

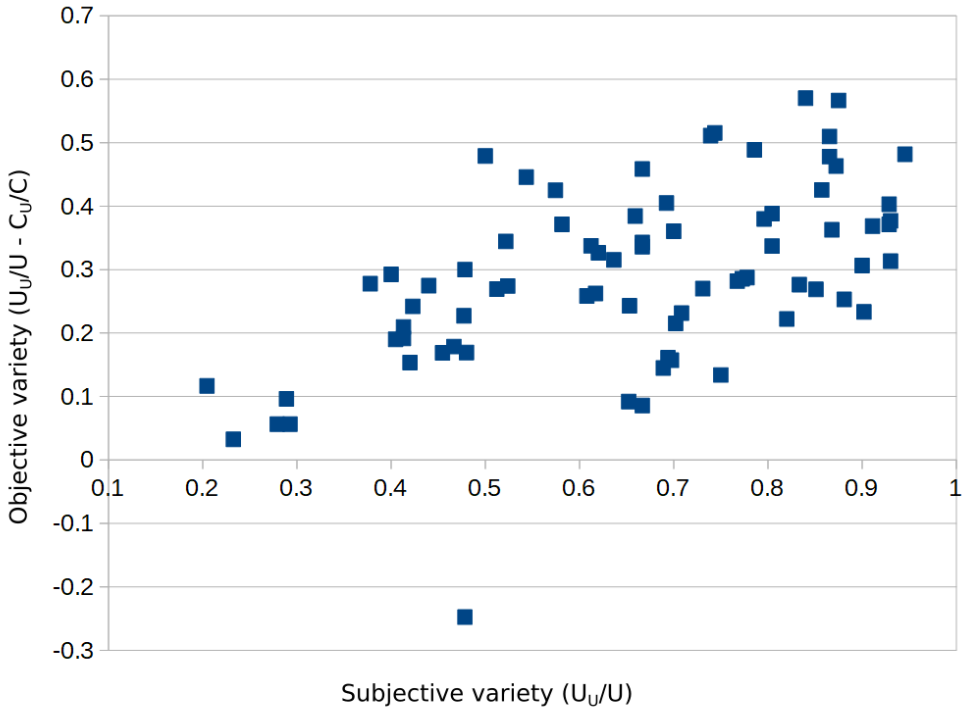


FIGURE 16: The objective variety score against the subjective variety score for each picture. There appears to be little correlation between the two scores.

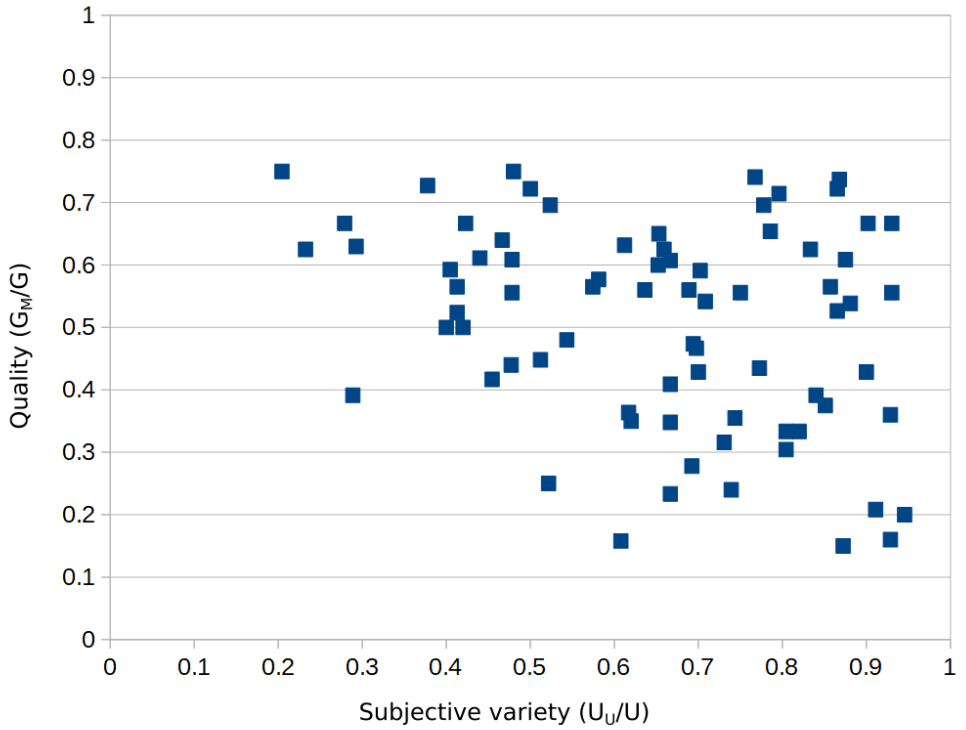


FIGURE 17: The quality score against the subjective variety score for each picture. There appears to be no correlation between the two scores.

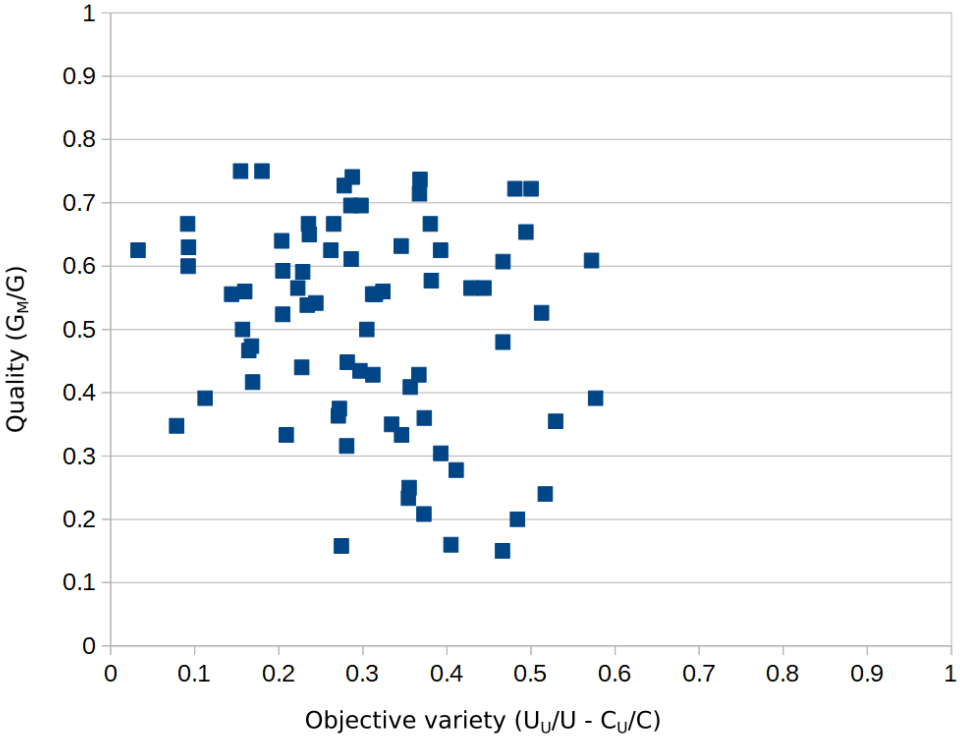


FIGURE 18: The quality score against the objective variety score for each picture. There appears to be no correlation between the two scores. (Picture 13 is omitted from this figure.)

identified as a real-world design. However, as none of our lowest scoring pictures on quality have any features that are unrealistic for real-world Gothic style windows and because our participants were unable to distinguish the real-world designs from our generated pictures, we believe that our presented quality score only reflects what the average person would find typical for real-world Gothic windows and does not necessarily reflect how closely our generated designs resemble Gothic window designs in the real-world.

7 Conclusion

In this paper we presented an approach towards creating a PCG system for human-made objects. While implementing our approach will still require a lot of manual work for the programmer compared to many other PCG systems as each module will need to be manually implemented, we believe this is inevitable and that a modular approach is the only way in which the work can be reduced while the quality of the generated content can still be guaranteed. The better the modules reflect understandable, real-world steps of a design process, the easier it is to extend and maintain the system and the more realistic the combinations of the different modules will be.

Following our approach, we successfully created a PCG system for Gothic window tracery designs that both provide more observed variety than copies of previously seen objects and that are indistinguishable from real world designs. Even though our system is not as expressive as the system created by Takayama [40], it is sufficiently expressive to create a wide variety of designs. An advantage of our approach is that it could also be applicable to automatically generate different objects in different styles.

Our evaluation method for the variety was strict enough so that the limits of our system could be evaluated and seems to reflect the actual variety of our system as the lowest scoring designs were the most common ones. However, our participants misidentified our copies more often than we expected and we could not reliably deduce why that might have been.

8 Future work

One open question that remains is how our proposed approach would work for a broader system that generates many different types of objects in many different styles. It would be interesting to find out how many design concepts and ideas can be generalized over multiple objects and multiple styles, so that more efficient ways of generating man-made objects can be realized.

Furthermore, it would be interesting to see how good such a system can be in creating entirely original styles by combining more elementary style elements in new ways. If there is a large PCG system with many, elementary modules and a parameter systems that can globally describe style rules and style elements, a system may be created that automatically makes a selection out of these modules and parameters to define a new, unique style for a set of virtual objects, and the distinguishability between and the coherency and variation within each style can subsequently be measured through a user survey.

Another interesting venue is combining our approach with machine-learning, in which a machine-learning system learns designs or design elements by the semantic representation of how they can be generated using a particular system that follows our approach and the generated representations are subsequently used to (partly) deterministically generate new content.

Future work may also evaluate our system or other systems following our approach in a different manner that possibly better resembles the ways in which such a system will be used in practice. Particularly, the usefulness of our approach may be tested by implementing it and testing it as a PCG tool in 3D-modelling or other design software or by using such a system to enrich a virtual game world fully automatically and let players explore and evaluate this world and its content directly.

Finally, future work may focus on solving the limitations of our approach, such as the inability to create many figurative ornaments and the inability to create realistic objects that include moving parts, such as doors, cogs and locks. Though it is technically possible to do the latter using modules creating aesthetic content alone, our generator should be able to create open and closed variations of a window if that window looks as if it can be opened without changing any of the design choices that are made. We believe an approach that can be explored is generating the armature of the mesh together with its geometry, as in most cases the way a hinge or slider moves should be as predictable as the way it looks.

References

[1] https://www.glfw.org/.

[2] https://eigen.tuxfamily.org/.

[3] https://www.google.nl/maps.

[4] Timothy F Brady et al. “Visual long-term memory has a massive storage capacity for object details”. In: *Proceedings of the National Academy of Sciences* 105.38 (2008), pp. 14325–14329.

[5] Nathalie Charbonneau, Dominic Boulerice, and David W Booth. "Computer-aided modeling applied to architectural know-how: the gothic rose window". In: *Journal of Information Technology in Construction (ITcon)* 11.26 (2006), pp. 361–372.

[6] Hau Hing Chau et al. "Evaluation of a 3D shape grammar implementation". In: *Design computing and cognition'04*. Springer, 2004, pp. 357–376.

[7] Peter R Cromwell. "The search for quasi-periodicity in Islamic 5-fold ornament". In: *The Mathematical Intelligencer* 31.1 (2009), p. 36.

[8] Arefe Dalvandi, Pooya Amini Behbahani, and Steve DiPaola. "Exploring Persian rug design using a computational evolutionary approach". In: *Electronic Visualisation and the Arts (EVA 2010)* (2010), pp. 121–128.

[9] Peter Ferschin, Monika Di Angelo, and Galina Paskaleva. "Parametric Balinese rumah: Procedural modeling of traditional Balinese architecture". In: *2013 Digital Heritage International Congress (DigitalHeritage)*. Vol. 2. IEEE. 2013, pp. 199–206.

[10] Ji Guohua. "Digital Generation of Chinese Ice-Ray Lattice Designs". In: *New Architecture* (2015), p. 05.

[11] Sven Havemann and Dieter Fellner. "Generative parametric design of gothic window tracery". In: *Proceedings Shape Modeling Applications, 2004.* IEEE. 2004, pp. 350–353.

[12] Sven Havemann and Dieter W Fellner. "Generative mesh modeling." In: (2005).

[13] Mark Hendrikx et al. "Procedural content generation for games: A survey". In: *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 9.1 (2013), pp. 1–22.

[14] Fei Hou, Yue Qi, and Hong Qin. "Drawing-based procedural modeling of Chinese architectures". In: *IEEE Transactions on Visualization and Computer Graphics* 18.1 (2011), pp. 30–42.

[15] Shih-Wen Hsiao and Ching-Hai Chen. "A semantic and shape grammar based approach for product design". In: *Design studies* 18.3 (1997), pp. 275–296.

[16] Diego Jesus, António Coelho, and António Augusto Sousa. "Layered shape grammars for procedural modelling of buildings". In: *The Visual Computer* 32.6-8 (2016), pp. 933–943.

[17] Craig S Kaplan and David H Salesin. "Islamic star patterns in absolute geometry". In: *ACM Transactions on Graphics (TOG)* 23.2 (2004), pp. 97–119.

[18] Craig S Kaplan et al. "Computer generated islamic star patterns". In: *Bridges* (2000), pp. 105–112.

[19] Tom Kelly and Peter Wonka. "Interactive architectural modeling with procedural extrusions". In: *ACM Transactions on Graphics (TOG)* 30.2 (2011), pp. 1–15.

[20] Richard Konecny, Stella Syllaiou, and Fotis Liarokapis. "Procedural modeling in archaeology: approximating ionic style columns for games". In: *2016 8th International Conference on Games and Virtual Worlds for Serious Applications (VS-GAMES)*. IEEE. 2016, pp. 1–8.

[21] Talia Konkle et al. "Conceptual distinctiveness supports detailed visual long-term memory for real-world objects." In: *Journal of Experimental Psychology: General* 139.3 (2010), p. 558.

[22] Ji-Hyun Lee et al. "A formal approach to the study of the evolution and commonality of patterns". In: *Environment and Planning B: Planning and Design* 40.1 (2013), pp. 23–42.

[23] Lin Li et al. “Semantic 3D modeling based on CityGML for ancient Chinese-style architectural roofs of digital heritage”. In: *ISPRS International Journal of Geo-Information* 6.5 (2017), p. 132.

[24] Shang-Lin Li et al. “Rapid modeling of Chinese Huizhou traditional vernacular houses”. In: *IEEE Access* 5 (2017), pp. 20668–20683.

[25] Antonios Liapis. “10 Years of the PCG workshop: Past and Future Trends”. In: *International Conference on the Foundations of Digital Games*. 2020, pp. 1–10.

[26] Peter J Lu and Paul J Steinhardt. “Decagonal and quasi-crystalline tilings in medieval Islamic architecture”. In: *science* 315.5815 (2007), pp. 1106–1110.

[27] Markus Mathias et al. “Procedural 3D building reconstruction using shape grammars and detectors”. In: *2011 International Conference on 3D Imaging, Modeling, Processing, Visualization and Transmission*. IEEE. 2011, pp. 304–311.

[28] Pascal Müller et al. “Automatic reconstruction of Roman housing architecture”. In: *International Workshop on Recording, Modeling and Visualization of Cultural Heritage*. Balkema Publishers (Taylor & Francis group). 2005, pp. 287–297.

[29] Pascal Müller et al. “Procedural 3D Reconstruction of Puuc Buildings in Xkipché.” In: *VAST*. Vol. 2006. Citeseer. 2006, 7th.

[30] Pascal Müller et al. “Procedural modeling of buildings”. In: *ACM SIGGRAPH 2006 Papers*. 2006, 614–623.

[31] Jeremy Noghani, Eike F Anderson, and Fotis Liarokapis. “Towards a Vitruvian shape grammar for procedurally generating classical Roman architecture”. In: (2012).

[32] Seth Orsborn and Jonathan Cagan. “Multiagent shape grammar implementation: automatically generating form concepts according to a preference function”. In: (2009).

[33] Yoav IH Parish and Pascal Müller. “Procedural modeling of cities”. In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. 2001, 301–308.

[34] Heather Richards-Rissetto and Rachel Plessing. “Procedural modeling for ancient Maya cityscapes initial methodological challenges and solutions”. In: *2015 Digital Heritage*. Vol. 2. IEEE. 2015, pp. 85–88.

[35] Marie Saldana. “An integrated approach to the procedural modeling of ancient cities and buildings”. In: *Digital Scholarship in the Humanities* 30.suppl_1 (2015), pp. i148–i163.

[36] Lionel Standing. “Learning 10000 pictures”. In: *The Quarterly journal of experimental psychology* 25.2 (1973), pp. 207–222.

[37] George Stiny. “Ice-ray: a note on the generation of Chinese lattice designs”. In: *Environment and Planning B: Planning and Design* 4.1 (1977), pp. 89–98.

[38] George Stiny. “Introduction to shape and shape grammars”. In: *Environment and planning B: planning and design* 7.3 (1980), pp. 343–351.

[39] George Stiny and James Gips. “Shape grammars and the generative specification of painting and sculpture.” In: *IFIP congress (2)*. Vol. 2. 3. 1971, pp. 125–135.

[40] Joe Takayama. “Computer-generated gothic tracery with a motif-oriented approach”. In: *Proceedings of IASDR 2013* (2013).

[41] Wolfgang Thaller et al. “Shape grammars on convex polyhedra”. In: *Computers & graphics* 37.6 (2013), pp. 707–717.

[42] Julian Togelius et al. “Search-based procedural content generation: A taxonomy and survey”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 3.3 (2011), pp. 172–186.

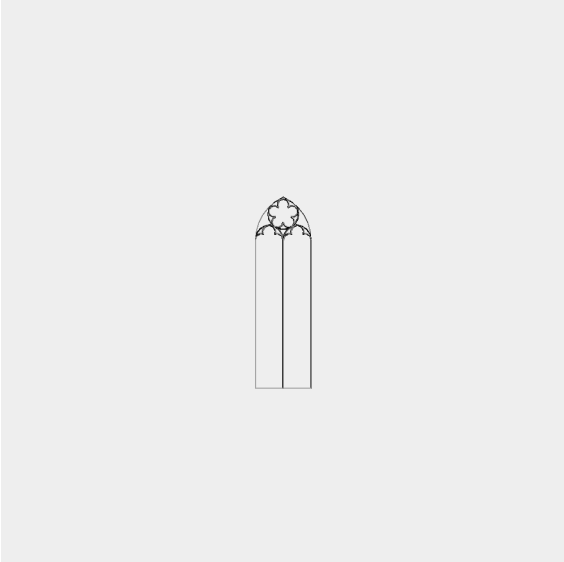
[43] Gert Van Maren, Nathan Shephard, and Simon Schubiger. “Developing with Esri CityEngine”. In: *ESRI Developer Summit. San Diego, CA* (2012).

[44] Jim Whitehead. “Toward proccedural decorative ornamentation in games”. In: *Proceedings of the 2010 workshop on procedural content generation in games*. 2010, pp. 1–4.

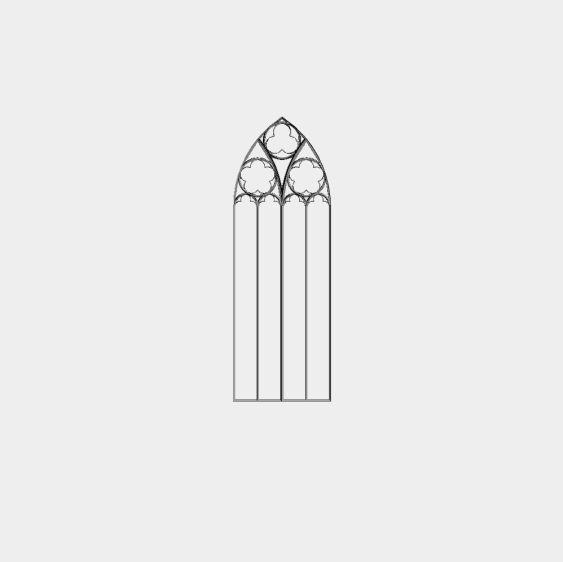
[45] Peter Wonka et al. “Instant architecture”. In: *ACM Transactions on Graphics (TOG)* 22.3 (2003), pp. 669–677.

[46] René Zmugg et al. “Procedural architecture using deformation-aware split grammars”. In: *The visual computer* 30.9 (2014), pp. 1009–1019.

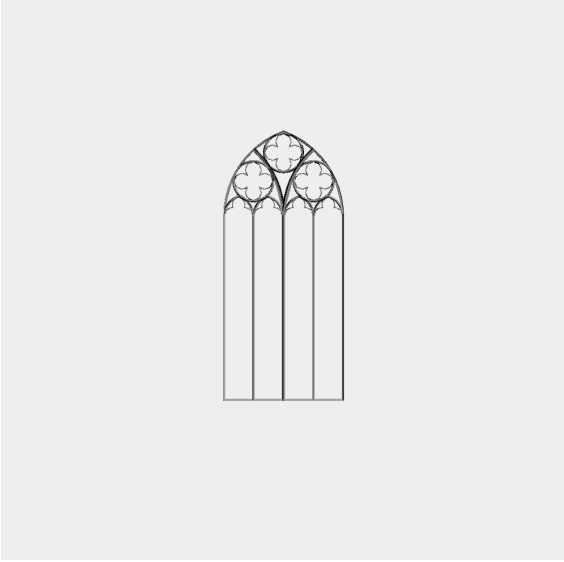
A Generated models



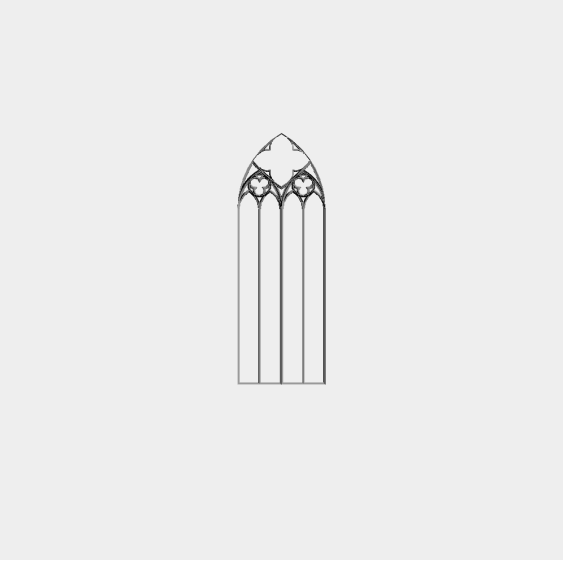
1



2



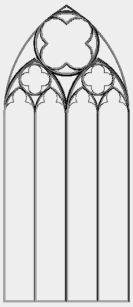
3



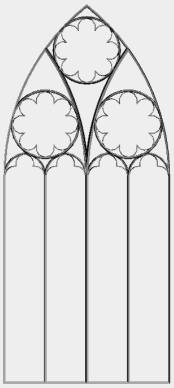
4

5

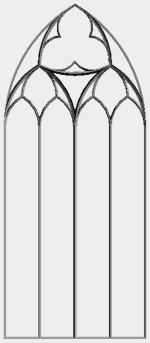
6



7



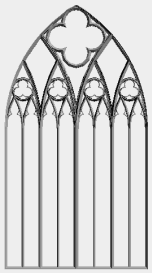
8



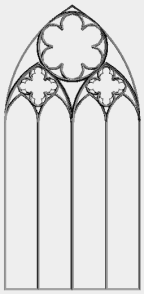
9

10

11

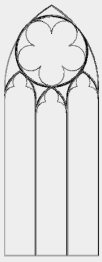


12

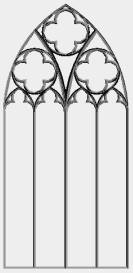


13

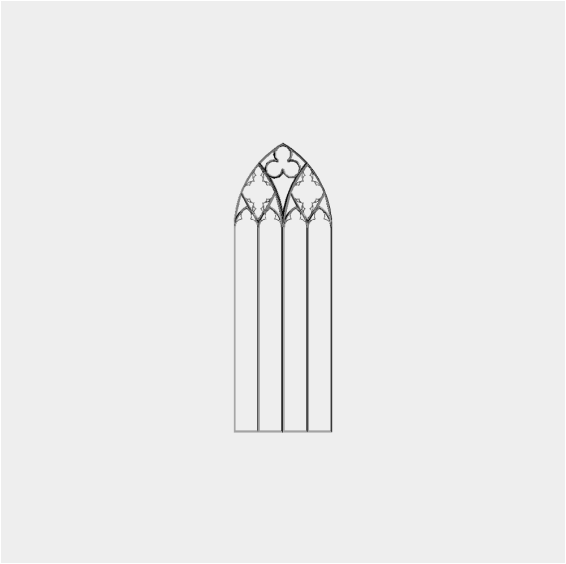
14



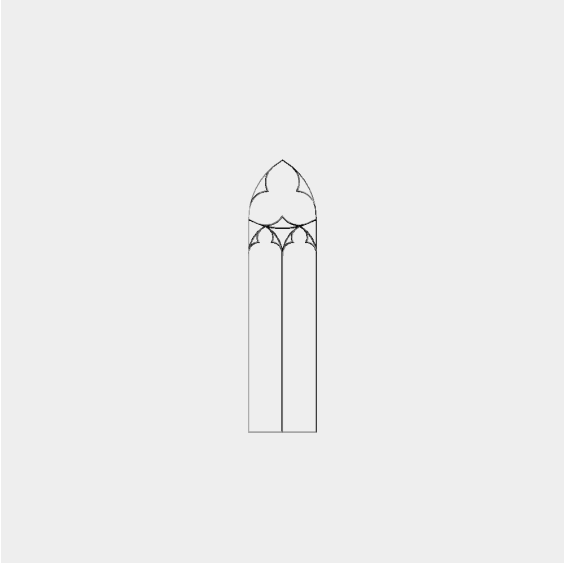
15



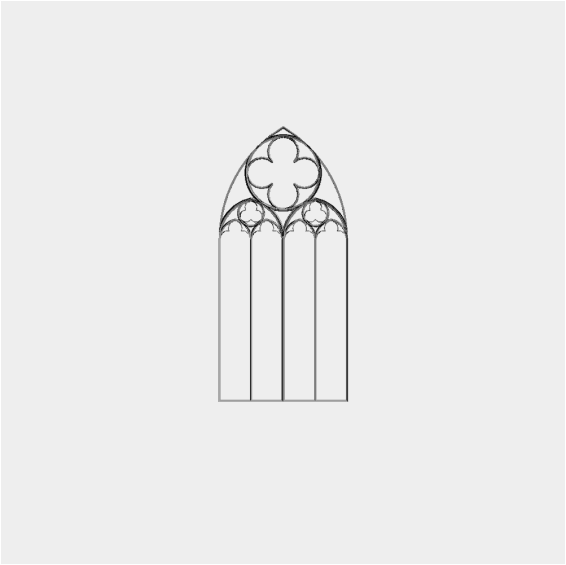
16



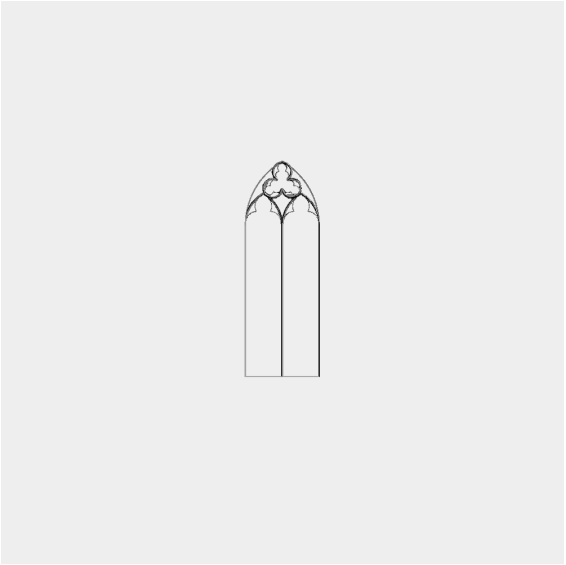
17



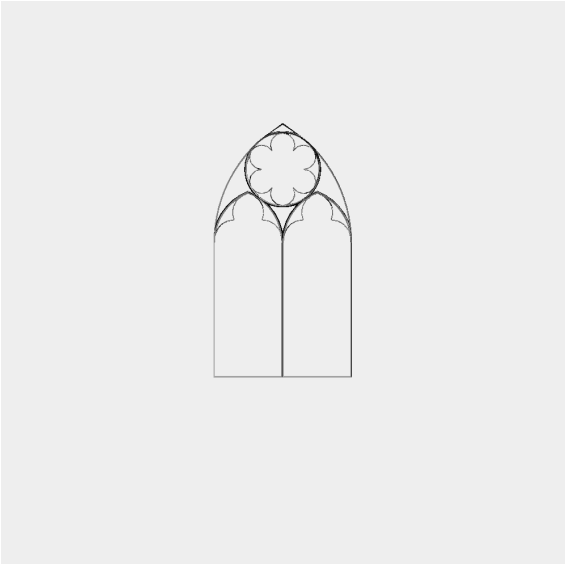
18



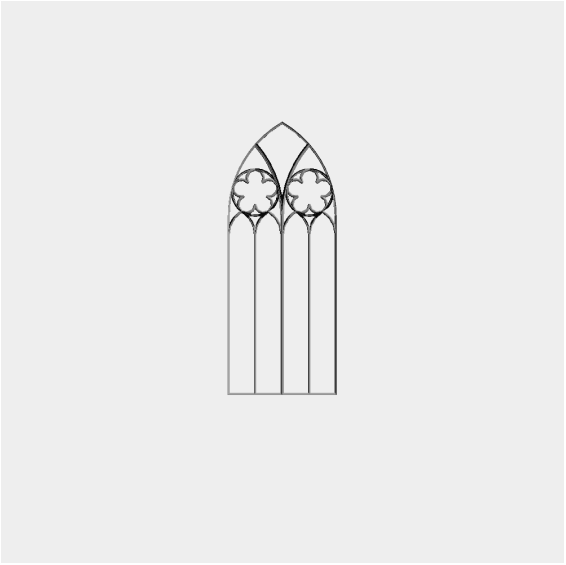
19



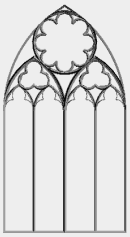
20



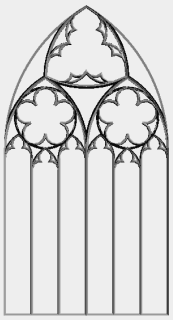
21



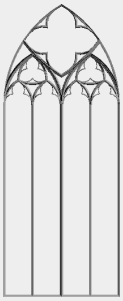
22



23



24

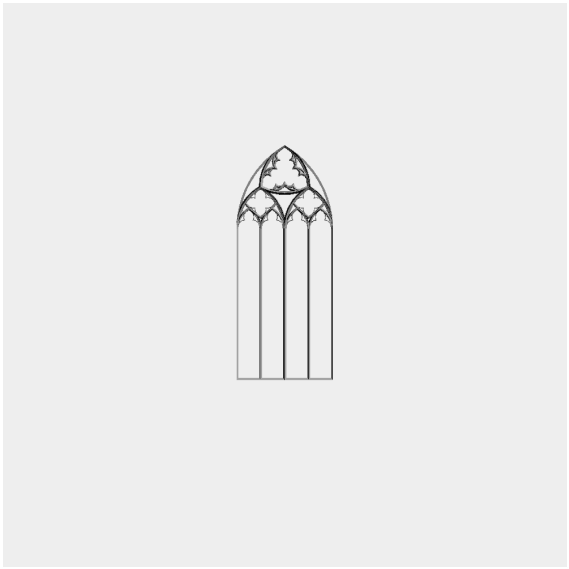


25

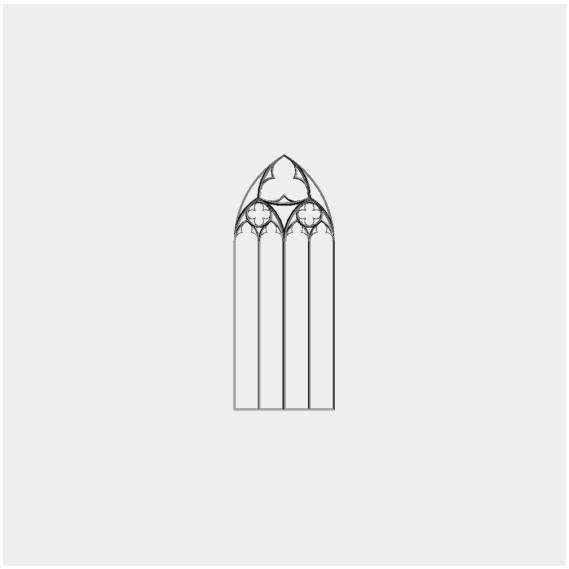
26

27

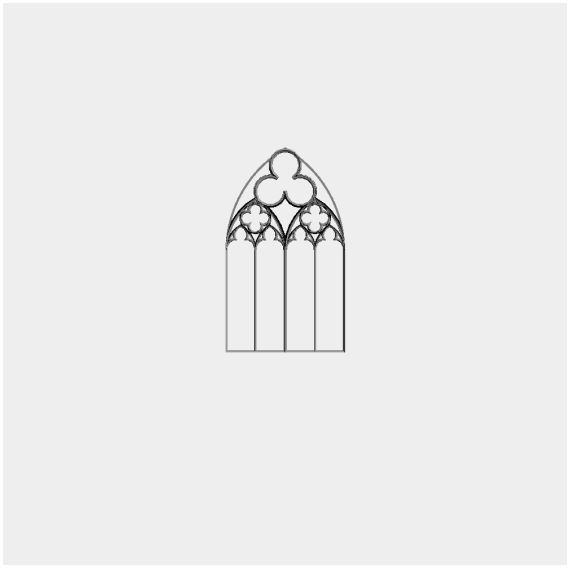
28



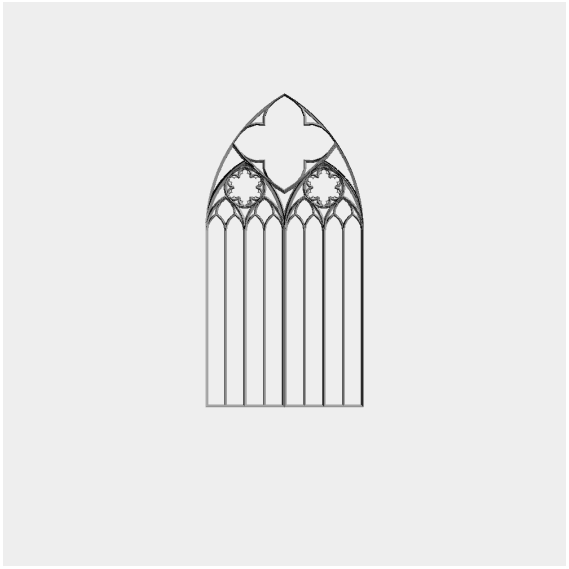
29



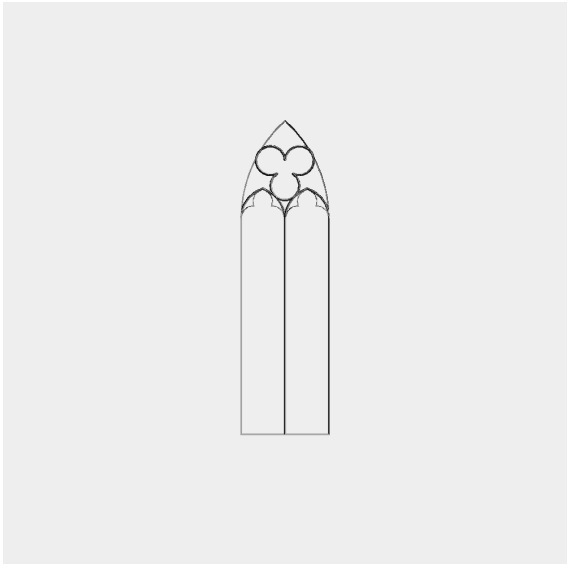
30



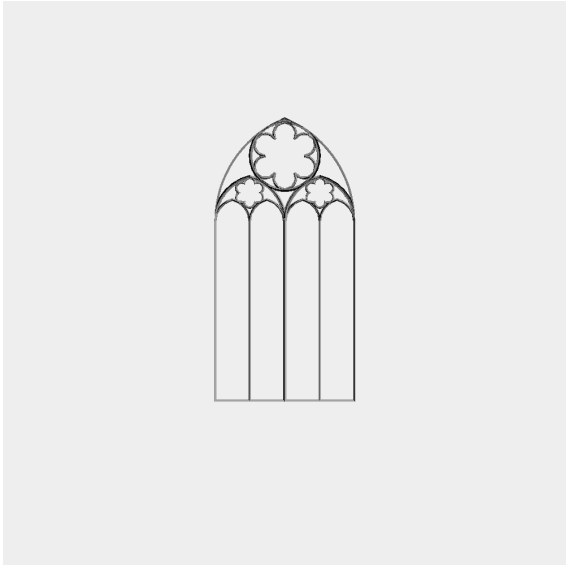
31



32



33

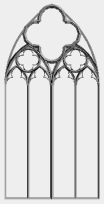


34

35

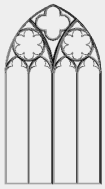
36

37



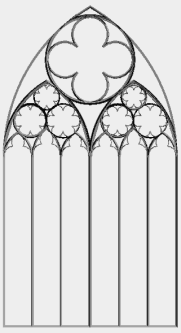
38

39

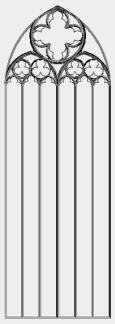


40

41



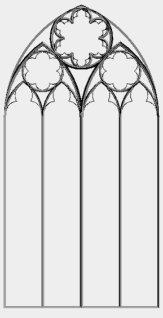
42



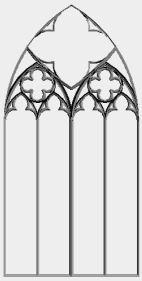
43

44

45



46

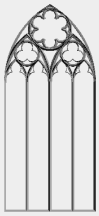


47

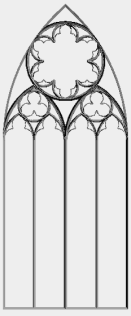
48

49

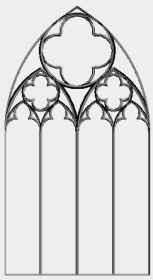
50



51



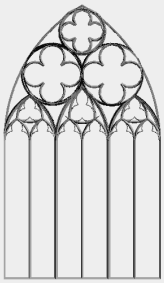
52



53

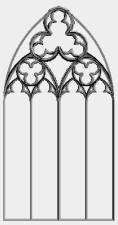
54

55

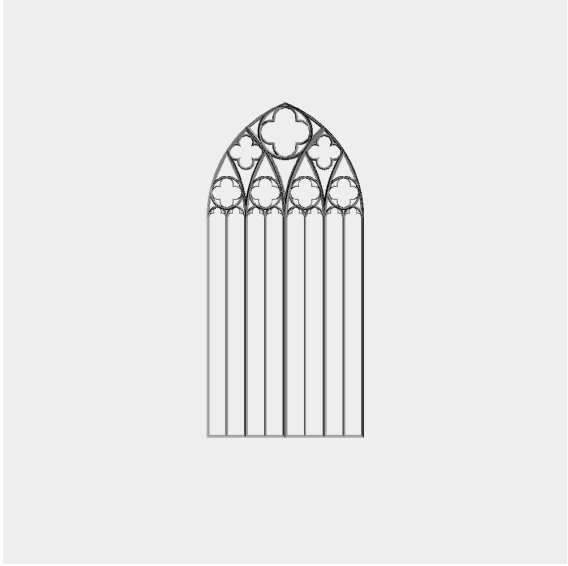


56

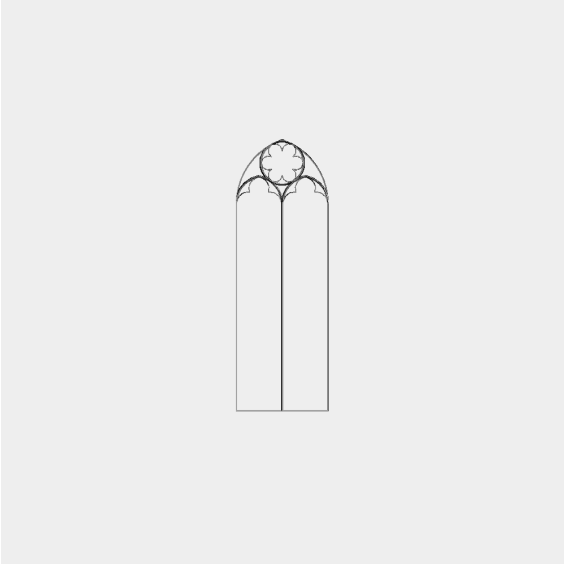
57



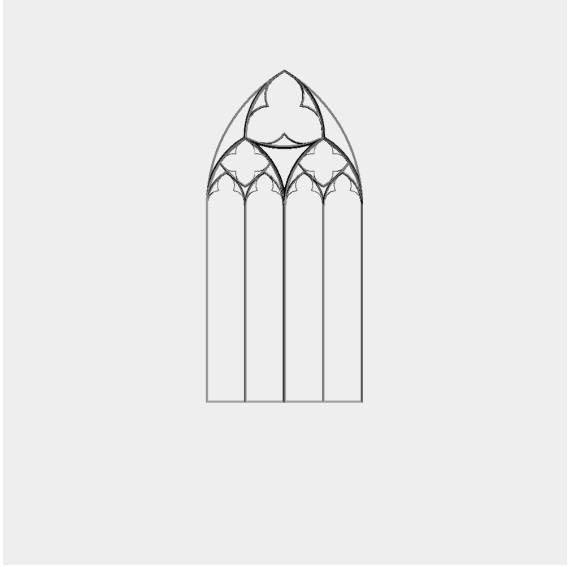
58



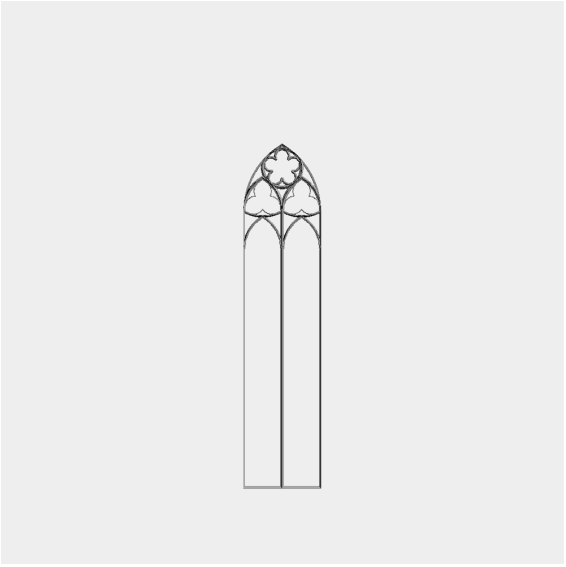
59



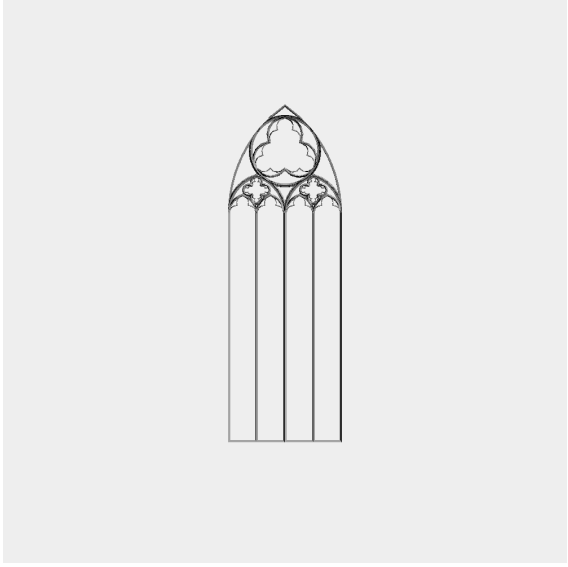
60



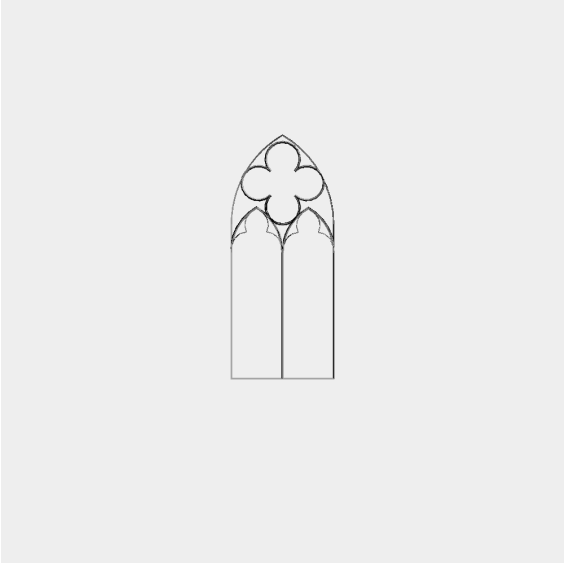
61



62



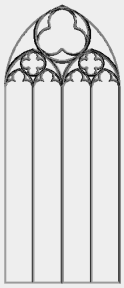
63



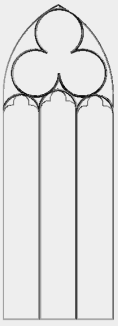
64

65

66

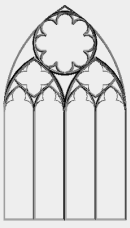


67

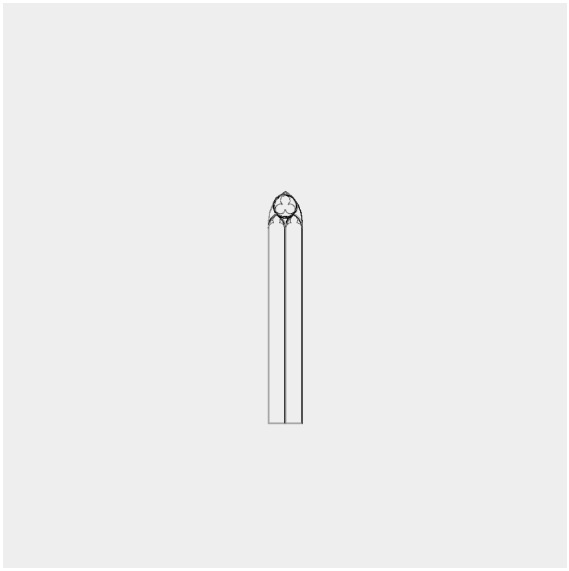


68

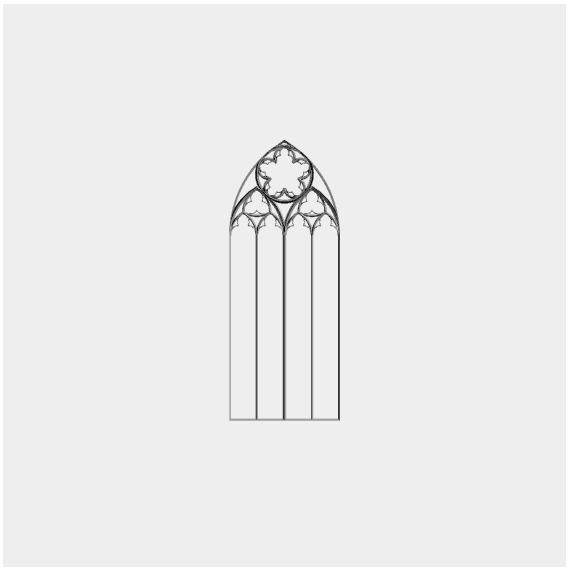
69



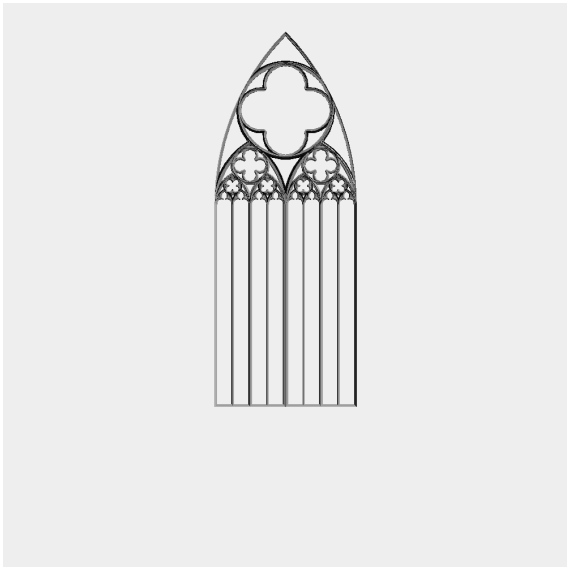
70



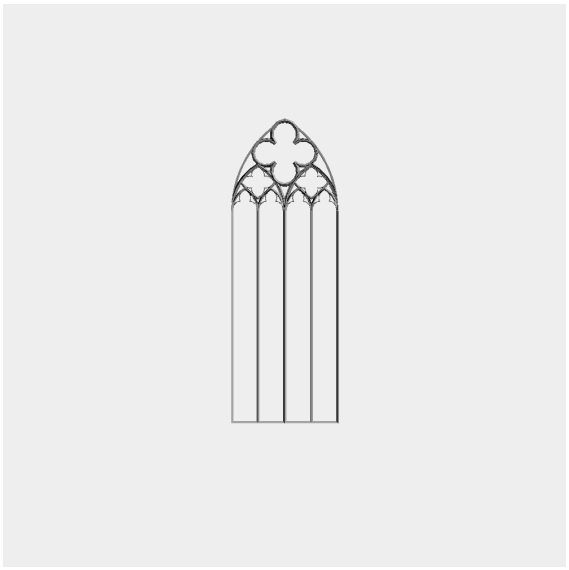
71



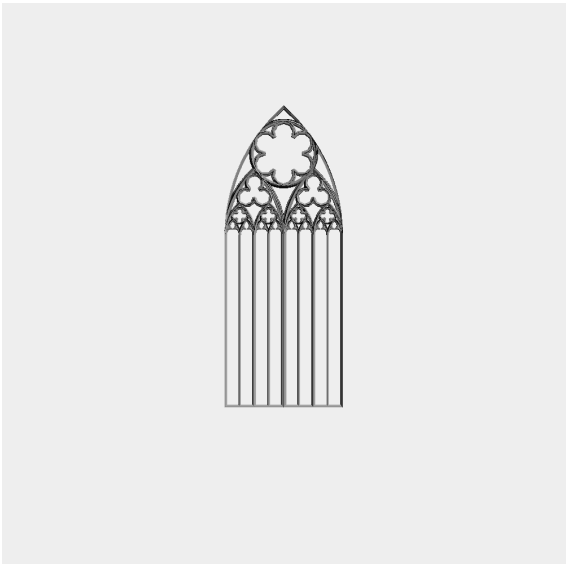
72



73

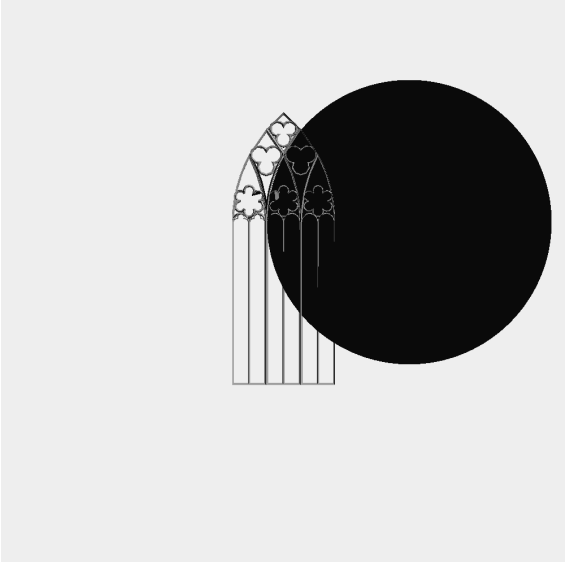


74

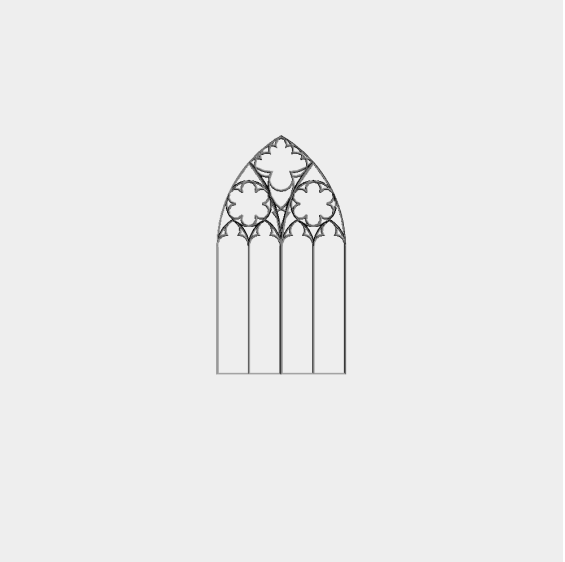


75

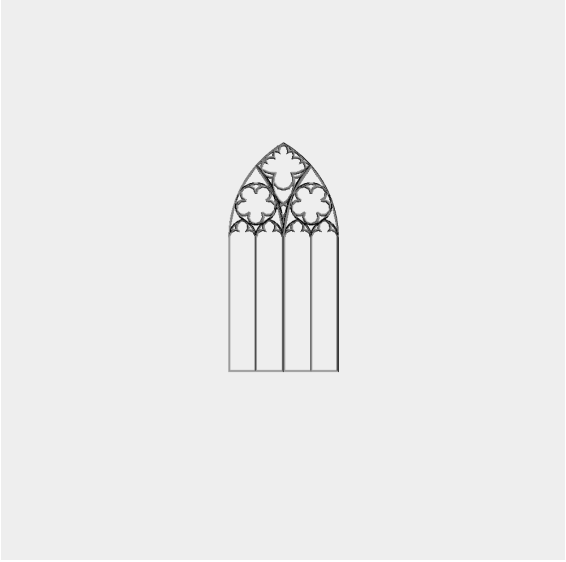
B Erroneous models



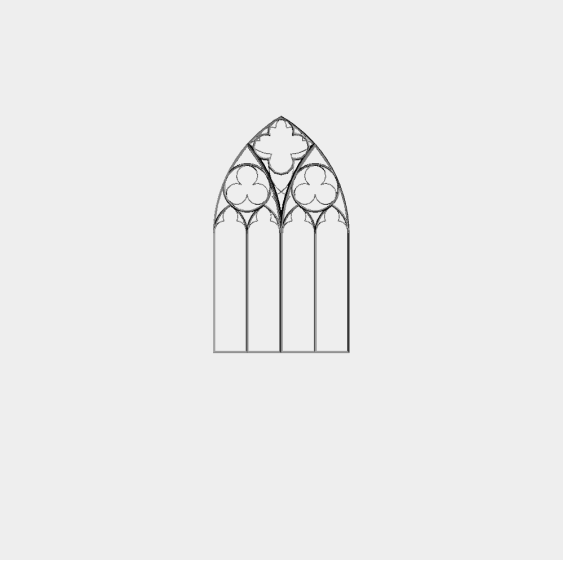
error_1



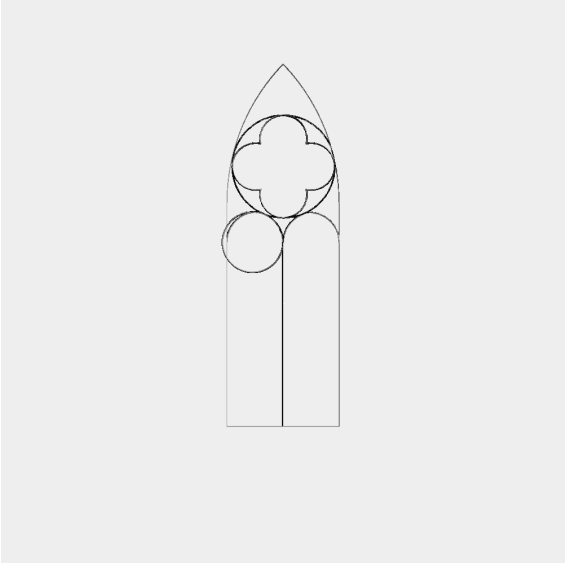
error_2



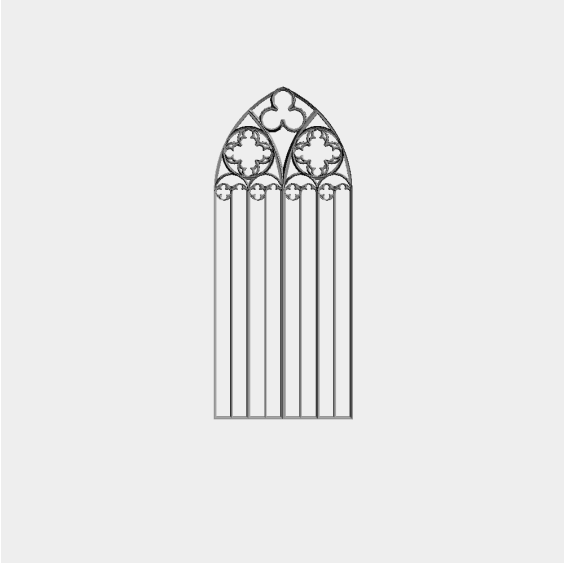
error_3



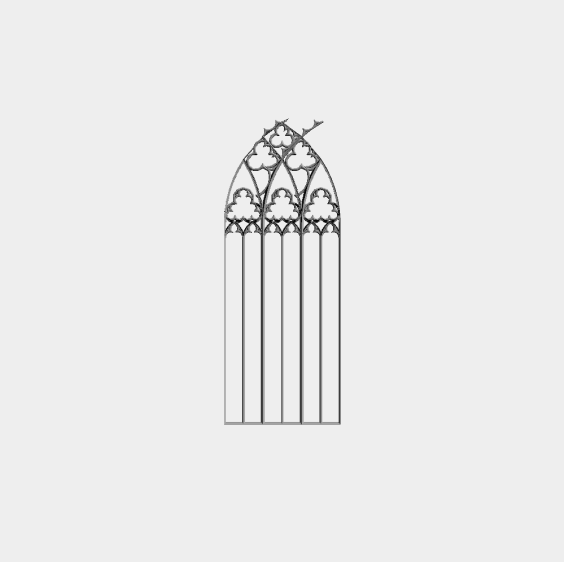
error_4



error_5

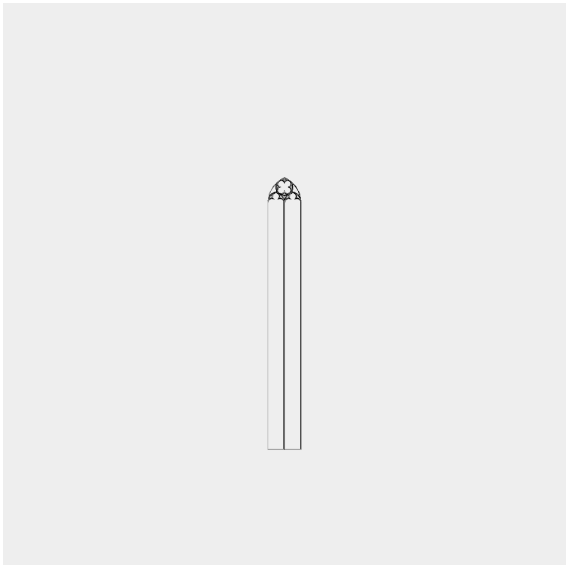


error_6

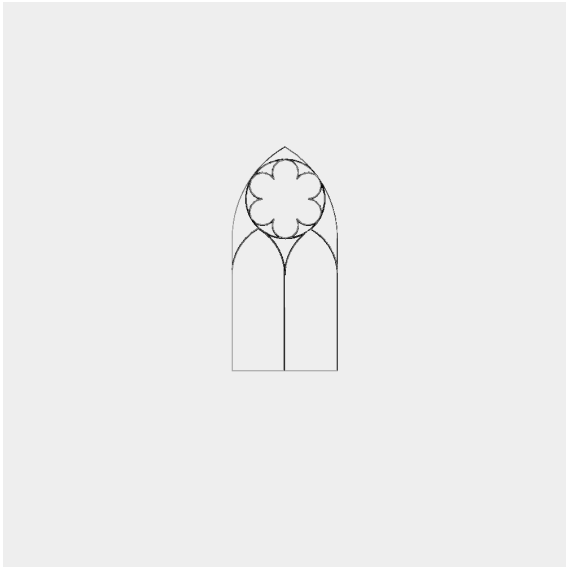


error_7

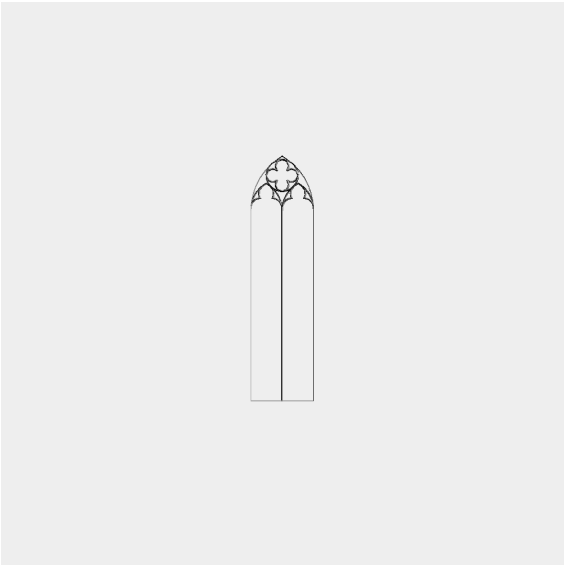
C Golden standard



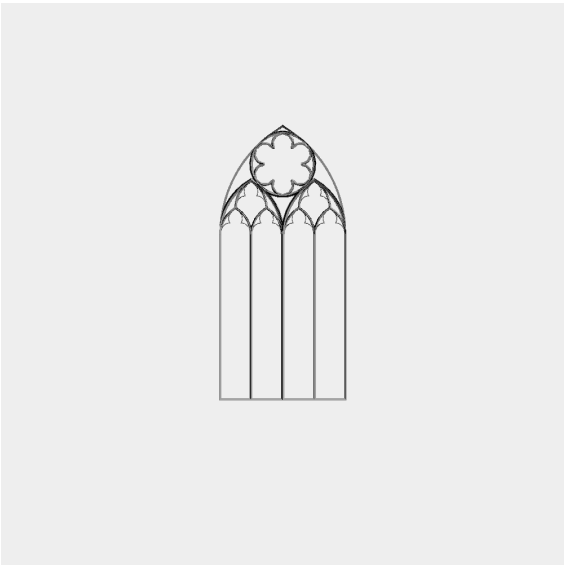
Picture 1, *Oudekerk, Amsterdam*



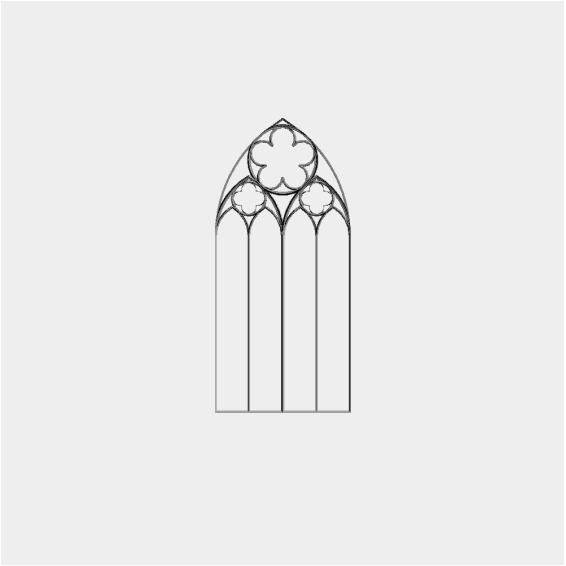
Picture 5, *Notre-Dame, Paris*



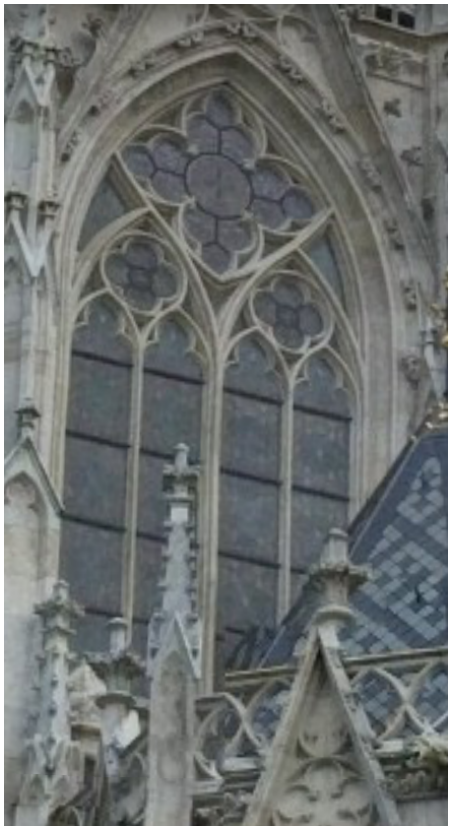
Picture 7, *Votivkirche, Vienna*

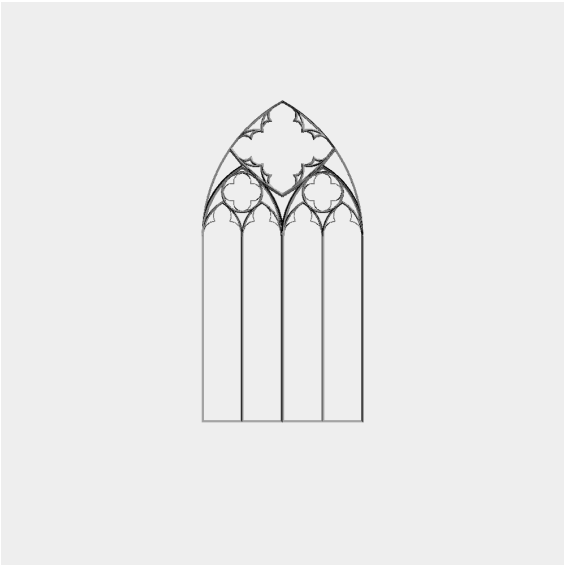


Picture 8, *Sint-Janskathedraal, Den Bosch*

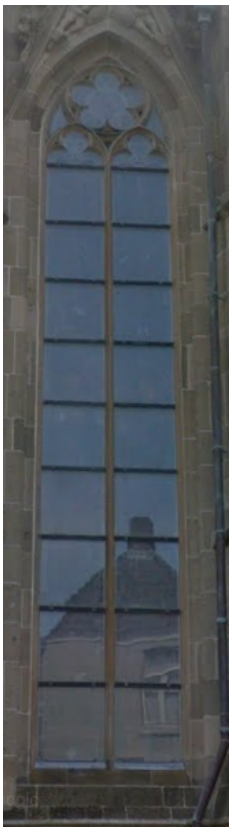


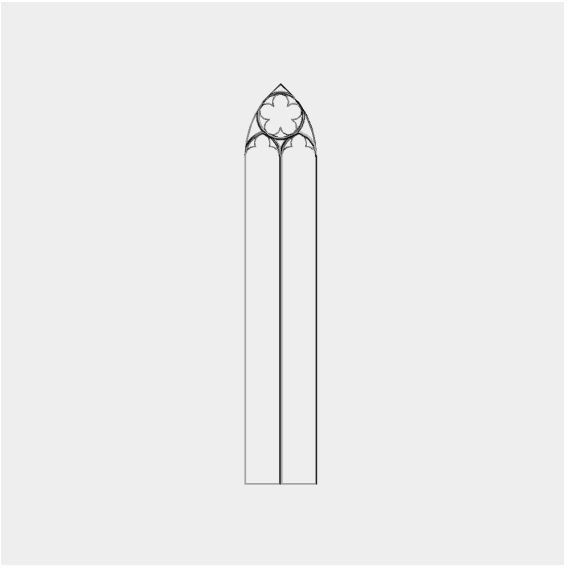
Picture 11, *Notre-Dame, Paris*



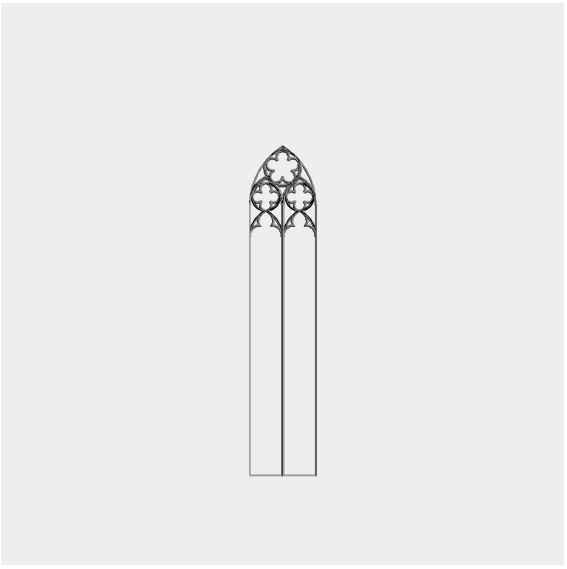


Picture 13, *Votivkirche, Vienna*

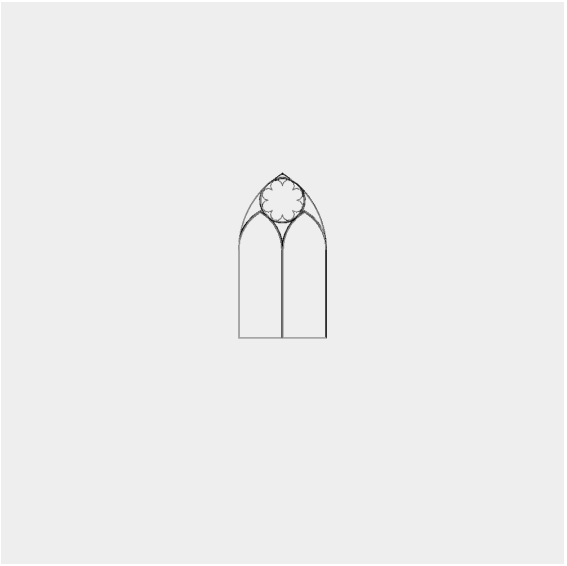




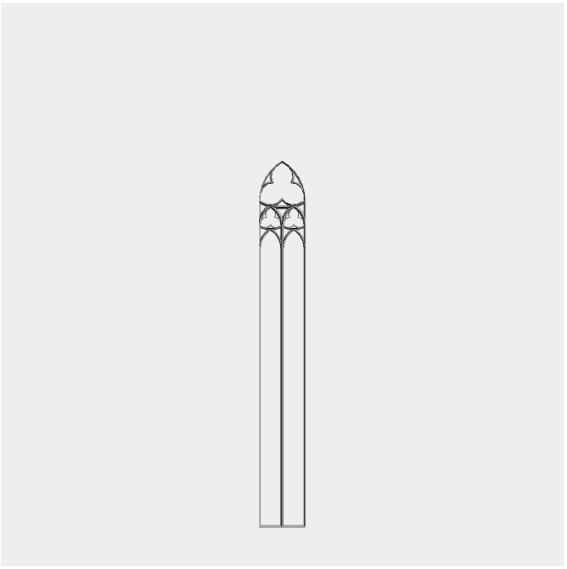
Picture 14, *Sint-Janskathedraal, Den Bosch*



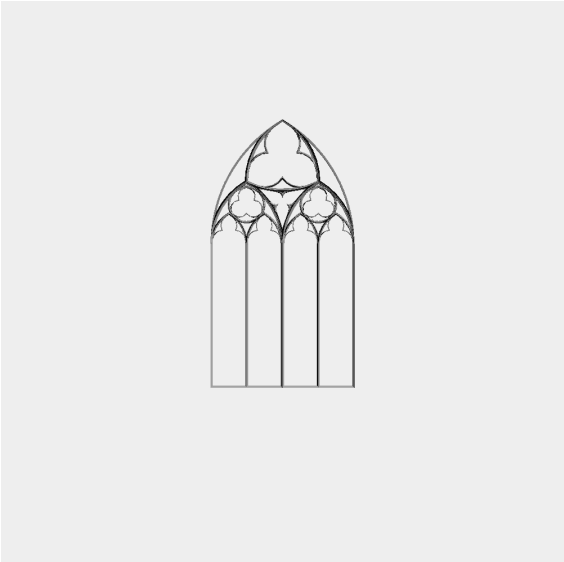
Picture 17, *Notre-Dame, Rouen*



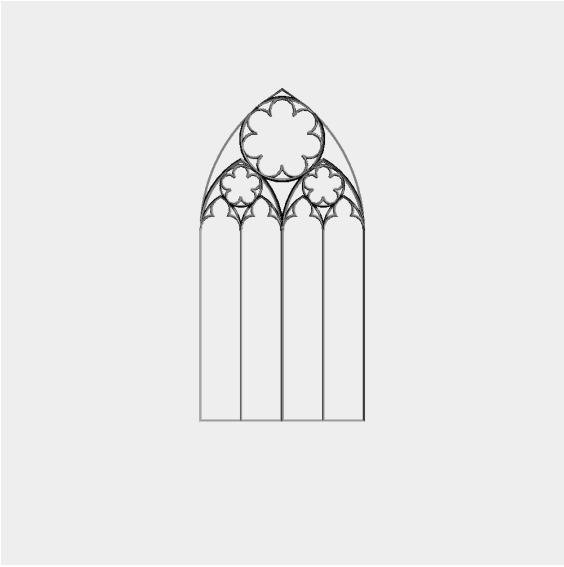
Picture 18, *Saint-Pierre, Beauvais*



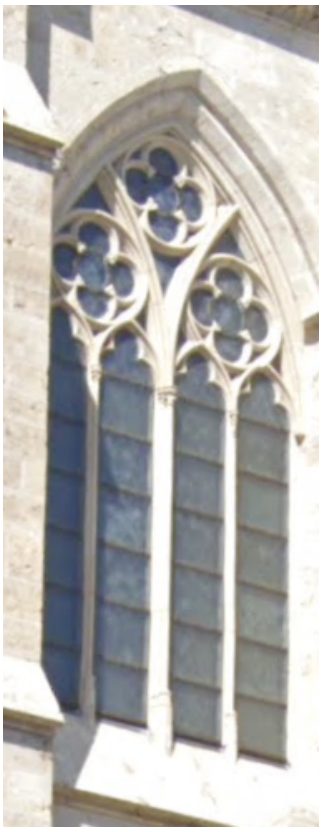
Picture 20, *Dom, Utrecht*

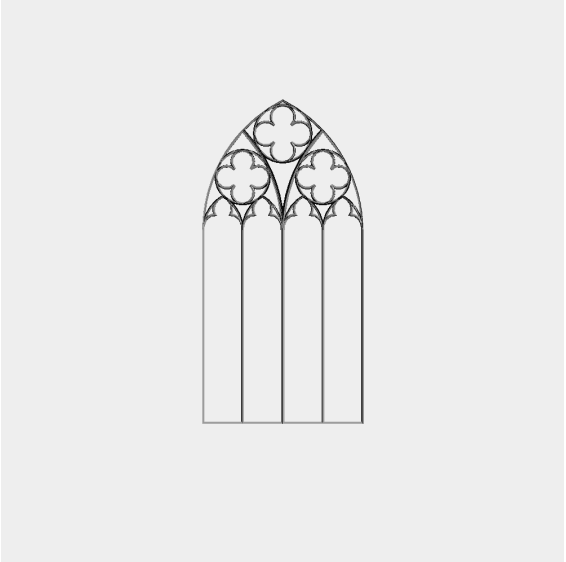


Picture 22, *Sint-Servaasbasiliek, Maastricht*

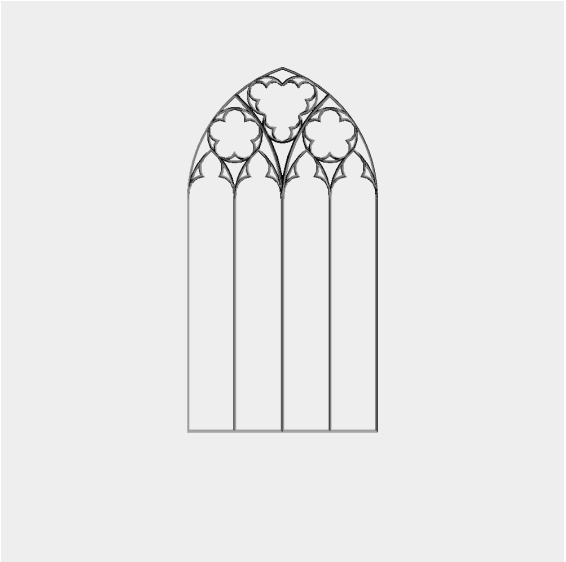


Picture 24, *Notre-Dame, Paris*

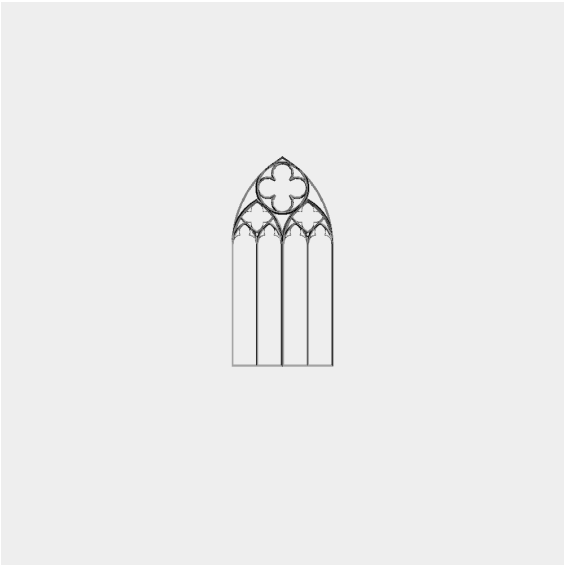




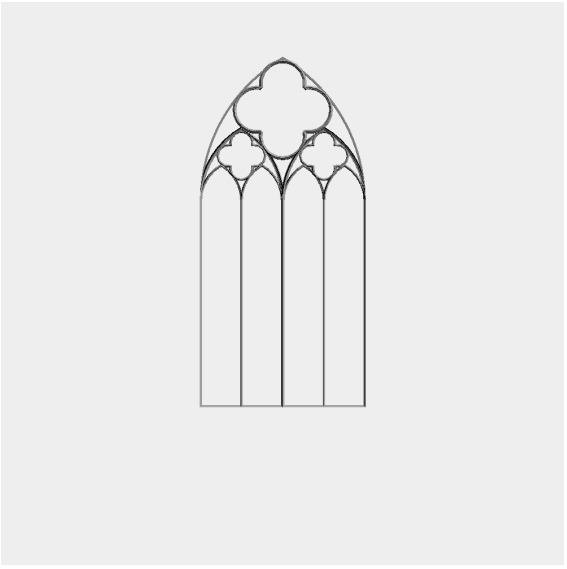
Picture 30, *Notre-Dame, Chartres*



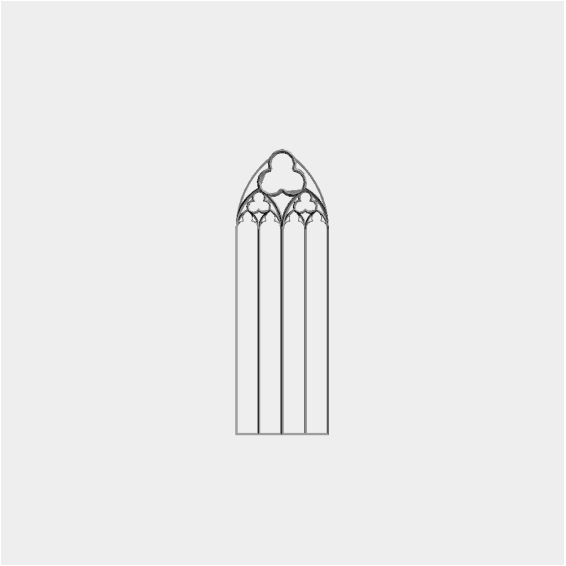
Picture 32, *Dom, Utrecht*



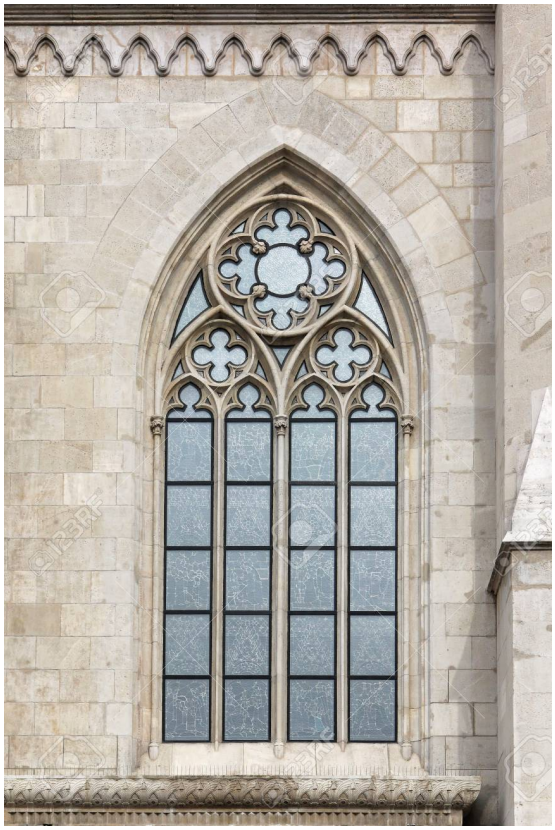
Picture 35, *Sint-Servaasbasiliek, Maastricht*

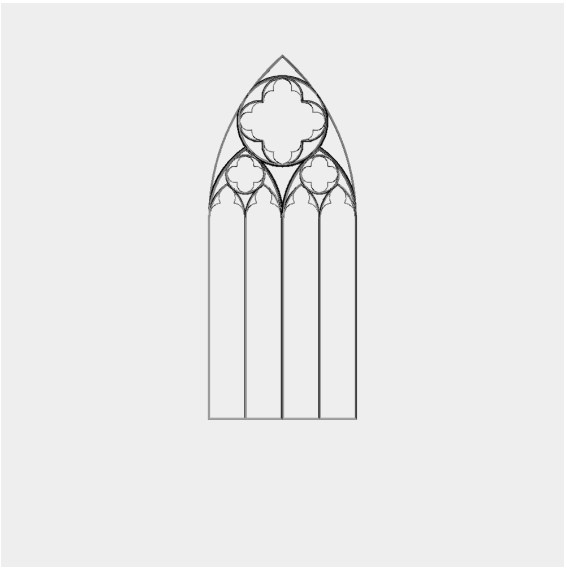


Picture 37, *Notre-Dame, Paris*

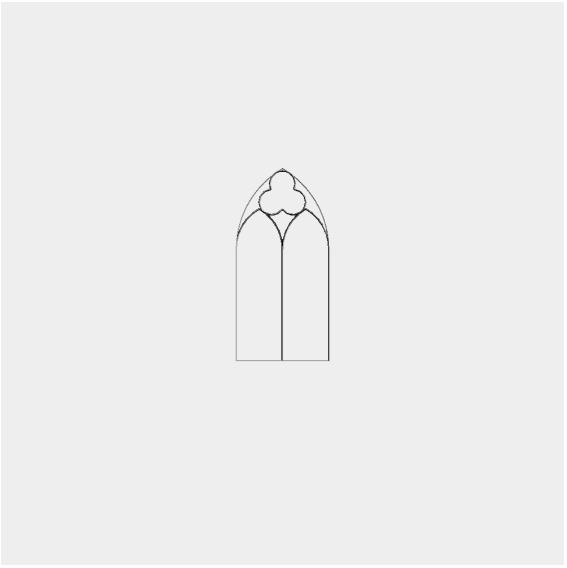


Picture 38, *Notre-Dame, Rouen*

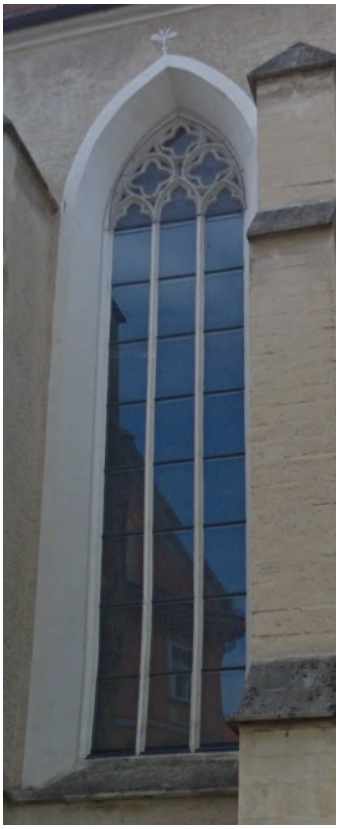


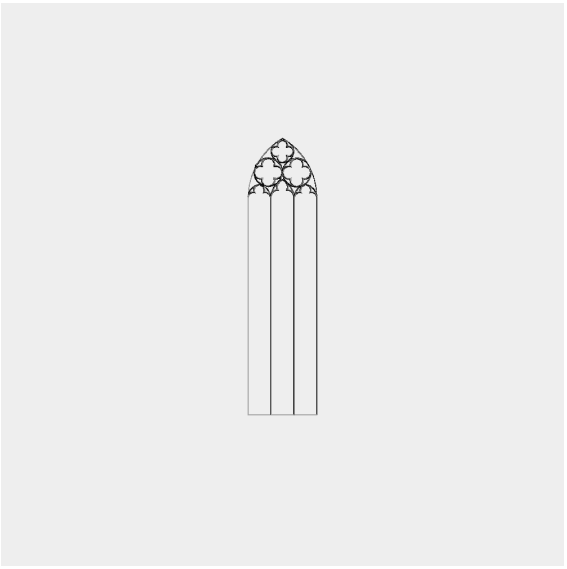


Picture 39, *Mátyás-templom, Budapest*

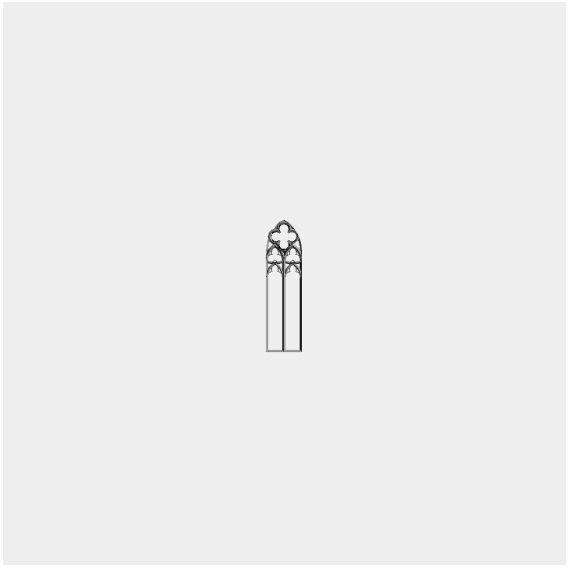


Picture 47, *Saint-Étienne, Sens*

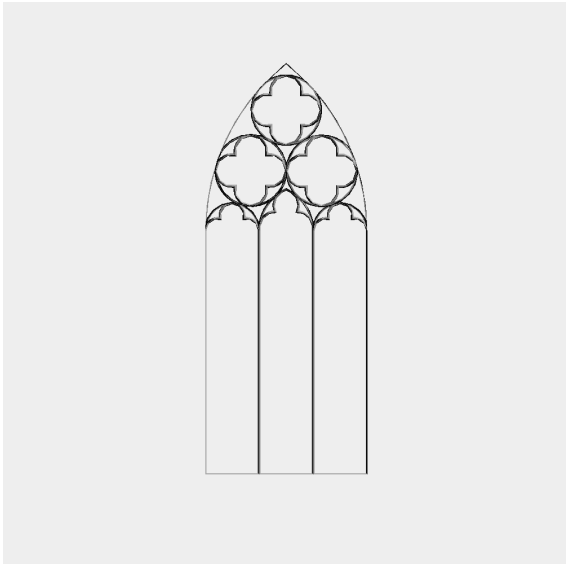




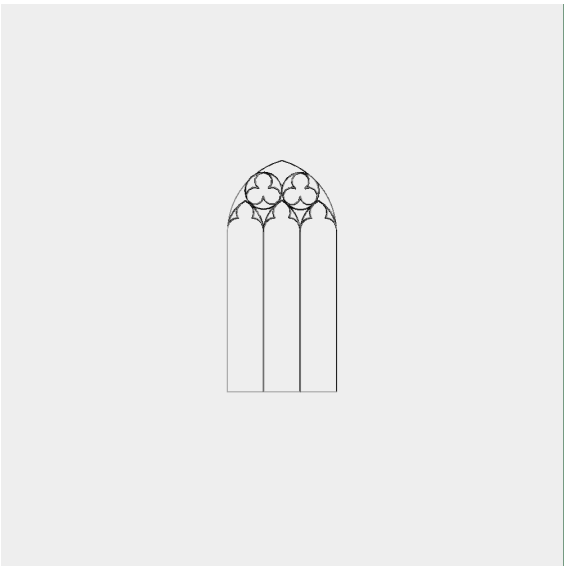
Picture 50, *Fransiskanerkirche, Graz*



Picture 52, *Catedral, Barcelona*



Picture 55, *Saint-Pierre-et-Saint-Paul, Troyes*



Picture 58, *Bavokerk, Haarlem*