Graphics and Visual Computing 4 (2021) 200018

Contents lists available at ScienceDirect

Graphics and Visual Computing

journal homepage: www.elsevier.com/locate/gvc

Technical Section

Volumetric procedural models for shape representation☆

Andrew R. Willis[a], Prashant Ganesh[b,1,*], Kyle Volle[b,1], Jincheng Zhang[a], Kevin Brink[c]

[a] University of North Carolina at Charlotte, Charlotte 28223, USA
[b] University of Florida, 1350 N Poquito Road, Shalimar 32579, USA
[c] Air Force Research Laboratory, Eglin Air Force Base, 32579, USA

ARTICLE INFO

Article history:
Received 6 October 2020
Revised 11 January 2021
Accepted 9 March 2021
Available online 20 April 2021

Keywords:
Modelling language
Procedural models
Shape grammar
Shape representation
Volumetric modelling

ABSTRACT

This article describes a volumetric approach for procedural shape modeling and a new Procedural Shape Modeling Language (PSML) that facilitates the specification of these models. PSML provides programmers the ability to describe shapes in terms of their 3D elements where each element may be a semantic group of 3D objects, e.g., a brick wall, or an indivisible object, e.g., an individual brick. Modeling shapes in this manner facilitates the creation of models that more closely approximate the organization and structure of their real-world counterparts. As such, users may query these models for volumetric information such as the number, position, orientation and volume of 3D elements which cannot be provided using surface based model-building techniques. PSML also provides a number of new language-specific capabilities that allow for a rich variety of context-sensitive behaviors and post-processing functions. These capabilities include an object-oriented approach for model design, methods for querying the model for component-based information and the ability to access model elements and components to perform Boolean operations on the model parts. PSML is open-source and includes freely available tutorial videos, demonstration code and an integrated development environment to support writing PSML programs.

1. Introduction

The generation of realistic shape models for use in computer graphics applications such as virtual reality, computer games and movie production can be a time consuming task. These models often are built in a manner similar to movie sets, i.e., facades are constructed which incorporate the geometric structure and appearance necessary to provide a visually convincing experience when photographed (movies) or experienced from a limited viewpoint (games). However, important contexts exist where the entire structure of the object(s) being modeled are of interest. Examples of these contexts include most real-world simulation applications such as physics modeling, finite element analysis, heat transfer analysis and other contexts such as archaeological research or, more generally, Building Information Modeling (BIM). This article details a new Procedural Shape Modeling Language (PSML) which is a programming language for specifying volumetric procedural models. The article describes PSML and several contributions associated with the existence of a complete language for procedural model specification in contrast to current approaches which extend existing languages. A PSML interpreter which executes PSML programs to generate 3D models is also described.

The initial goal of PSML is to provide programmers the ability to describe objects as a semantic hierarchy of 3D shape elements where each element may be a semantic group of objects, e.g., a floor of a building, or an indivisible object, i.e., a brick within a building. Each indivisible object, e.g., a brick from a building, is modeled in terms of its geometry and appearance. Semantic groupings of objects, e.g., a floor of a building are represented as a geometric 3D primitive which serves to define the approximate scope, i.e., location, pose and size, of elements that are considered to be part of the semantic group. This hierarchical approach allows the programmer to make changes at one level that propagate to other levels. For instance, changing the dimensions of a brick can automatically adjust the element count of a wall that uses that brick definition.

Modeling shapes in this way facilitates the creation of models that more closely approximate the organization and structure of their real-world counterparts. As such, users may query these models for information defined over the semantic labels which is difficult to provide using existing model-building approaches.

☆ This article was recommended for publication by Marco Attene.

* Corresponding author.
E-mail address: prashant.ganesh@ufl.edu (P. Ganesh).

[1] Equal contributor.

https://doi.org/10.1016/j.gvc.2021.200018

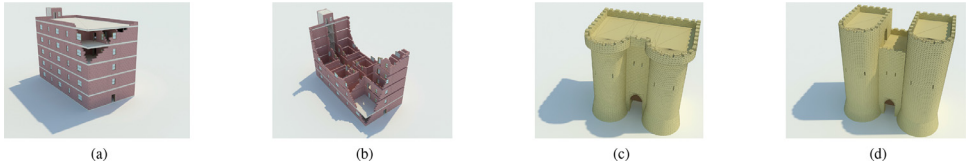


Fig. 1. This article describes a volume-only based Procedural Shape Modeling Language referred to as PSML which enables users to generate volumetric 3D models. An office building (a,b) and a castle gate (c,d) are shown which have been modeled exclusively in terms of their volumetric elements using PSML.

For example, a building model may include a large number of bricks and some volume of mortar needed to bind these bricks together. PSML constructed models may be generated which can readily extract quantities for this information in terms of both the number of bricks and volume of mortar (as well as the location and pose for all the *brick* and *mortar* elements).

Work on this topic was initially inspired from collaborative work with anthropologists and archaeologists who require highly accurate models for historically-important structures (typically architecture). In this context, 3D models have the potential to be extremely useful for addressing tasks which are typically both time-consuming and tedious to perform in their current implementations. The most important of these tasks is documenting and tracking the architectural elements, i.e., the building blocks and structural adornments, of ancient structures. In such contexts, cultural heritage researchers require 3D models similar to those provided for Building Information Modeling (BIM), i.e., the geometry of each indivisible element must be modeled.

This work describes a system that facilitates efficient generation of models which can incorporate this information. As part of our approach, we define a new language, PSML, which is similar in syntax and structure to Java but also incorporates shape grammar programs as elements within the sequential programming code. Considering that the language borrows programming concepts with Java, users familiar with Java or other object orientated programming will be able to easily learn and create new objects. As in other procedural modeling implementations [1,2], these grammars serve to replace shapes with other shapes using *split()* and *repeat()* operations. The resulting models may be applied for BIM systems as contemporary construction typically incorporates structural regularity, i.e., indivisible objects that are similar in geometry and appearance and whose relative positions and orientations are often governed by a small set of simple rules. It is envisioned that PSML could eventually be used for the documentation of historic structures. However, the irregularities in both the indivisible elements (since they are typically hand-made) and irregularities in the construction commonplace to ancient architecture makes modeling such structures difficult using the current implementation of PSML. To summarize, the contributions of this article are as follows:

- We model shape as a hierarchical collection of 3D structures with the intention of modeling objects from their large-scale sub-structures down to their indivisible volumetric elements. As a result, generated models are similar to their real-world counterparts and, as such, may be interrogated to provide estimates of important quantities for real-world objects.
- The volumetric shape grammar associates labels to both object-space, i.e. geometric objects, and void-space, i.e. the empty space that bounds objects. These labels may be used to facilitate analysis that requires knowledge of both types of information, e.g., furniture placement, accessibility, navigation of virtual spaces, path-planning, etc.
- We introduce a language that integrates sequential programming with non-sequential and highly recursive shape grammar programming. Shapes created in shape grammar programs may be referenced from and operated on (using Boolean operations) in sequential code and shape grammars may be initiated using shapes and variables defined from sequential code. This integration enables new context-sensitive programming capabilities which are otherwise difficult to address.
- PSML supports an object-oriented approach for model design. Complex models can be created by assembling simpler parts that are written independently. This feature facilitates collaboration between designers by allowing them to simultaneously work on independent parts. In addition, PSML provides methods to select volumes defined in one grammar and populate them with instances of other grammars, effectively "injecting" new designs into existing models without altering their code or introducing dependencies to them.

Fig. 1 serves to visually convey the central concept of PSML and depicts some of instances of these capabilities that are explained in more detail in the following sections. In (a,b) some volume elements have been deleted to demonstrate that PSML allows users to model structures at the building-block level and also captures the relationships between external and internal structures such as hallways, offices and stairs. (c,d) show two models from a single castle gate shape grammar specified in PSML. These figures demonstrate how high-level semantic relationships between shapes can be detected and used to automatically transform the geometry to suit the circumstance. (a) shows a castle gate with a high connected roof. Note the geometry of the roof and battlements automatically adjust to splice together smoothly. (b) shows a castle gate with separated roofs and a recessed walkway in-between. Here, the geometry has been modified to suit the lower walkway and doorways have been automatically added to the walkway to allow for foot traffic (see Section 3.3.5 for details). Detecting and acting on the volumetric relationships between shapes in this way is not possible using conventional, i.e., surface-based, procedural models.

2. Related work

Procedural modeling is becoming more popular and has been applied in various fields, e.g., Plant modeling using L-Systems [3], terrain modeling using fractals [4,5], model synthesis for generating large variations of given input models [6,7] and procedural creation of Gothic window tracery using Generative Modeling Language [8,9]. Production systems were generally used for generating highly detailed models for large urban scenes which include cities [10–12], road networks within cities [13,14], the buildings within these cities [2,15] and to generate graphics for games [16]. Smelik in [17] does a survey on the use of procedural modeling for virtual worlds. The net effect of these tools and extensions thereof, e.g., extruded surfaces as implemented in [18], coherence-based facade editing in [19], automatic computation of the layout and organization of residential buildings [20] and automatic placement of furniture within the model as implemented in [21–23], have demonstrated impressive capabilities for efficient generation of geometric shapes.

Current implementations for grammar-based procedural models start with geometric *primitives* which are solid geometries that typically come from a predefined set. Initial rules of the procedure

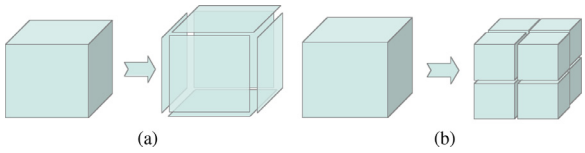


Fig. 2. (a) Depicts the component split operation on a cube which operates on the cube shape as input and generates 6 faces of the cube as output. (b) Depicts a split of a cube in PSML. PSML restricts shape grammar operations such that all elements must be volumetric.

create instances of these primitives and place them into a relative configuration to generate a *mass model* (other approaches generate the mass model by extruding a closed 2D polygon). The surfaces of the mass model are then extracted from the mass model by decomposing each primitive into its constituent planar faces using a *component split* operation (see Fig. 2). Subsequent rules then operate on the faces of the mass model to generate a 3D "shell" model. Such methods have been shown to produce highly detailed models for building exteriors and, for appropriate definitions of the mass model, can be used to model shells of both the interior and exterior of a building, e.g., some of the Roman buildings of the virtual Pompeii in [2] have courtyards. As such, this work solves problems of great importance for generating massive models of cities with highly detailed buildings. Other applications of procedural modeling used in archaeology include [24–26].

However, "shell" models are not appropriate for more advanced applications such as physical simulations and spatial analysis. Several approaches concentrated on using volumetric grammars for such areas. Whiting et al. created a volumetric grammar derived from CityEngine's Computer-Generated Architecture (CGA) language [27]. This volumetric grammar is optimized to produce structurally sound buildings. Terminals generated by this grammar are restricted to visible volumetric shapes and no labeling of empty space is provided as it is not required for structural optimization. Another work by Cutler et al. addressed the creation of solid models [28]. Cutler et al. introduced a procedural approach for converting closed surfaces into solid models. The goal of their work, however, is restricted to converting surfaces into volumes and does not aim to generate complicated structured shapes and hierarchies. Similarly, Jesus et al. in [29] uses a layer based modeling technique which gives a structured approach for an algorithm to merge two large grammars that are developed independently. This allows for simpler syntax for larger models and writing grammar with less lines of code. More recent work [30] has provided the ability to model solid and empty spaces in a hierarchical framework but without some of the flexibility afforded by PSML.

The key difference between PSML and previous implementations is that PSML is constrained to work *entirely* from closed geometries *and* associates semantic labels to non-terminals, visible terminals and empty space. In contrast to [27], PSML's ability to label and represent empty space volumes facilitates the use of space information in both the creation process and as data readily available for a post-processing step. The closed geometry of terminal symbols generated by PSML can be converted into solid geometry for further processing if needed. Thus, we view work of [28] as a possible post processing step that may be applied to terminal symbols.

PSML's ability to create volumetric hierarchies with associated semantic labels allows one to interrogate the model to ask important questions such as the number and volume of some semantic element such as a brick. This information allows the impressive models of ancient cities such as ancient Pompeii [2] and other ancient structures such as the Puuc building shown in [31] to be interrogated for information that anthropologists and archaeologists find useful. It is clear that these numbers will be approximate, yet, development of high-fidelity models validated by cultural heritage researchers shows promise for providing new insights on research on these structures such as building methods, units of measure and construction techniques employed by ancient civilizations.

Another important application of procedural modeling is computer vision. There has been a considerable amount of recent work that investigates the use of shape grammars for vision tasks with a large number of articles being produced that focus on segmentation of architecture [32–34] within images or segmentation of building facade images [35–39] with impressive results. However, these techniques were limited to 2D grammars since the labeled primitives produced by the used grammars were limited to be 2D faces. It is envisioned that PSML may be an effective context within which these vision algorithms may be applied. The problem statement here is typically one where the user seeks to infer the content of an image using a shape grammar program as a model for the image content. The grammar constrains the solution space to plausible organizations of semantic elements by requiring solutions to come from the language of the grammar. In practice, this is accomplished via an optimization algorithm that searches for the shape grammar variable values the best "fit" the grammar shapes to the image data. Not all grammars are equally amenable to this type of optimization and there is interesting research into how to make grammars as functional as possible [40]. Work in [41] represents a recent example of using machine learning to generate structural shape programs that seek to be geometrically consistent with their real-world counterparts. Other approaches use Monte Carlo methods [42,43] to achieve similar procedural modeling goals. These works also investigate the opportunity for interpolating between generated models [44]. All of these applications may benefit from PSML as a new representation for their shapes.

2.1. Why not use an existing language directly?

There exist languages for specifying sequential programs, e.g. C, C++, Java, etc., and languages for specifying grammars, e.g. ANTLR, YACC/Lex, Bison, etc. However, to our knowledge, there is no language that incorporates both within a single source file. One benefit of procedural models is their context-sensitive behavior. This refers to instances when the generated geometry changes due to a specific contextual condition, e.g., a visibility or occlusion constraint in [2]. Detecting and acting on these contexts requires sequential logic (scripting or special functions) which serve to detect the context and change the geometry to suit the specific situation. Hence, sequential logic is intrinsic to specifying procedural models. On the other hand, procedural models are typically specified as formal grammars and, as such, they require a syntax appropriate for specifying formal grammars. Since the syntax for sequential code and formal grammars are quite different a new language is proposed which explores a specific combination of these two languages.

2.2. Why not extend an existing language?

Published implementations for procedural modeling syntax have been implemented as extensions to existing languages. For example, CityEngine [10,11] extends the Python scripting language. In this context, users write python code and call shape grammar (.cga) files which are associated with a separate interpreter that executes the shape grammar and produces geometry. Such methods have proven to be effective for generating shape as evidenced by the popularity of this approach and the impressive results it has produced [2]. Yet, from a programming perspective, it can be difficult to develop within this context as the logic within the Python code is separate from the logic applied for shape generation in the CGA shape grammar. This separation between the Python

script and the shape grammar complicates the development of logic that bridges this gap and requires the Python interpreter to communicate with the shape grammar interpreter. The extent of compatibility between these two interpreters can limit the possible operations between the sequential logic and the shape grammar.

The fundamental reason that we do not extend an existing language is based on the concept that PSML programs *are* shapes, i.e., the fundamental object within the language is a shape. All PSML programs are derived instances of this "base class." In this regard, PSML programs can be seen as a semantic collection of shape specifications. The base class of an existing language, e.g., Java's *Object* class, is a core element upon which all abstractions are based. Redefinition of such classes requires significant restructuring of the language as a whole which motivated development of a separate complete language. We are still investigating methods to extend existing sequential languages, particularly Java, in ways that are tractable and still allow us to maintain the desired abstractions and syntax presented as PSML in this article.

3. PSML shape grammars

The approach PSML takes is to separate sequential statements from the grammatical statements (rules) using simple syntax. PSML is a combination of sequential code which has a structure and syntax inspired by Java and shape grammar code which has a structure and syntax inspired by L-systems [3]. The overall structure of a PSML program includes one *grammar* declaration that contains one or more *method* declarations. Each method declaration must include at least one *rules* declaration. A very basic example of this structure is shown in Algorithm 1.

As mentioned in Section 2.2, each grammar is intended to represent some shape, i.e., the CoffeeMug.psm grammar in Algorithm 1 is intended to make geometric models of coffee mugs as shown in Fig. 3. Typically, a collection of methods is defined within each grammar and each of these methods takes as input a reference shape and an optional list of arguments and operates on the reference shape, replacing it with a sub-tree of other shapes generated from grammars defined within that method. Arguments passed to methods allow each grammar to use context-sensitive information to influence the shape generation process which, among other things, is useful for controlling level-of-detail. Rule blocks must be defined within each method

Algorithm 1. The contents of PSML file: CoffeeMug.psm

```
1  public class CoffeeMug extends ShapeGrammar {
2    float t = 0.4, w_top = 4.5, w_bottom = 3, h = 8.5;
3
4    public CoffeeMug() {
5      rules {
6        axiom::I(conicfrustum, {w_top, w_bottom, h}) R(
             Math.PI/2,0,Math.atan(h/(w_top-w_bottom)))T((
             w_top+w_bottom)/2,0,0)I(cylinder,{w_bottom, t
             }){vessel, handle_c};
7        vessel::split(y, {t, scope.h - t}){vesselBody,
             vesselTop};
8        vesselTop::split(r, {scope.r - t, t}){space,
             vesselBody};
9        vesselBody::appearance(diffuse,{1,0,0}){terminal};
10       handle_c::split(theta,{Math.PI,Math.PI}){space,
             handle_v};
11       handle_v::split(r, {scope.r - t, t}){space, handle
             };
12       handle::appearance(diffuse,{0,1,0}){terminal};
13       space::void(){terminal};
14     }
15   }
16   public static void main(String[] args) {
17     rules {
18       Axiom::{CoffeeMug()};
19     }
20   }
21 }
```

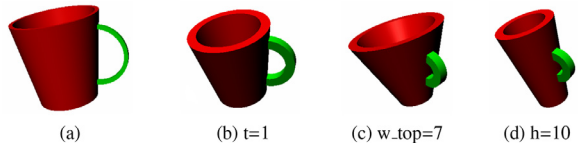


Fig. 3. (a) A visualization of the shape generated by CoffeeMug.psm (Algorithm 1) (b–d) shows how varying PSML program variables introduce semantic variations to the generated coffee mug models.

and are used to specify shape grammars. These grammars use the passed shape and argument variables and locally defined variables to generate an instance of the grammar shape, e.g., the grammar of Algorithm 1 generates a "coffee mug." The algorithm defines a conic frustum (line 6) with a hole (lines 7 and 8) that make up the body of the cup. The handle is a green (line 12) cylinder (line 6) with a hole removed from center (line 11) and cut in half (line 10), aligned with and touching the side of the cup (as per its defined positioning). Fig. 3 shows various realization of the coffee mug object and demonstrates how the representation enforced shape constraints despite several parameter variations, e.g., the handle and mug vessel surfaces connect consistently for these variations. More generally, PSML's syntax for detecting the size and position of the current volume allows the user to develop shapes that re-organize their sub-components consistently over parametric variations, e.g., anisotropic scaling.

3.1. PSML programs

Algorithm 1 shows an example of a PSML program file. As in Java, the program file name must match the name of the grammar declaration which begins the program. The program body is defined with braces '{}'. As in Java/C++/C, braces '{}' define scope delimiters and are generically used to denote the beginning and ending of execution blocks for flow control statements, e.g., if/else conditionals, for loops, while loops, etc. They also delimit our newly defined *rules* blocks which contain PSML shape grammar rules. Scopes also serve to control variable resolution in a manner similar to Java.

The body of each PSML grammar may include variable declarations and initializations as well as a collection of methods which are analogous to functions. Methods may only be called from shape grammar rules and are invoked when they appear as a non-terminal successor symbol within a rule of some shape grammar (see Rule Block and Rules for details). The *import* directive serves to import grammars and their methods to another grammar and determines the collection of methods which may be called from any grammar (similar to Java import for classes). Rules which invoke grammar methods transfer control to the statements of the selected method.

3.2. Rule blocks and rules

Each method must include a rules block which is denoted by the keyword *rules* and has a scope as defined by matching braces. Each rules block contains a set of shape grammar production rules where each production rule has the generic form as specified in (1).

$$predecessor : condition : successor ; \tag{1}$$

Execution of a production rule causes the non-terminal symbol referred to as the *predecessor* to be replaced by the *successor* which may be one or more terminal or non-terminal symbols. The *condition* term is a branching expression: if *condition* evaluates as true, then the rule is executed; else, the next instance of a production rule having the same predecessor is executed. The

syntax of PSML rules are inspired by those described for L-systems in [3] but differ by not allowing for in-line scripts within rules. These scripting functions have been replaced by the availability of successor *methods.*

Multiple rules may be written for a given non-terminal symbol to allow for distinct behaviors based on the evaluation of the rule *condition.* Precedence for these rules is defined by the order that these rules appear within the grammar, where the first rule for the symbol has highest precedence. One may designate a new shape to be a real-world object by indicating the shape is terminal using the special string *terminal* (see Algorithm 1, lines 8, 11, 12). Terminal symbols do not appear as predecessors in any production rule and are visible elements that exist in the final 3D model with the exception of terminals declared to be void() (see Section 3.3, *Modeling void-space* for details). While the treatment of non-terminal symbols in PSML is similar to [45], a key difference is PSML's ability to associate semantic labels to terminal and non-terminal symbols which allows further analysis to be conducted based on the semantic meaning of shape elements.

The way PSML processes rule blocks has several advantages. Since blocks are part of the language itself, rules have direct access to variables available within the sequential scope in which they are defined. In addition, the rule blocks themselves can be regarded as variables within the same scope, allowing subsequent sequential code to access and post-process shapes created in an earlier rules block.

3.3. Rule functions

Each rule within a shape grammar may include a sequence of one or more special functions which are referred to as rule functions. These functions are inserted before the list of successor symbols and serve to transform the predecessor shape into the successor shape(s), e.g., changing the shapes diffuse reflectance or color (see Algorithm 1, lines 8, 11). PSML rule functions include special functions for creating and transforming (rotating/translating/scaling) objects as well as the *split*() and *repeat*() operations as described in [1] and [2]. However, noticeably absent from the list of available operations is the component split or *Comp*() operation which divides 3D volumes into shapes of lesser dimension, i.e., faces, edges or vertices (see Fig. 2(a) for an example of a face split of a cube). Omission of this function effectively constrains all our grammar shapes to remain volumetric and preserves the semantic interpretation of each terminal as a virtual object. This tends to produce more complex models, yet these models encode important information which allows PSML to perform operations that may be difficult to deal with using shell-based representations. The following sections provide a complete list of available rule functions.

3.3.1. Creating shapes instances

This function replaces the predecessor shape with a completely new shape and has syntax: I(String *type*, double[] *params*) where *type* indicates the shape to be created and *params* define the parameters necessary to create the shape. The function may be invoked multiple times and successor symbols are associated with each created shape in the same sequence as they are created within the rule. For example, in Algorithm 1 (line 5) a conic frustum is created and associated to the symbol *vessel* and a cylinder is created and associated to the symbol *handle_c.*

All instances draw from a pre-defined set of 3D shape primitives. There are six basic primitives: {*box, cylinder, sphere, cone, ramp, tetrahedron*}. Each primitive may be subdivided/split along its principle axes into other derived shapes, making it possible to create 19 different primitives as shown in Fig. 4. For example, a conic ring (the side of the coffee mug) is generated on line 7 by splitting the conic frustum into two parts having different radii and the mug handle is generated on line 10 by splitting a cylinder into a ring (the handle) and a space. Triangulations of these primitive shapes are pre-defined in the PSML built-in function library.

Similar to [46], PSML supports cylindrical, and spherical coordinate systems in addition to the Euclidean coordinate system. The Euclidean coordinate system is used for boxes, ramps, and tetrahedrons. The cylindrical coordinate system is used for cylinders and cones, and the spherical coordinate system is used for spheres and enables PSML to model architecture with round geometry.

3.3.2. Split/repeat

These functions replace the predecessor shape with a sequence of new shapes and require two arguments. The first argument for these functions indicates the coordinate axis of the the split, i.e., new shapes will be defined by slicing through the predecessor shape with a plane perpendicular to the indicated axis direction. The second argument determines where these divisions will spatially occur along that axis (see Algorithm 1 lines 6, 7, 9, 10). The repeat function takes in an optional third argument which is used to introduce an offset for the initial division which is particularly useful in structures with offsets such as sequential rows of bricks in a wall.

Note that split operations act along axes of the coordinate system defined for each shape. Rules may be adapted to different primitives by specifying primitive-sensitive code, either using sequential flow-control statements, e.g., if/then, or using conditional rules that only execute on specific primitive types.

3.3.3. Spatial transformations

These functions operate on the predecessor shape by changing the linear transformation that determines the size, position and orientation of the object. There are three transformation operations: *scaling, rotation* and *translation.* The scaling function is denoted by $S(s_x, s_y, s_z)$ and scales the predecessor shape by adjusting its bounding box in *xyz* by a factor given by s_x, s_y and s_z respectively. The rotation function is denoted by $R(\theta, \phi, \psi)$ and specifies a rotation matrix as a sequence of axis-aligned rotations (Euler angles) around the *xyz* axes; in that order. The translation function is denoted $T(t_x, t_y, t_z)$ and translates the predecessor shape by moving the center of its bounding box in *xyz* by an amount of t_x, t_y and t_z respectively. These functions may be invoked in combination with instancing or constructor functions to create and place primitives which is often a part of the axiom rule of a PSML shape grammar.

3.3.4. Appearance

A generic function is provided to alter the appearance of the predecessor shape. The function is denoted

$$appearance(attrType,\ attrValue)$$

where *attrType* is a string value that specifies the type of appearance attribute and *attrValue* denotes the values that the attribute will take. The following list describes the attribute types which may be modified directly in PSML: *color*, *material*, *ambient*, *emissive*, *diffuse*, *specular*, *shininess*, *bump, bumpmap*, *bumpweight*, *texture*, *transparency*.

3.3.5. Void-space

A special function *void*() is used to make the associated symbol an invisible terminal in the final model (see Algorithm 1 line 13). Voids specified in PSML models are particularly useful for void-space analysis, i.e., detecting context-sensitive situations that require special-purpose placement of semantic objects. As mentioned earlier, this ability to label empty space terminals is one of the key differences from previous volumetric approaches such as [27,45,47]. Formal definitions for these spaces and the ability to query and operate on these spaces afforded by PSML shows

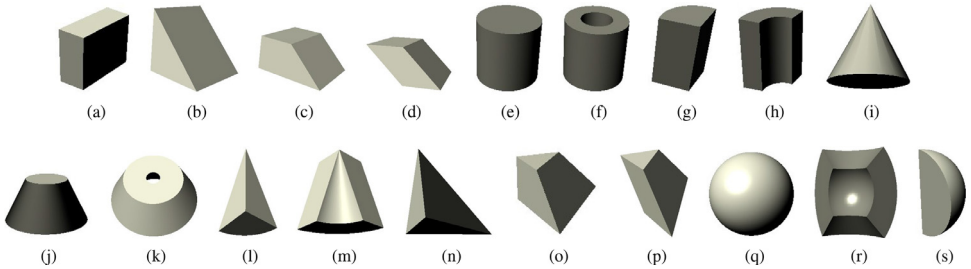


Fig. 4. (a–s) Depict 19 different volumetric primitives which may be used in PSML programs to construct 3D shapes. These shapes are generated by splitting one of six basic primitive shapes which are (a) the box, (b) the ramp, (e) the cylinder, (i) the cone, (n) the tetrahedron, and (q) the sphere. Other shapes are derived from these basic shapes as follows: (c,d) are derived from (b), (f–h) are derived from (e), (j–m) are derived from (i), (o–p) are derived from (n), and (r, s) is derived from (q).

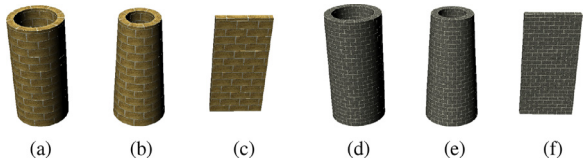


Fig. 5. (a) A sequence of models generated by running the PSML program Bricks.psm (Algorithm 2). (b) Demonstrates the utility of the *useAttributes()* function, in particular, this model is generated by substituting "rock" for "sand" on line 26. Lines 4, 5 show the utility of the *instanceof* operator for determining the coordinate system of the passed primitive which enables distinct methods for placing the bricks that depends on the coordinate system of the passed primitive. Lines 7–10 demonstrate the utility of attributes and their syntax when used within a method.

promise for enabling new approaches for automatic building layout design [20] and automatic placement of furniture [21–23]. Examples of uses for void spaces are shown in Figs. 1(c, d), 9 and 10 and will be further discussed in the Results section.

3.4. Attributes

Procedural shape models often include a large number of constants used to define object attributes such as their appearance, size, position, adornment, etc. In many cases specific choices for these constants determine the "look-and-feel" of the generated shapes. Since the constants merely determine the starting values for variables within the shape grammar, we created attribute files as a mechanism for making these constants external to the shape program.

Inclusion of attribute files allows users to interact with the shape generation process without the requirement that they understand PSML code. In this context, users modify the attribute files and generate instances of different shapes from the grammar. This provides inexperienced users methods to control the shape generation process and give their models visually distinct styles (see Fig. 5). We refer to these constants collectively as attributes and PSML provides a rule function,

$$useAttributes(filename, attrGroup)$$

where *filename* references an external attribute file such as that shown in Algorithm 3 and *attrGroup* denotes a specific collection of attributes to load from the attribute file. Each call to this function loads a collection of these constants from a file and makes their values available to subsequent rules. Context-sensitive uses of this rule function allow a single shape grammar to exhibit different geometry and appearance depending upon the context of the rule. After invoking useAttributes, the loaded values are referenced using the prefix '@', and then writing the attribute name (see Algorithm 2: line 26 loads an attributes file, lines 7–10 reference brick dimension attributes. Algorithm 3 shows the contents of referenced attribute file).

Algorithm 2. The contents of a PSML file: Bricks.psm. Discussion of the program is provided as part of Fig. 5.

```
1  public class Bricks extends ShapeGrammar {
2    float[] defaultBrickDims = {0.4, 0.25, 0.2, 0};
3    public Bricks() {
4      float isEuclidean=
5          myShape instanceof Shape3D.CartesianShape;
6      float isCylindrical=
7          myShape instanceof Shape3D.RotaryShape;
8      rules {
9        parent::repeat(y,{@brick.height,@brick.height},0){
             even, odd};
10       even:isEuclidean:repeat(x, {@brick.width},0) {
             brick};
11       odd:isEuclidean:repeat(x,{@brick.width},@brick.
             width/2){brick};
12       even:isCylindrical:repeat(theta,{@brick.width/
             myShape.r},0){brick};
13       odd:isCylindrical:repeat(theta,{@brick.width/
             myShape.r},(@brick.width / myShape.r)/2){
             brick};
14       brick::appearance(texture,@brick.texture)
15           appearance(specular, {0,0,0}){terminal};
16     }
17   }
18   public static void main(String[] args) {
19     rules {
20       axiom::T(5, 0, 0)I(box, {4, 8, 0.5})
21           I(conicfrustum, {1.5, 2, 8})
22           T(-5, 0, 0)I(cylinder, {2, 8})
23           {wall, cone, cyl};
24       cone::split(r, {1.5, 0.5}){space, wall};
25       cyl::split(r, {1.5, 0.5}){space, wall};
26       wall::useAttributes(brick.properties, sand){Bricks
             ()};
27       space::void(){terminal};
28     }
29   }
30 }
```

Algorithm 3. The contents of PSML attribute file: brick.properties referenced on line 26 of Bricks.psm from Algorithm 2.

```
1  attributes sand {
2    brick.width = 1.5;
3    brick.height = 0.6;
4    brick.texture = sandStone.jpg;
5  }
6  attributes rock {
7    brick.width = 0.7;
8    brick.height = 0.3;
9    brick.texture = rock.jpg;
10 }
```

3.5. The myShape and scope variables and Namespaces

As mentioned earlier, methods may be invoked from PSML shape grammar rules. These methods are implicitly passed the shape of the appropriate successor symbol for the invoking rule. This shape is assigned as a special *Shape* variable in the invoked method where the identifier of the variable is *myShape*. The *myShape* variable is similar to the *this* reference available in many object-oriented programming languages and carries with it many important attributes of the shape passed to the method. These

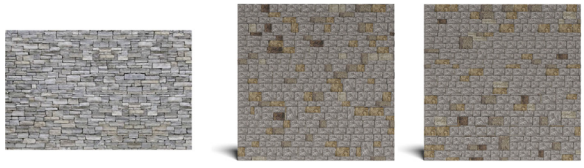


Fig. 6. Irregular patterns: a) is an image of an actual stone wall in reality with irregular patterns. b) Shows two of our realizations of a randomized stone wall using a randomized recursive grammar.

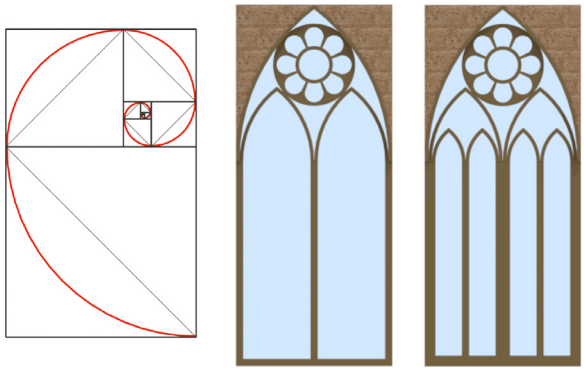


Fig. 7. Recursive nested patterns: (a) a Fibonacci spiral generated by a 7-line recursive grammar. (b) A Gothic window generated with two levels of recursion. (c) A Gothic window generated with three levels of recursion.

include the *primitive type* of the passed shape, the *coordinate system* of the passed shape and *values* for all of the parameters which define the shape. Namespace variables may be applied to detect the primitive type of the passed primitive shape using the *instanceof* operator. This operator is intended to behave in exactly the same way as the Java *instanceof* operator. However, this operator serves to detect instances of different predecessor shapes or to detect the preferred coordinate system of the predecessor shape. There are 22 namespace variables, 19 of these account for each primitive shape type as shown in Fig. 4 and an additional 3 are provided which detect only the coordinate system type of the passed primitive type (see Algorithm 2 lines 4, 5). Sequential statements may evaluate expressions that use the *myShape* variable to perform context-specific rule blocks as shown Fig. 5.

The *scope* variable is very similar to *myShape*. The difference is that *scope* refers to the predecessor of a rule. This allows access to the dimensions of the immediate predecessor in each rule and facilitates the construction of recursive grammars as explained in the next section.

3.6. Recursion

Recursion is an intrinsic property of grammars. A recursive grammar is one that has a non-terminal β that produces further down the derivation tree a successor that is β. Recursion allows PSML to produce complicated shapes, such as irregular patterns (Fig. 6), or nested patterns (Fig. 7) that are hard to achieve using sequential code such as [47]. PSML provides the scope variable as means to check the attributes of the current non-terminal and stop the recursion if some criteria is met, which is used to terminate the shape derivation for Fig. 7(a).

3.7. Execution model

Statements are divided into sequential statements, i.e., Java-style code, and rule blocks, i.e., shape grammar code. Execution begins at the *main()* method of the designated initial PSML grammar. The *main()* method is passed the unit cube as it is initial shape. Statements in each method are executed sequentially until a shape grammar is encountered as indicated by a *rules* block. Within each rules block, a shape grammar is defined which consists of a collection of rules. The first rule of the grammar is taken as the *axiom* of the grammar, i.e., the initial production rule which is applied using the shape passed to the method as a predecessor. Execution of the shape grammar continues by looking up rules/methods to expand each successor shape until all of the symbols are terminal symbols. The specific sequence of executed production rules generates a *parse tree*. This tree records the sequence of production rules as a hierarchy where non-terminal symbols are the internal nodes of the tree and terminal symbols are the leaves of the tree. As in [2], a scenegraph is used to track the sequence of grammatical productions executed to generate each terminal object, i.e., the parse tree (or derivation tree). The resulting scenegraph includes a hierarchy of semantically meaningful groupings of objects from very large collections of objects in the vicinity of the root, e.g., the floor of a building, down to indivisible elements at the leaves, e.g., a brick of a building. The end of each rule block may be followed by additional code that may query the scenegraph using the *terminals()* or *instances()* functions and operate on the returned symbols to make decisions regarding the addition of model details or take other needed context-specific actions (Section 3.8 explains this in more detail). Execution ends when the last statement of the *main()* method has been executed.

The execution model for the shape grammars is depth-first, i.e., the interpreter expands the first successor completely to all of its terminal elements before proceeding to the next symbol. This choice of execution model for PSML limits some functionality afforded by priority-based execution models as described in [2]. Specifically, priority-based grammars allow the programmer to require specific rules to be evaluated (executed) before other rules are evaluated. This may be used to create geometry that will impact a context sensitive criterion such as visibility tests in subsequent rules. However, these limitations are somewhat offset by the fact that correct utilization of priority values within priority-based grammars requires the user to be aware of the possibly complex relationships between the priorities assigned to all of the rules in order to predict the model construction outcome. For models involving a large number of grammars, each of which may incorporate a separate priority scheme, tracking the priority of a given rule for a specific derivation can be complex for both the programmer and interpreter. Hence extensive use of priorities has the potential to confuse the execution sequence and make it more difficult to predict the derived model given a particular input. For this reason, we feel the benefits afforded by a priority-based execution model is offset by the added complexity required of the user to effectively exploit rule priorities.

3.8. Boolean operations on shapes

PSML provides unique functions, described in the next paragraph, that allow programmers to access and operate on derived non-terminal and terminal shapes from sequential code. This is accomplished using a *shape path*. The shape path for a specific parse tree node is created by tracing the ancestry of that node and concatenating the shape names of each parent in this ancestry using the '/' character as a name delimiter. For example, in Algorithm 2 the shape path "Axiom/wall/Bricks/brick" refers to the "brick" nodes generated as children of the "wall" non-terminal.

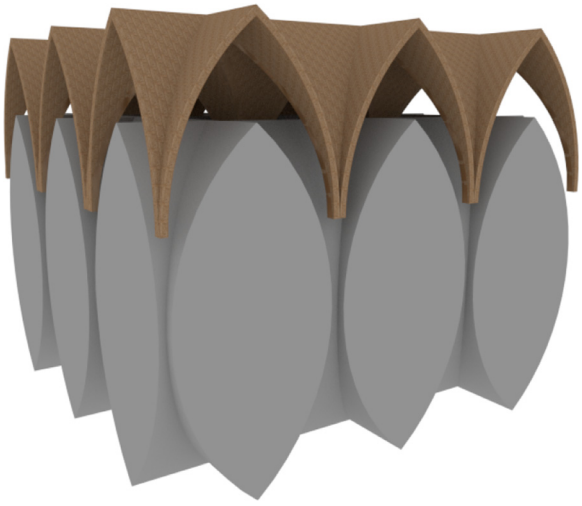


Fig. 8. Boolean operations are used to create a series of Gothic groin vaults. This is accomplished by creating a collection of volumes (in gray) that serve to remove bricks from the overhead ceiling using a Boolean subtraction operation. The remaining bricks (in brown) form the stones of the groin vault.

PSML provides two access functions that allow sequential code to reference shapes generated in shape grammars (rule blocks): (1) *terminals*(*symbolPath*)and (2) *instances*(*symbolPath*). The parameter *symbolPath* is interpreted as a "match any" regular expression, i.e., the shapes returned will be all nodes whose symbol path can match, in order, the elements of the *symbolPath*, e.g., the path "wall/Bricks" references all shapes having symbol paths that match the regular expression ".*wall.*/.*Bricks.*"

This allows programmers to intuitively reference large and possibly complex arrangements of shapes using this semantic "shorthand." By default, the search starts in the parse tree at the sub-tree associated with the invoked method but prepending the string with a '/' character initiates the search from the parse tree root. Both functions return an array of shape variables. Context sensitive operations can then be accomplished using sequential logic that inspects the attributes of these shapes such as scope, geometry, orientation, appearance, etc.

Two separate access functions are necessary as they provide different types of references to these nodes. Specifically, the *instances*() function returns a collection of non-terminal shapes, i.e., shapes that semantically group together terminals but are not visible in the final model. The *instances*() function returns *copies of* the referenced non-terminal geometries. These shapes can be used to test to detect context-sensitive situations and to create new shapes. In contrast, the *terminals*() function returns *references* to shapes and allows the user to modify the geometries of all terminal shapes that are descendants of the referenced shape. At the moment, shapes returned by the *terminals*() function are modified by applying Constructive Solid Geometry (CSG) Boolean operations on the elements of the associated shape array as (see Fig. 8).

Since the number and order of symbols generated within a rules block cannot be tracked or a-priori known in general, both of these functions *always* return an array of shapes and this array may have length 0.

Boolean operations may be invoked in sequential code using the function

$$Shape1.geometricBoolean(Shape2,\ op)$$

where *Shape*1 and *Shape*2 are Shape variables and *op* is a String that denotes the type of operation. Valid values for *op* are "+" (union), "-" (difference), or "&&" (intersection). These Boolean operations produce new shapes which may then be queried by the size of their bounding box using scope variables or by their volume using the special function volume() which may be invoked on any shape primitive. Fig. 10 shows how these functions can be used for the context-sensitive placement of a door based on the presence of a walkway.

Table 1
The table summarizes how Boolean operations impact the geometric shape of symbols as a function of the type of Shape passed as the 1st and 2nd operand to the geometric Boolean operation. We refer to the 1st operand as Shape 1 and the second operand as Shape 2 and NT denotes non-terminal shape variables (generated by invoking instances()) and T denotes terminal shape variables (generated by invoking terminals()).

Shape 1	Shape 2	Operation	Result
NT	NT	Any	New NT
NT	T	Any	New NT
T	NT	Any	T changed
T1	T2	+, &&	T1 changed, T2 deleted
T1	T2	-	T1 changed

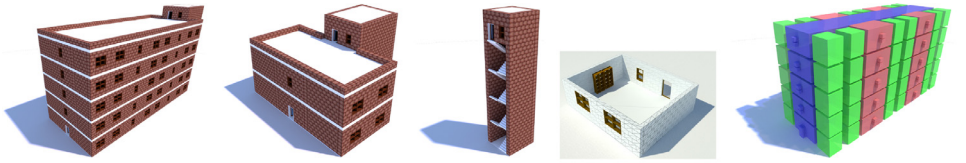
Boolean operations typically generate shapes that cannot be modeled as one of the 19 shapes from Fig. 4. Hence, they cannot be used directly as initial symbols for new shape grammars, i.e., rules blocks. To cope with this issue, rules blocks that use Boolean shapes as their predecessor symbol are passed the bounding box of the Boolean shape as the initial shape for the rules block as seen in Fig. 10 when the bounding box of the intersection is passed to the rule block which performs the subtraction and replacement with a gate.

The impact of a Boolean operation on a shape variable depends upon the type of reference associated with both of the involved operands. When the first operand, i.e., the left-hand operand, is not a terminal reference, i.e., the first operand is a shape generated by a call to *instances()*, the Boolean operation generates a new shape and does not effect geometry of the parse tree shapes. When the first operand is a terminal reference, all of the terminal shapes referenced by the variable are acted upon with the Boolean which can change the geometry of the terminal shapes of the parse tree. When both the first and second operands are terminals and the Boolean operation is a union or intersection, the geometry of the first operand receives the result of the Boolean operation and the terminal shapes referenced by the second operand are deleted. These rules are summarized in Table 1. Addition of Boolean operations for modifying terminal shapes and for initiating new rules blocks enables a rich variety of context-sensitive behaviors and greatly expands upon the representation capabilities of the language.

Boolean operations on polygonal models have proven to be difficult to implement due to numerical instabilities that occur when determining the intersection of polygonal models. Recent approaches [48–50] use exact predicates to determine when and where surface intersections occur and address many of these shortcomings. In this work, boolean operations are completed using the Carve software library [51] which has been used for CSG operations by other open source projects, e.g., Blender.

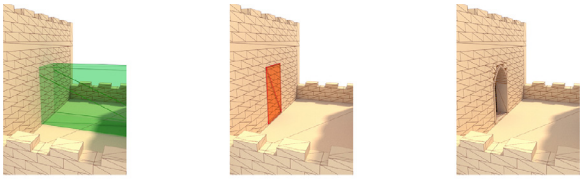
3.9. PSML development tools

We have implemented an Integrated Development Environment (IDE) for writing PSML code by customizing Oracle's Netbeans software. Through an ANTLR supported grammar (http://www.antlr.org/), the IDE supports PSML syntax and programming, which includes syntax highlighting, indentation and code completion. The prototype Netbeans IDE has an integrated PSML interpreter and allows users create, edit, run and visualize models specified using PSML. A collection of PSML programs are also available for download. These models range from simplistic



(a) (w,h,d)=(15, 20, 36) (b) (w,h,d)=(20, 10, 10) (c) The *Stairwell* subtree (d) An *Office* subtree (e) *Void* spaces colored by function

Fig. 9. PSML is used to model an office building. (a, b) Show two models generated using different (**w**idth,**h**eight,**d**epth) dimensions for the building. (c,d) Show sub-structures extracted as sub-trees of the global parse tree. (e) Depicts the 3d void spaces of the build and color-codes these spaces by function (see Section 4.1 for discussion).



(a) (b) (c)

Fig. 10. (a–c) depicts how void space can provide context-sensitive geometry generation. (a) shows a part of the castle terrace from Fig. 1(d) and an adjoining tower with a walkway void on top of the terrace in green. (c) shows the Boolean intersection between the walkway void and the adjoining tower in orange. (d) shows the result of subtracting the intersection volume and invoking the *GothicDoor()* grammar on the subtracted volume to replace this volume with a Gothic arch opening.

models similar to those used for explanations above to more advanced geometries such as the castle, office building and cathedral shown in Figs. 1, 9, 11, and 13.

4. Results

Our results are summarized through the detailed analysis of 5 PSML models: (1) an office building, (2) a medieval castle gate, (3) a Gothic cathedral, (4) a populated bookcase and (5) a Puuc Mayan house. All PSML programming and model generation was performed using the developed Netbeans IDE on a a laboratory desktop workstation with 16 GB of RAM and a 2.67 GHz Intel i5 Quad-core CPU running Linux. PSML models shown in the figures below were exported in Maya's Wavefront (OBJ) file format and rendered using the open-source 3d modeling software Blender.

4.1. Office building

The PSML code for the office building took approximately 5 h to specify and includes 25 distinct dimensions of variation which include dimensions for doors, windows, stairs, bricks and three alternative styles for windows, i.e., separate window styles for offices, the stairwell and the hallways. Fig. 9(a,b) show two instances of the PSML office building created by varying the width, height and depth of the building bounding box. The larger building (a) took 2.57 s to generate (not including the OBJ file exporting time, same below) and (b) took 1.2 s to generate. The PSML code includes 15 grammar files and 7 of these grammar files are "generic." Generic grammars are shape grammars that can be re-used in other buildings. Generic grammars used as a part of the office building model serve to generate the following sub-structures: *Bricks*, *Doors*, *Windows*, *Frames (for Doors and Windows)*, *Stairs and UStairs (two stairways that share a common platform)*. Fig. 9(c, d) shows renderings of the *Stairwell* and *Office* non-terminals and depict how large collections of geometries can be retrieved using relatively simple semantic queries into the parse tree. Fig. 9(e) provides a visualization of the void-space within the office building which consists of three distinct types: (1) blue regions denote voids associated with the *East-West hallways*, (2) red regions denote voids associated with *North-South hallways* and (3) green regions denote *office* voids.

Table 2
A count of element instances and volumes for the office building models shown in Fig. 9(a,b).

Name	Num. Fig. 9(a)	Num. Fig. 9(b)	Vol. Fig. 9(a)	Vol. Fig. 9(b)
Wall internals	4	4	27.78	16.02
Stairway steps	100	40	22.00	10.30
Stairway platforms	5	2	10.15	4.75
Window, door frames	2508	336	66.69	8.80
Roof railing parts	62	10	2.60	0.42
Bricks	14537	4078	2466.3	721.8
Floors	7	4	1294.86	250.62
Stairway base	95	38	10.45	4.89
Ceilings	1	1	22.80	22.80
Window, door glass	477	64	4.40	0.68
Exterior molding	63	11	4.52	0.79
North-South halls	100	0	1542.82	0
East-West halls	25	10	3491.31	414.83
Office spaces	60	8	1624.90	645.12

Use of the*instances()*, *terminals()* and *volume()* functions allow us to collect statistics on the frequency and volume of each semantic element in the models. The statistics for the model shown in Fig. 9(a) are provided in Table 2. Since these functions are available to sequential code logic, users can use PSML to construct shape optimization programs which search the space of shapes described by the PSML program (sometimes called the language of the grammar) for models that satisfy specific desirable volume or dimensional criteria.

4.2. Medieval castle gate

The medieval castle gate model is a good demonstration of utilizing void space to create context-sensitive grammars. This PSML model is a representation of the specific gate complex at Apollonia-Arsuf. Apollonia-Arsuf is a fortress constructed approximately 10 miles North of Tel-Aviv in the 12th century which was razed after being captured by the Mamluks during the 3rd Crusade in the 13th century. The PSML code required approximately 15 h to write and the resulting model includes 36 dimensions of variation which include dimensions for the towers, bricks, crenellation, arrow slits, doorways and the gate and portcullis complex.

Fig. 1(c,d) shows two instances of the PSML castle gate where the depth of the balcony (overhang) at the top of the tower and the overall height of the two towers varies. Note that the model deforms in a meaningful way and parts of it are inserted/removed in a context-sensitive manner. In particular, when the bridge is at the same level as the towers, inner crenels are removed as they do not serve a purpose in positions other than the edges of the roof. In addition, doors providing access to the bridge are only created when there is an intersection between the walking space of the bridge and the tower's side wall. Both of these behaviors are accomplished through the use of void space. Fig. 10 demonstrates the creation of the doors.

The model shown in (c) took 9.2 s to generate and the model in (d) took 11 s to generate. The PSML code includes 12 grammar

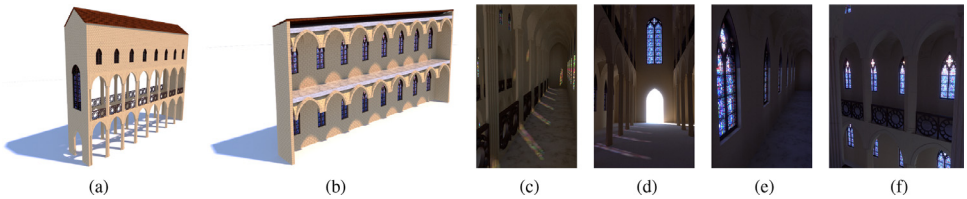


Fig. 11. A PSML model of a Gothic cathedral is shown. (a) shows an exterior view of a realization of the cathedral. (b,c) show large-scale components of the cathedral and (d) shows how these components combine. (e–g) show views of the interior of the cathedral that depict details in the model not visible from the outside.

Table 3
A count of element instances and volumes for the castle gate models shown in Fig. 1(c,d). Entries with * have volumes computed by bounding boxes.

Name	Num. Fig. 1(c)	Num. Fig. 1(d)	Vol. Fig. 1(c)	Vol. Fig. 1(d)
Tower roof	3	3	62.5	81.3
Bridge roof	16	16	73.75	46.15
Vault stone	28	36	3.06	3.94
Arch stone	8	12	2.97	4.01
Walls	8	4	44.10	51.98
Bricks	8870	12224	1038.61	1444.61
Portcullis	1	1	1.86	1.86
Gate, Gothic door	4	6	61.26 *	73.4 *
Totals	**8938**	**12302**	**1288.11**	**1707.25**

files and 5 of these grammar files are "generic." Generic grammars used as a part of the castle gate model serve to generate the following sub-structures: *Arch, Bricks, BarrelVault, Crenellation, GothicDoor* and can be re-used in other Gothic architectural models such as the Cathedral models in Fig. 11. An accounting of the volumes within the castle gate model is provided in Table 3.

4.3. Gothic cathedral

This PSML code represents the architecture of a Gothic cathedral. This model, and particularly its interior, is an example of how PSML can be used to represent complex and highly-detailed architecture. The code performs many Boolean operations to create the groin vaults and the Gothic windows, which causes the program to take 1 min and 18 s to complete. The model contains 49 Gothic windows (~30 Boolean operations per window) and 32 groin vaults (~100 Boolean operations per vault). The number of Boolean operations required is inversely proportional to the size of the vault stones and bricks. Fig. 11(a–f) show sub-tree components of a global cathedral model (not shown). Images (b,c) show instances of the nave and aisle of the cathedral as separate components. (d) shows how these two elements are combined to construct the cathedral model. Fig. 11(e–g) are renderings of the interior of the cathedral and highlight the details present in the interior of the model.

4.4. Bookcase

The bookcase grammar demonstrates the use of object-oriented design and post-processing to populate empty space. The bookcase model consists of a shelving unit populated with books and vases, as shown in Fig. 12. The shelving unit is generated by a very small grammar that creates a uniform grid of cells. The books and vases are generated by two other grammars that populate their scope with a randomized arrangement of books or vases. In this case, the free-form vase object is imported as a polygon model (in.OBJ Alias-Wavefront format) that is substituted into the empty shelf space volume (see Fig. 12(a)-right). If the imported model is too large to fit into the bounding box of their non-terminal parent shape, it is appropriately scaled-down to meet this constraint.

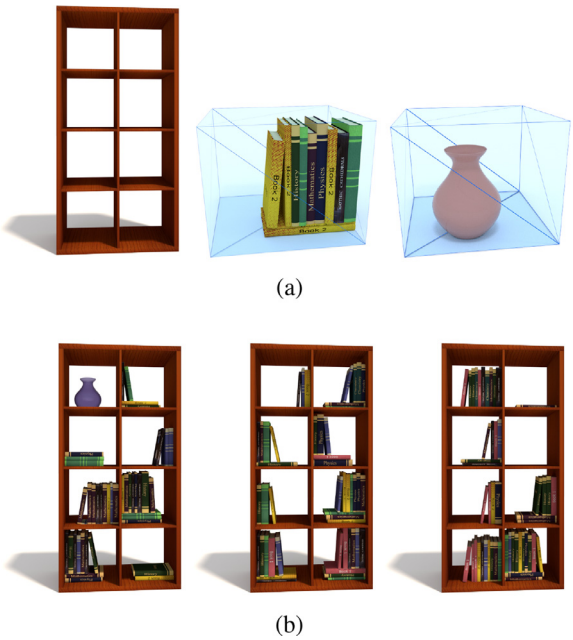


Fig. 12. A populated bookcase model generated by combining different independent components. a) shows each of the components separately. Those components are, from left to right, an empty shelving unit with labeled space for the empty cells, a book stack that populates a box (shown in blue) with a randomized stack of books and a vase, which is randomly placed inside a box. b) shows three different realizations of the populated bookcase. The populated bookcase is generated using a fourth grammar consisting of 5 lines, which creates the empty shelving unit, queries the cell space, then randomly creates book stacks or vases in some of the selected cells.

The shelving unit, books and vases grammars are completely independent and have been written separately. Another grammar combines them together and creates the populated bookcase. This shows that PSML is capable of supporting object-oriented concepts that allow for complicated structures to be built while keeping individual components relatively simple.

4.5. Mayan house

PSML programs represent entire classes of volumetric shape models. In this example we show how such models can be applied for physical simulation. Specifically, a PSML model of a Mayan Puuc building from the classical period is shown in Fig. 13(a). Fig. 13(b–d) shows keyframes of the model collapse due to a simulated earthquake event (oscillation of the ground plane). By varying the model derivation, researchers, e.g., archaeologists, we see possibilities for performing virtual experiments that may lead to better understandings of how the size/volume and shape/organization of buildings relate to their collapsed remains for architectural structures.

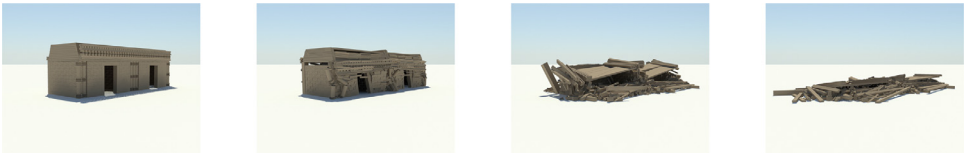


Fig. 13. Images from left to right show the physical simulation involving a Mayan building constructed using PSML programs. Since PSML models consist of 3D volumetric objects, they can be used directly in physical simulations.

Table 4
A summary of PSML code structure and length for models of this article.

Fig. model	PSML functions and subclasses	Lines of code	Totals
Fig. 1(a,b) OfficeBuilding	Building layout		15
	Floor Layout,Cluster, Room, Roof	(65, 6, 14, 15)	100
	Wall, WallSegment, FullWall, Door	(28, 27, 19, 38)	112
	Window, WindowFrame, DaylightWindow	(11, 23, 25)	59
	Stairs, Staircase, StaircasePlatform	(35, 34, 25)	94
		Total	**380**
Fig. 1(c,d) CastleGate	Building Layout		15
	InnerWalls, OuterWall, Corners, Crenels, Doors, Bricks, Arch	(56, 54, 25, 78, 13, 10, 26)	262
	Gate, Rectangular Tower, Circular Tower	(13, 88, 43)	144
		Total	**421**
Fig. 11 Cathedral	Building Layout		355
	Barrel vaults, Groin vaults, walls	(22, 32, 88)	142
	Gothic doors, windows, tracery	(29, 75, 36)	140
		Total	**637**
Fig. 12 Bookcase	Layout	**Total**	**19**
Fig. 13 MayanHouse	Building Layout		5
	Roof Layout, Column, Walls, Ends	(122, 61, 19, 26)	228
	House Layout, Doors, Walls, Columns	(32, 9, 120, 35)	196
		Total	**429**

4.6. Programming effort analysis

Statistics of the PSML program code were gathered to approximate the effort required to generate the models of this article. For our analysis we tabulate the lines of code required to derive the geometry and exclude the lines of code written for visualization of the derived geometry which is also possible using the development environment. Table 4 includes the lines of code for each PSML class where the supporting sub-classes are explicitly shown. PSML sub-classes are group by their semantic purpose (generating walls, generating roofs, etc.) and each sub-class serves to generate substructures within the larger model. Their lines of code are listed individually (column 3) and then summed in the "total" column (column 4) of the table. Subtotals are provided for each structure and give an indication of the number of code lines required to generate the PSML model. It is important to note that no effort has been made to optimize the code and code that is often re-used across structures (e.g., brick walls) is counted multiple times in Table 4. As such, it is entirely possible that the numbers presented could be significantly reduced using alternative PSML programming approaches.

5. Limitations

The proposed language and methods for generating volumetric models are limited by the vocabulary of the grammar, i.e., only those shapes which can be described as a collection of the volumetric elements from Fig. 4, can be represented. While this limits the flexibility of the proposed shape grammar for representation of completely generic shapes, the breadth of shapes that can be represented using this grammar is quite large; especially with the incorporation of CSG Boolean operations on PSML shapes. Current work on PSML investigates methods to expand the applicability of the proposed language for modeling geometries that are not well parameterized by a Euclidean, cylindrical, or spherical coordinate system.

6. Conclusion

This article has introduced a volumetric-only approach for procedural modeling of shapes and a new Procedural Shape Modeling Language (PSML) that facilitates the implementation of this approach. PSML introduces a distinct programming construct for procedural model generation that borrows structure and syntax for sequential statements from Java and borrows structure and syntax for shape grammars and their production rules from L-systems. PSML supports an object-oriented approach for designing models which enables creation of complex models by assembling simpler models. While creating new models is relatively easy, the language requires the user to have a basic understanding of object oriented programming to use the language efficiently. New users can download the PSML program development IDE, read documentation and run example code from resources made available at the following GitHub url: **<link removed for review>**. Additionally, a number of new functions are introduced for use in both shape grammar contexts and sequential contexts which allow for context-sensitive generation of highly detailed volumetric models. Such models are of importance for Building Information Modeling (BIM) systems and the related issue of closely documenting and tracking the complex structural state of historic structures. The resulting models are often dense in terms of their resource utilization, i.e., they often require a large number of polygons and include details that may not be required for visualization at a distance. However, this dense collection of information may be processed by client applications which may extract portions of the model of interest. The structure of the language and its execution model is detailed and the application of PSML for generating context-sensitive volumetric operations and for physical simulation is demonstrated.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research has been partly funded by the US National Science Foundation.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.gvc.2021.200018.

References

[1] Wonka P, Wimmer M, Sillion F, Ribarsky W. Instant architecture. ACM Trans Graph 2003;22(4):669–77.
[2] Mueller P, Wonka P, Haegler S, Ulmer A, Van Gool L. Procedural modeling of buildings. ACM Trans Graph 2006;25(3):614–23.
[3] Prusinkiewicz P, Lindenmayer A. The algorithmic beauty of plants. Springer; 1991.
[4] Belhadj F, Audibert P. Modeling landscapes with ridges and rivers: bottom up approach. In: Proceedings of the 3rd international conference on computer graphics and interactive techniques in Australasia and South East Asia, GRAPHITE '05. New York, NY, USA: ACM; 2005. p. 447–50. ISBN 1-59593-201-1.
[5] Belhadj F. Terrain modeling: a constrained fractal model. In: Proceedings of the 5th international conference on computer graphics, virtual reality, visualisation and interaction in Africa, AFRIGRAPH '07. New York, NY, USA: ACM; 2007. p. 197–204. ISBN 978-1-59593-906-7.
[6] Merrell P. Example-based model synthesis. In: Proceedings of the 2007 symposium on interactive 3D graphics and games, I3D '07. New York, NY, USA: ACM; 2007. p. 105–12. ISBN 978-1-59593-628-8.
[7] Merrell P, Manocha D. Continuous model synthesis. In: Proceedings of the ACM SIGGRAPH Asia 2008 papers, SIGGRAPH Asia '08. New York, NY, USA: ACM; 2008. p. 158:1–158:7.
[8] Havemann S. Generative mesh modeling. TU Braunschweig; 2005.
[9] Havemann S, Fellner D. Generative parametric design of gothic window tracery. In: Proceedings of the 2004 shape modeling applications, 2004; 2004. p. 350–3.
[10] Parish H, Muller P. Procedural modeling of cities. In: Proceedings of the ACM SIGGRAPH 2001; 2001. p. 301–8.
[11] CityEngine. http://www.esri.com/software/cityengine.
[12] Gröger G, Plümer L. Citygml–interoperable semantic 3D city models. ISPRS J Photogramm Remote Sens 2012;71:12–33.
[13] Chen G, Esch G, Wonka P, Muller P, Zhang E. Interactive procedural street modeling. ACM Trans Graph 2008;27(3).
[14] Lipp M, Scherzer D, Wonka P, Wimmer M. Interactive modeling of city layouts using layers of procedural content. Comput Graph Forum (Proc EG 2011) 2011;30(2):345–54.
[15] Pottmann H, Schiftner A, Wallner J. Geometry of architectural freeform structures. Int Math Nachr 2008;209:15–28.
[16] Konecny R, Syllaiou S, Liarokapis F. Procedural modeling in archaeology: approximating ionic style columns for games. In: Proceedings of the 8th international conference on games and virtual worlds for serious applications (VS-GAMES). IEEE; 2016. p. 1–8.
[17] Smelik RM, Tutenel T, Bidarra R, Benes B. A survey on procedural modelling for virtual worlds. In: Proceedings of the computer graphics forum, 33. Wiley Online Library; 2014. p. 31–50.
[18] Kelly T, Wonka P. Interactive architectural modeling with procedural extrusions. ACM Trans Graph 2011;30(2).
[19] Musialski P, Wimmer M, Wonka P. Interactive coherence-based facade modeling. Comput Graph Forum 2012;31(2pt3):661–70.
[20] Merrell P, Schkufza E, Koltun V. Computer-generated residential building layouts. ACM Trans Graph 2010;29(6).
[21] Merrell P, Schkufza E, Li Z, Agrawala M, Koltun V. Interactive furniture layout using interior design guidelines. ACM Trans Graph 2011;30(4).
[22] Germer T, Schwarz M. Procedural arrangement of furniture for real-time walkthroughs. Comput Graph Forum 2009;28(8):2068–78.
[23] Yu L, Yeung SK, Tang C, Terzopoulos D, Chan TF, Osher S. Make it home: automatic optimization of furniture arrangement. ACM Trans Graph 2011;30(4):86.
[24] Kitsakis D, Tsiliakou E, Labropoulos T, Dimopoulou E. Procedural 3D modelling for traditional settlements. the case study of central Zagori. Int Arch Photogramm Remote Sens Spat Inf Sci 2017;42:369.
[25] Saldana M. An integrated approach to the procedural modeling of ancient cities and buildings. Digit Scholarsh Humanit 2015;30(suppl_1):i148–63.
[26] Dylla K, Frischer B, Müller P, Ulmer A, Haegler S. Rome reborn 2.0: a case study of virtual city reconstruction using procedural modeling techniques. Comput Graph World 2008;16(6):62–6.
[27] Whiting E, Ochsendorf J, Durand F. Procedural modeling of structurally-sound masonry buildings. ACM Trans Graph 2009;28(5):112.
[28] Cutler B, Dorsey J, McMillan L, Mller M, Jagnow R. A procedural approach to authoring solid models. In: Proceedings of the 29th annual conference on computer graphics and interactive techniques, SIGGRAPH '02. New York, NY, USA: ACM; 2002. p. 302–11. ISBN 1-58113-521-1.
[29] Jesus D, Coelho A, Sousa AA. Layered shape grammars for procedural modelling of buildings. Vis Comput 2016;32(6-8):933–43.
[30] Kutzner T, Chaturvedi K, Kolbe TH. Citygml 3.0: new functions open up new applications. PFG – J Photogram Remote Sens Geoinform Sci 2020:1–19.
[31] Müller P, Vereenooghe T, Wonka P, Paap I, Gool LV. Procedural 3D reconstruction of Puuc buildings in Xkipch. In: Proceedings of the eurographics symposium on virtual reality, archaeology and cultural heritage (VAST); 2006. p. 139–46.
[32] Koutsourakis P, Simon L, Teboul L, Tziritas G, Paragios N. Single view reconstruction using shape grammars for urban environments. In: Proceedings of the IEEE international conference on computer vision; 2009. p. 1–8.
[33] Dick AR, Torr PHS, Cipolla R. Modelling and interpretation of architecture from several images. Int J Comput Vis 2004;60(2):111–34.
[34] Kyriakaki G, Doulamis A, Doulamis N, Ioannides M, Makantasis K, Protopapadakis E, Hadjiprocopis A, Wenzel K, Fritsch D, Klein M, et al. 4d reconstruction of tangible cultural heritage objects from web-retrieved images. Int J Herit Digit Era 2014;3(2):431–51.
[35] Hohmann B, Krispel U, Havemann S, Fellner D. Cityfit: high-quality urban reconstructions by fitting shape grammers to images and derived textured point clouds. In: Proceedings of the ISPRS international workshop; 2009. p. 1–8.
[36] Zhao P, Fang T, Xiao J, Zhang H, Zhao Q, Quan L. Rectilinear parsing of architecture in urban environment. In: Proceedings of the IEEE conference on computer vision and pattern recognition; 2010. p. 1–8.
[37] Muller P, Zeng G, Wonka P, Gool LV. Image-based procedural modeling of facades. ACM Trans Graph 2007;26(3):1–9.
[38] Teboul O, Kokkinos I, Simon L, Koutsourakis P, Paragios N. Shape grammar parsing via reinforcement learning. In: Proceedings of the IEEE conference on computer vision and pattern recognition; 2011. p. 2273–80.
[39] Teboul O, Simon L, Koutsourakis P, Paragios N. Segmentation of building facades using procedural shape prior. In: Proceedings of the IEEE conference on computer vision and pattern recognition; 2010. p. 1–8.
[40] Jiang H, Yan D-M, Zhang X, Wonka P. Selection expressions for procedural modeling. IEEE Trans Vis Comput Graph 2018.
[41] Jones RK, Barton T, Xu X, Wang K, Jiang E, Guerrero P, Mitra NJ, Ritchie D. ShapeAssembly: learning to generate programs for 3D shape structure synthesis. ACM Trans Graph (TOG) 2020;39(6):1–20.
[42] Ritchie D, Mildenhall B, Goodman ND, Hanrahan P. Controlling procedural modeling programs with stochastically-ordered sequential Monte Carlo. ACM Trans Graph (TOG) 2015;34(4):1–11.
[43] Talton JO, Lou Y, Lesser S, Duke J, Měch R, Koltun V. Metropolis procedural modeling. ACM Trans Graph (TOG) 2011;30(2):1–14.
[44] Lienhard S, Lau C, Müller P, Wonka P, Pauly M. Design transformations for rule-based procedural modeling. In: Proceedings of the 2017 computer graphics forum, 36. Wiley Online Library; 2017. p. 39–48.
[45] Krecklau L, Pavic D, Kobbelt L. Generalized use of non-terminal symbols for procedural modeling. Comput Graph Forum 2010;29(8):2291–303.
[46] Edelsbrunner J, Havemann S, Sourin A, Fellner DW. Procedural modeling of architecture with round geometry. Comput Graph 2017;64:14–25.
[47] Leblanc L, Houle J, Poulin P. Component-based modeling of complete buildings. In: Proceedings of the graphics interface 2011; 2011. p. 87–94.
[48] Pavić D, Campen M, Kobbelt L. Hybrid Booleans. In: Proceedings of the computer graphics forum, 29. Wiley Online Library; 2010. p. 75–87.
[49] Zhou Q, Grinspun E, Zorin D, Jacobson A. Mesh arrangements for solid geometry. ACM Trans Graphics (TOG) 2016;35(4):1–15.
[50] Barki H, Guennebaud G, Foufou S. Exact, robust, and efficient regularized Booleans on general 3D meshes. Comput Math Appl 2015;70(6):1235–54.
[51] Sargeant T, Marie S, Martin P. Carve: A fast, robust constructive solid geometry library for boolean operations on polyhedra 2009. https://github.com/VTREEM/Carve.