Component-based Modeling of Complete Buildings

Luc Leblanc * Jocelyn Houle Pierre Poulin

LIGUM, Dept. I.R.O., Université de Montréal

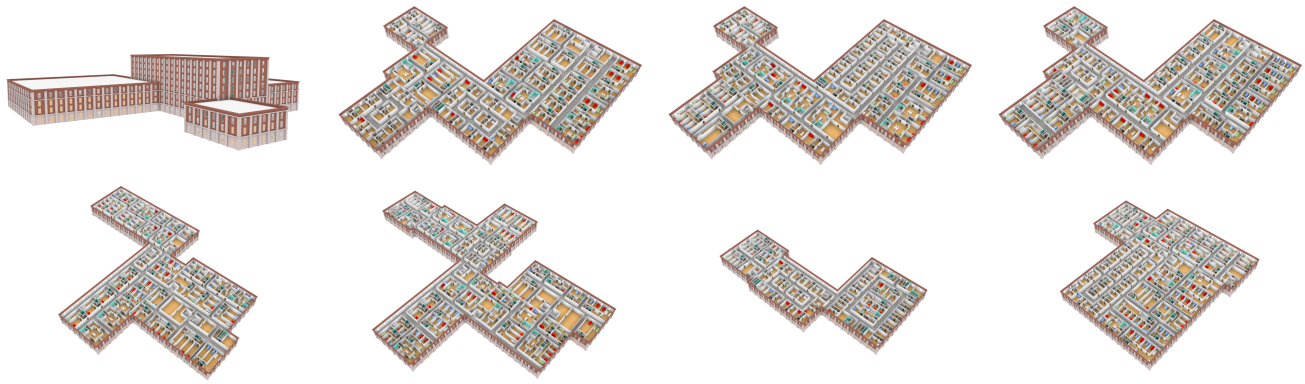

Figure 1: Variations on a building. Top: Random variations on the distribution of apartments, secondary corridors, rooms, and furniture for one randomly generated configuration of wings in a multi-storey building. Bottom: Random variations on the wing shapes and their content.

ABSTRACT

We present a system to procedurally generate complex models with interdependent elements. Our system relies on the concept of components to spatially and semantically define various elements. Through a series of successive statements executed on a subset of components selected with queries, we grow a tree of components ultimately defining a model.

We apply our concept and representation of components to the generation of complete buildings, with coherent interior and exterior. It proves general and well adapted to support subdivision of volumes, insertion of openings, embedding of staircases, decoration of façades and walls, layout of furniture, and various other operations required when constructing a complete building.

Keywords: Procedural Modeling, Architecture, Shape Grammar, Boolean Operation

Index Terms: Computer Graphics [I.3.5]: Computational Geometry and Object Modeling

1 INTRODUCTION

Buildings host a great deal of modern human activity. As such, every immersive computer graphics (CG) project, whether it be movie special effects, virtual reality systems, or video games, is bound to eventually require buildings. Our familiarity with buildings mandates a high degree of fidelity, and therefore, many adopted solutions rely mainly on manual labor from artists. Consequently, creating an entire building, or worse, all the buildings of a city, quickly becomes a daunting endeavor.

Procedural modeling is an excellent method to tackle the complexity of reality. Instead of relying on long and sustained human involvement, arbitrarily complex objects can be generated with little input from a user. This approach forgoes defining every little manual detail in favor of a succinct set of automatic rules able to satisfy most cases reasonably well. Various procedural techniques have been fairly popular in specialized modeling domains of CG, such as fractals for landscapes, L-systems for plants, particle systems for fluids, and shape grammars for building exteriors.

Shape grammars constitute the state-of-the-art in procedural modeling of building exteriors, and have produced high-quality results [4]. However, even though modeling building interiors and exteriors appears similar, shape grammars have not yet proven to be a good solution for modeling complete buildings. In fact, since their creation, only a small number of grammars, such as the palladian [30], have been produced for 2D floor plan generation, and better solutions have been provided by optimization techniques. Moreover, despite 10 years of development, shape grammars have seemingly yet to be used to model complete buildings.

This paper presents our solution to generate procedural buildings with coherent interiors and exteriors. We introduce a system capable of simulating split grammars and executing CSG (Constructive Solid Geometry) operations within a unified context. Our technique consists of executing a series of operations (i.e., a program) on a set of shapes selected by a query mechanism. These operations and queries are implemented as a programming language, and consequently, our system retains the flexibility and generality of programming languages, which is an asset in procedural modeling. The language is devoted to modeling with components, which is different than a library of tools on top of a regular programming language. Our system is currently not intended for general artists, but rather for designers with some programming skills. Moreover, our goal is to generate believable and coherent buildings for game and special effects environments, similar to those from recent CG shape grammars. While we hope to explore more advanced architectural issues in the future, we are not architects, and our system first addresses the basic needs for building design. It provides tools, but intelligence is still in the designer's hands. However, with careful design, the procedural modeling aspect in our system allows for

*e-mail: { leblanc, houlejo, poulin } @iro.umontreal.ca

inherent variability and reusability.

The main contribution in this paper is the system proper: the components and their attributes, the organisation of the components into a tree, the query mechanisms to select any subset of components, the operations applied to create new components, the constraints to respect, etc. Since this is a large system, some concepts may have appeared elsewhere, in one form or another. However, by putting them together, it creates a flexible system that has many advantages in the construction of complete buildings.

The paper is organized as follows. After a brief survey of key contributions in procedural modeling for buildings in Section 2, we define all of the concepts, elements, and operations of our technique in Section 3. We then apply it specifically to building creation and describe typical designing steps in Section 4. We present and analyze some of our results in Section 5, before concluding and discussing future extensions in Section 6.

2 Previous Work

Buildings are studied in a multitude of research domains: architecture, engineering, sociology, arts, etc. Each domain has its concerns and methodology. However, because our goal is to procedurally generate buildings for CG applications, we will restrict our study to this specific field.

Procedural modeling with grammars is an efficient method to generate scenes with rich geometric details. Generally speaking, its production rules modify a structure by adding details at each iteration. More than just automating the design process, it allows the exploration of design alternatives, and may even suggest possible innovative designs. A large number of design grammars have been introduced, notably, with a representative reference: L-systems [26], Chomsky grammars [28], graph grammars [6], shape grammars [29], and attributed grammars [13].

In architecture, Stiny [29] and his collaborators have introduced a number of different types of grammars to model and analyze buildings. While they laid the basis for what was to follow in CG, their grammars require much user involvement, which hinders the type of efficient generative modeling generally required for buildings in CG applications.

Nonetheless, a number of shape grammars have been dedicated to very specific architectural styles, but very few are accompanied by computer implementations. In an early exception to this remark, Heisserman [10] developed a boundary-solid grammar formalism and his implementation showed interesting variations on Queen Anne houses.

Parish and Müller [25] initiated much of the current trend on procedural generation of buildings for CG applications. While their buildings have simple shapes and their façades use basic sets of textures, they successfully created complete and believable cities comprised of various buildings distributed along generated road maps.

The next step, by Wonka et al. [33], significantly improved on Parish and Müller's façades by introducing split and control grammars. They created large databases of architectural elements for façades that can be combined to produce different decorations and styles. Larive and Gaildrat [15] derived a wall grammar for their façades, integrated with ground-level building plan contours extracted from photos to extrude general shapes of buildings. Krecklau et al. [14] later generalized the definition of non-terminal symbols in an object-oriented way, allowing easy addition of new operators such as free-form deformations.

Another set of improvements for façades came from photos of real buildings. Aliaga et al. [2] manually identified blocks on calibrated photos of a building, and reorganized these blocks to create different structures. Müller et al. [23] automatically segmented and interpreted such photos to recreate the associated geometry. Xiao et al. [34] showed how to extract a wider variety of façades from street-view video footage. Nan et al. [24] demonstrated a general interface, *smartboxes*, to improve on the reconstruction of patterns even in regions with lower-quality data.

Building shapes and façades were coherently integrated when Müller et al. [22] introduced *CGA Shape*, a shape grammar for architecture, to create complex building shapes of high quality with numerous façade details. They used the control grammar of Wonka et al. [33], but among several other contributions, developed context-sensitive shape rules (snap lines and occlusion queries) to improve the quality of façade elements in the presence of intersecting shapes of a building. Further improvements in the form of an interactive visual grammar editor have since been proposed by Lipp et al. [19]. This allows for a more intuitive modeling, and avoids a combinatorial explosion of rules by permitting local persistent changes.

While shape grammars can produce high-quality building exteriors, they do not always produce the best associated details and underlying polygonizations. The appearance of buildings can be improved by the cellular brick textures of Legakis et al. [18], which adapts itself on edges and corners of shapes, and by the generative mesh modeling of Havemann [9], better adapted for complex details and ornaments. Recently, Teoh [31] presented a technique specifically designed to generate traditional East Asian architectures.

Compared to research on building exteriors, less work has appeared in CG on the design of building interiors. Harada et al. [8] described a method for interactive context-sensitive floor plan manipulation. They did not work on actual 3D space, nor building shapes. This argument applies in fact to most floor plan layout techniques studied in architecture, engineering, and games (e.g., [12, 32]). Hahn et al. [7] created simple layouts for building interiors. They propose general generation steps to create stairwells, corridors, and rooms. However, their layouts are restricted to very simple axis-aligned grid subdivision structures. They do not handle room details, such as windows, decorations, etc., except for doors (portals), and simple texture mapping. Bradley [3] presented 2D interiors on more complex non-convex regions.

Other work addressed the reconstruction of a 3D model from provided 2D floor plans [11, 35]. Recently, Merrell et al. [21] described a stochastic optimization technique trained on real floor plan data using a Bayesian network. Although they generate complete two-storey town-house type buildings, they concentrate on automated 2D floor plan generation and no details are given as to how the 3D buildings were created.

To our knowledge, no work has yet presented a unified system to procedurally generate both interiors and exteriors with a high degree of control. Our system is therefore one of the first attempts to address this complex problem. The buildings produced as examples in this paper and in the associated website [16] illustrate the potential of our system.

3 Concepts

In our system, a procedural object (e.g., a building), is defined using a program composed of a series of statements. Each statement, executed in sequence, operates on a set of components to modify its attributes or to create new components.

A component represents a shape positioned in space with a number of attributes (system- and user-defined). We use queries to restrict the set of components affected by each statement. Queries retrieve components sharing a common set of attributes, and since components are never replaced, we have access to any previously created component (unlike grammars, which only work on the current active symbols). Again, contrary to a grammar, the operation contained in the statement is not restricted to a single component at a time, but can be applied to the whole set at once, possibly even disjoint components, opening the door to CSG operations and the likes. While creating new components, a tree is formed, linking source components to their children (see Figure 2).

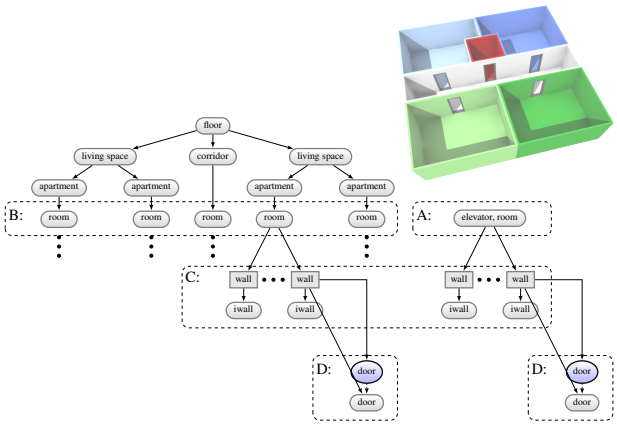

Figure 2: Component tree associated with the simple floor layout of four rooms, a corridor, and an elevator shaft (shown in the image). A curved box represents a 3D component, a rectangular box represents a 2D component, and a blue ellipse represents a region. Three dots means the same structure is repeated. Labeled dashed boxes (A-D) refer to modifications of the component tree after executing a portion of the code in Figure 3.

```
// Main component.
component( label="floor", size={10, 2.5, 10} )

// Creation of the apartments.
for c in query( "floor" ) do
   split( c, "Z", { label="living space", rel=1 },
                  { label="corridor"    , abs=2 },
                  { label="living space", rel=1 } )
end

for c in query( "living space" ) do
   split( c, "X", { label="apartment", rel=1 },
                  { label="apartment", rel=1 } )
end

// Creation of the elevator shaft (A).
component(
   label   ={"elevator", "room"},
   size    ={2, 2.5, 2},
   position={4, 0, 2}
)

// Creation of rooms cut by the elevator shaft (B).
for c in query( "apartment" or "corridor" ) do
   subtract( c, query( "elevator" ), { label="room" } )
end

// Extrusion of room walls, with a color attribute (C).
var i = 0
for c in query( "room" ) do
   i = i + 1
   for f in fquery( c, "SIDE" or "BOTTOM" ) do
      component( c, label="wall", boundary=f )
   end
   extrude(
      query( c, "wall" ),
      -0.05, { label="iwall", color=i }
   )
end

// Creation of doors by using regions (D).
for c in query( "wall" and not parent("corridor")
                       and occlusion("corridor") > 0 ) do
   region( c, label="door" )
end

for r in rquery( "door" ) do
   connect( componentFromFile( "door01" ), r )
end

// Creation of the actual geometry.
for c in query( "iwall" ) do
   solidGeometry( c, c.color )
end
```

Figure 3: Pseudocode to generate the simple example of Figure 2.

Not only can components be modified or created, they can also be connected together with the help of regions. Regions are shapes defining positional and orientational constraints on how two components can be connected. This can be seen as something akin to assembly modeling [27] sometimes used in CAD.

Our procedural modeling system unifies the equivalent of grammars, assembly modeling with connecting operations, generic modeling operations (CSG, extrusion, etc.), as well as some simple optimization techniques explained in Section 4.1. This combination of techniques allows choosing the best operations for different parts of the model. For example, we use split operations for façades, connections for decoration placements (doors, windows, furniture, lights, etc.), and CSG operations for interior space partitioning. Although not fully implemented yet, more sophisticated optimization operations could be devised within our system for more automated interior design and furniture placement.

3.1 Components

A component is a basic 2D or 3D shape arbitrarily positioned in space. Its shape is encoded as a *boundary* consisting of a polygon (2D) or a polyhedron (3D) contained in an oriented bounding box. The oriented bounding box defines a local space useful when applying certain operators.

A component can have an arbitrary number of child components; we refer to this hierarchy as the *component tree*. This tree can structure volumes hierarchically (e.g., a room in an apartment on a storey in a wing of a building). Multiple trees can exist simultaneously in the system.

A set of arbitrary attributes supplement the component's definition by allowing the user to specify various meta-information related to the component (e.g., floor level, wall thickness, molding style). Child components inherit their parent's attributes.

A list of labels describe the component's type. These labels are used in queries to select specific components we apply operations to. The labels are not inherited, since we typically want to work on a component, rather than its whole subtree. Nevertheless, a component has multiple, usually refining, labels (e.g., a room can also be a bedroom and a master bedroom).

The boundary and labels are determined at creation time, and they cannot be altered. The attributes can be altered at will.

3.1.1 Faces

A face is a polygon from a component's boundary and can be tagged with an identifier (a *face ID*). Unlike the component's labels, each face can have only a single ID, and those are not inherited, but rather carried to newly created faces. For example, when splitting a component in half with a split operator, two child components are created with a new boundary each. Each face of the new boundaries originating from the parent boundary, keeps the same ID as the original one. In Figure 10 (left), this mode of propagation enables, for instance, to position doors on the *front center* face, from a list of face IDs such as *front left*, *front center*, *front right*, *side*, and *back*.

Face IDs can also limit the scope of certain operations. A notable example is the boolean union, which will only merge faces sharing the same ID. Thus, adjacent co-planar sections can be kept distinct rather than being merged into a single polygon.

3.1.2 Regions

A region is a semantically meaningful shape (polygon or polyhedron) belonging to a component, positioned relative to it, and defining a range of valid positions (see Figure 4). A region also consists of a range of valid orientations, specified as minimum and maximum angles around each axis of a coordinate system.

A component is attached to another component's region by aligning its connector (coordinate system) to a valid position and orientation defined by the region. A component can contain multiple regions, but has a single connector. A region's area or volume defines (in non-degenerate cases) an infinite number of potential connector positions and orientations.

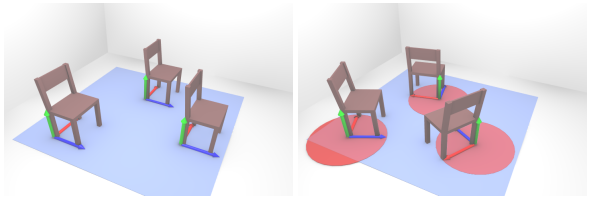

Figure 4: Left: A rectangular floor region (in light blue) with three chair components positioned on it; each connector is drawn as an axis system. This region, being smaller than its room component, restricts the placement of chairs. Right: Rotated variations of the original chair positions (valid orientations shown as red discs, i.e., indicated as 0° to 360° around the UP axis).

```
// Operations on every queried component.
for c in query( "label" ) do
   operation1( c, ... )
   operation2( c, ... )
   ...
   operationN( c, ... )
end

// Operation on all of the queried components at once.
operation( query( "label" ), ... )

// Nested queries.
for c in query( "label1" ) do
   operation( c, query( "label2" ), ... )
end
```

Figure 5: Examples of statement patterns.

3.2 Program

The construction of a building consists of executing a program, which applies a sequence of operations to a tree of components, thereby modifying the tree. Operations work on all or some of the components, selected with a query mechanism.

The basic element of a program, the statement, is generally composed of a query and one or multiple operations. Typical statement patterns are illustrated in Figure 5. For instance, a sequence of operations can be applied on each component returned by a query, or a single operation can be applied on the entire set of components returned by a query. In another example (nested queries), an operation can be applied on each component returned by a first query, and this operation is applied with a second set of components returned by a different query.

Since our program is written as a script, it benefits from the addition of conditional and generic loop statements. As an example, a probability of execution (if-statement) can determine if a sequence of operations (or an individual operation) is executed, thus creating families of buildings. This access to conditions and random generators (random values, random seeds, controled seeds) is at the heart of the procedural creation of variations at any level of the building structure. This concept is illustrated in Figure 1, as well as provided by some images and video sequences available on the associated website [16].

3.3 Queries

In order to limit the scope of the operations, a query mechanism parses through all of the components to select those respecting arbitrarily complex criteria. If source components are passed as arguments to a query, the parsing will be limited to these components and their children.

Three types of criteria are available for filtering: component attributes (e.g., labels, face ID, user-defined), component proximity, and component occlusion. A query with the proximity criteria finds all components within a certain distance of a given component. The occlusion factor between a component and a group of components is computed by projecting the boundary of the group on the component, and evaluating the fraction of the component area covered by this projection. In Figure 3, occlusion is used to identify a wall (2D component) adjacent to a corridor in order to place a door. Specialized queries allow retrieval of regions (for connection) as well as of boundary faces of components (e.g., to create 2D components to be used for extrusion).

3.4 Operations

The operations described in this section are but a few of all of the possible operations one could imagine or require. They have been used to produce the results presented in this paper.

3.4.1 Alteration

The simplest operation is to add, modify, or delete component attributes. Components can contain an arbitrary number of generic attributes.

3.4.2 Connection

Connecting a component to the region of another component consists of aligning the source component's connector (a coordinate system) onto the other component's region. The resulting rigid transformation is then propagated to the source component's children.

Since a region typically defines multiple possible coordinate systems, we allow the user to specify the exact position and orientation at the time of the connect. When no position or orientation is specified, a random one is generated within the range specified by the region. This is illustrated in Figure 4.

3.4.3 Creation

Since our technique relies on generating a hierarchy of components, there are a number of ways to generate them.

Instancing explicitly creates a component, either as a root component, or from a parent. Child components are defined in the parent's space, and by default, a child component has its boundary clipped by its parent's boundary, as to not extent outside its parent. This clipping is optional but very useful to create complex space partitioning in buildings. A face of a boundary can also be used to apply operations to it, but it first needs to be converted into 2D components.

Slicing generates multiple components by cutting a parent with a single specific size along one of the three main axes of its reference coordinate system. This operation is similar to the *repeat* operation from [33, 22], with the distinction that we have multiple policies to distribute the left-over space (give to first, give to last, split amongst both, or spread across everyone).

Splitting (also present in [33, 22]) creates child components along one of the three main axes using a list of sizes (relative, absolute, or both). Absolute sizes are first deducted from the parent's component dimension, and the remaining space is divided amongst all of the relative ones, weighted using their specified importance.

Boolean operations (union, intersection, and subtraction) can be applied to either 2D or 3D components. Union and intersection work on a set of components, and the result is stored as a root component to avoid conflicts in parent attributes. The subtraction cuts a set of components from a single other one, and stores the result as a child of the component undergoing the subtraction.

Extruding a 2D component along an arbitrary vector generates a 3D component. When applied to a set of 2D components, each edge shared by two adjacent 2D components is extruded in a single direction computed from these 2D components, thus generating two new 3D components which share a face, as illustrated in Figure 6. This is used extensively to create coherent non-overlapping walls and façades.

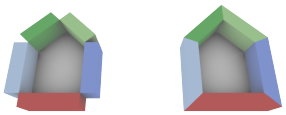

Figure 6: Comparison of extrusion modes. Left: Sides are extruded separately. Right: Sides are extruded together.

Roofing is a specialized extrusion to create simple roofs. The base polygon of the 2D component is extended towards a degenerate version of that same polygon in such a way as to yield gables.

The creation of regions, stored in components, as well as constraints, stored in user variables (see Section 3.5), complete our current list of operations.

3.5 Constraints

Constraints are spatial entities (polyhedra, polygons, or planes) that restrict or enforce the location or dimension of components. While any operation could be applied with a requirement to satisfy a list of such constraints, at this moment, they are mainly used with *split* and *slice* operations.

Constraints can serve to avoid having perpendicular walls crossing windows (see Figure 7), to align corridors or floors across wings of a building, or any other case where structural elements may affect neighboring components.

The user is solely responsible for determining the priority of the elements, and must therefore decide whether, for example, the interior walls will affect the façade layout, or whether it is the other way around. We have not devised an optimizing solution, as the scope of such work extends beyond this current research. We have instead opted to use a simpler solution, similar to what Müller et al. [22] have introduced.

Constraints affect the boundary of each component created with the *split* and *slice* operations. With the use of repulsion or attraction, the component's sides can avoid or snap to the constraints. This is achieved by reducing the problem into 1D along the axis of the operation. A specialized version of the *slice* operation can directly match a given list of constraints and use them as cutting planes.

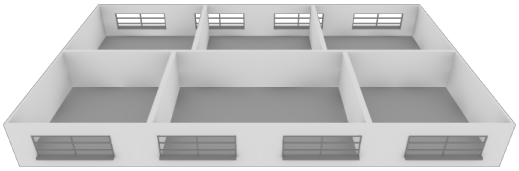

Figure 7: An example of creating and enforcing constraints, with the results without (far rooms) and with (near rooms) their application.

3.6 Geometry

The component hierarchy created by our technique is purely abstract, and only represents the spatial partitioning of the building. It serves as containers for real geometry that can be attached to it. Any component node can carry geometry, not only the leaf components of the hierarchy.

The geometry can come from a conversion of the component's boundary (this is the *solidGeometry()* call of Figure 3), or from any other external source (such as commercial modeling softwares). When using the *solidGeometry* operation, a surface parameterization is automatically generated for the geometry to enable the placement of a texture (a texture map or procedural texture).

The geometry used to define the building's structure (i.e., floors, walls, ceilings, doors, and windows) requires support of solid modeling, since doors and windows cut holes in walls (see Section 4.4 for details). It is important to note that this use of solid modeling is not the same as the CSG operators described previously in Section 3.4.3, since in this case, we are working on geometry as opposed to components. In contrast, the geometry used for decoration (furniture, appliances, fixtures, etc.) is not involved in CSG operations, and can therefore take advantage of instancing. We use our own system to model all our geometry; it is based on a block primitive [17].

4 Building Creation

Building creation can be divided in four steps:

(1) *Space partitioning* divides the building in sections, mainly hallways, stairwells, rooms, and closets.

(2) *Base geometry creation* adds walls, floors, and roofs.

(3) *Architecture elements*, such as doors, windows, and staircases, are applied to the base geometry, generally with connect operators.

(4) *Decoration* adds the final touch with placement of furnitures, light fixtures, paintings, etc.

The first three steps may alternate in parts. For instance when a window should affect wall placement, steps for space partitioning and base geometry creation would also occur after the application of the related architecture elements.

4.1 Space Partitioning

In order to obtain the 3D components required for the base geometry, we successively partition the volume of the building using various operations. In general, we proceed as follows.

We first represent the building using a group of 3D components accounting for the general appearance of the building's boundary. For each component requiring a roof, its top face is converted in a 2D component, which becomes the polygonal base used by the roofing operation. With a boolean merge operation, we combine the various parts of this base shape into a single component. We then slice the whole building into multiple storeys, assigning automatically a level number (an attribute) to every created component.

For every component of a storey, we segment it into corridors and apartments, using a combination of operations such as split, slice, and explicit component creation. The level number is automatically inherited as attribute to all newly created components. Similarly to levels, apartment numbers can be assigned.

Apartments are then subdivided into rooms, again with a combination of split, slice, and explicit component creation. Room categorization (living room, kitchen, bedroom, bathroom, etc.) is done using the component's label.

Some components created after the storeys generation may span across multiple storeys (stairwells, elevator shafts, auditoriums) and overlap other components. Since the base geometry requires non-overlapping space partitioning, intersecting volumes must be assigned to a single component. This is done by judiciously cutting volumes out of components, either explicitly, or through an automatic system. In the automatic case, the user specifies a priority (an attribute) in the components, and each overlapping component will get cut by any higher priority component. Since a component's boundary is fixed at creation, we store the results of the boolean subtractions using child components. This is illustrated in Figure 2, where component "elevator" (a label) is overlapping components "apartment". A subtraction operation creates the new non-overlapping components "room".

Stairwells, being shafts across multiple floors, require storey separations in order to create flights of stairs properly aligned with floors. Although we could mimic the slice operations previously used to create the storeys, we can also use constraints. By querying the components neighboring a stairwell, we can extract their floor 2D components, and use them to create a list of planar constraints. We then slice the stairwell with that list and recover our exact storey separations.

Constraints are also handy for other situations. To avoid having perpendicular walls crossing a window, we specify repulsion constraints on window areas we first collected using the neighborhood query. These constraints will then guide the placement of the walls being generated by splitting and slicing. Figure 7 gives an illustration of the results. On the other hand, snapping constraints can be used to align corridors across adjacent wings of a building.

While this way of creating space partitioning is flexible and powerful, it can also be difficult to produce the necessary set of statements. This is especially true when trying to create a building sporting multiple distinct variations. A simpler way would be to add a new operator capable of partitioning a component with a 2D floor plan optimization technique, such as the one of Merrell et al. [21]. Such an operator would fit nicely in our system, providing precise control where needed (using other more manual operators), and letting the system optimize in other cases. It would also be useful to propose alternatives or variations in spatial designs.

4.2 Base Geometry

We refer to the *base geometry* of a building as the geometry associated to the spatial division of the building. It is comprised of all the walls, floors, and ceilings of every room, corridor, and stairwell, as well as façade elements. It excludes openings, such as doors and windows, as well as decorations, such as furniture and moldings.

We form the base geometry by first extracting a subset of 3D components from the tree of components. We then extrude together the faces (2D components) of those components to form the walls, floors, and ceilings. This can be seen in Figure 2 where components with label "room" are chosen to create 2D components "wall" that are extruded towards the inside "iwall". These "iwall" 3D components are converted to solid geometry at the end of the program.

Interior components (rooms, corridors, stairwells, etc.) are extruded towards the inside of the room, while exterior components (façade elements) are extruded towards the outside of the building. Figure 8 illustrates the interior and exterior extrusions of a simple building.

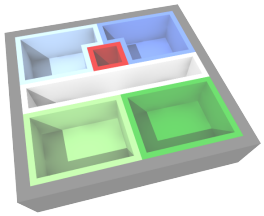

Figure 8: The resulting base geometry of Figure 2 extended with a simple façade (in dark gray). For each room, walls are extruded towards the interior; façade walls are extruded towards the exterior.

4.3 Façades

Façades are made from the exterior walls (2D components) of the building. They are created using a series of split and slice operations in a manner very similar to previous façade-generating techniques, such as [33]. Extrusions of the 2D components create ledges, bricks, etc., while regions are defined for windows, balconies, and doors (see Section 4.4).

We obtain the exterior walls by merging all of the wing components making the building. These merged components form the outer shell of the building. We can merge the wings either before or after they are sliced into storeys, depending on whether or not we want façade elements to match floor locations. When floor location is respected, the level attribute can be used to control the variations (see Figures 9 to 11). The face IDs (originally defined in the wings) can also be used to vary the façades, e.g., by identifying the main entrance at the front of the center wing.

4.4 Architectural Elements

Up to this point, all of the generated geometry is hermetically closed. To access rooms, we need a way to cut geometry out of them. This is done by performing boolean operations (subtracts in particular) on the component's geometry (not its boundary).

Doors and windows are therefore created with a combination of boolean operations that will first cut a hole in the wall prior to adding the frame, then add the door or window pane. This composite boolean operation (a subtract followed by a union) is stored with the component's geometry, and its placement is controlled through the connection mechanism within regions. Placing doors next to corridors is done by leveraging the occlusion factors to find the proper room sides.

Other architectural elements, such as balconies (see Figure 11 (top)) and light fixtures, also rely on connections for their placement, but do not require cutting holes.

Again, variation is easily possible through an interpretation of attributes or face IDs (e.g., vary the door type being connected if it leads to a corridor, another room, or a closet).

4.5 Furnishing Elements

Furniture is positioned using regions placed on the floor plan of a room. Some elements, such as shelves, cabinets, and appliances, typically adjoin walls, while some other elements, such as tables, carpets, couches, usually tend to be more centered in a room. We define and use floor regions accordingly.

Light fixtures are placed on the ceiling of rooms, not too close to walls, while floor lamps are placed near other furniture. Care must be taken for wall-mounted elements (picture frames, shelves, etc.) to not overlap windows and doors, the latter including their movement range. For that, we use a combination of constraints with neighboring queries and occlusion factors.

Although we have procedurally generated all elements in our system, the user is free to attach any external geometry to any component. For instance, models from commercial softwares can be connected anywhere in our component forest.

It should be noted that we have not tried, nor is it the goal of this research, to optimize placement, even if it should offer nicer results. We have simply used *ad hoc* placement policies, and used randomization to vary the results.

5 Results

Figures 1 and 9 to 11 show various building models created with our system. In order to give a little more insights on how the system was used, the actual code used to generate a number of buildings is provided in the website associated to this paper [16].

In the accompanying video, fly-over and walk-through sequences in some of these buildings give a more immersive feeling of the divisions of spaces, and of the decorations attached to some structural elements. A sequence of more gradual variations on a number of parameters shows how our system produces floor subdivisions adapted to the design and the constraints.

Another sequence of variations on much larger buildings show how random generators can be used to procedurally create families of buildings. One generator is assigned to the wing shapes, another one to the distribution of apartments. In a cutaway view of the

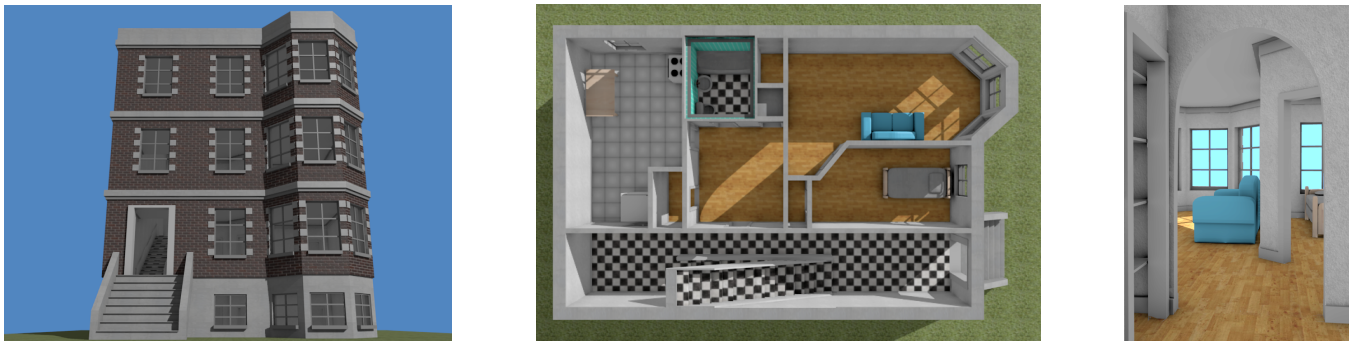

Figure 9: Row house, view of the first floor, and view from inside.

Figure 10: A small hotel, loft-style office building, and block-style apartment complex.

second storey (of multiple storeys), the sequence shows corridors, apartments, rooms, and furniture for variations on connected wing shapes. For one building shape, the sequence shows variations only on the distribution of apartments and secondary corridors. Figure 1 shows a subset of these images. Note that the design of the apartments and furniture have been simplified, the primary intention being to illustrate the variations.

Running the program to generate the component tree for each building took less than 5 seconds on an Intel Core 2 Duo laptop running at 2.2 GHz on a single thread. Creating its geometry (i.e., executing boolean operations for doors, windows, and all the other geometries) has been taking up to 2 minutes for the largest models (e.g., in Figure 1), and produced between 20K to 500K triangles.

The following table gives approximate numbers of triangles and operations executed to produce their respective buildings (see Figures 9 to 11).

building	row house	house	block apt	hotel	loft
triangles	90K	36K	160K	33K	78K
operations	80	67	22	25	58

Creating from scratch all of the operations for an entire building can be time consuming, taking a few hours. Nevertheless, since most buildings share a lot of common elements, we have evolved our operations into reusable libraries (e.g., corridor configurations, staircase patterns, façades and apartment layouts). As such, creating buildings based on previous templates becomes increasingly faster.

6 Conclusion

We have presented a technique to procedurally generate complex buildings with coherent exteriors and interiors. In our procedural building program, the sequence of operations creates a building by modifying a hierarchy of components, each describing various 3D and 2D elements of the building. We feel this paradigm, which more closely matches the traditional approach of programming, is powerful enough to handle the complex interrelations present in complete buildings. In fact, we do not see any limitation on the types of buildings or building elements that can be generated with our technique. Even though highly curved surfaces could prove a challenge in our current implementation, it is only a technical issue for efficiency.

While we have shown in this paper results of mainly static final buildings, the procedural nature of the process and the implementation as a programming language allow the creation of variations at most stages of our design through the use of random generators and conditional statements. Some of these variations are shown in the accompanying video sequences [16].

Although our design scheme is intended for users with programming skills, and as such it breaks away from the current trend of catering to the needs of non-experts, we believe that game and special effect industries have access to such programmers, and that a system similar to ours should prove appropriate.

In our opinion, a number of key concepts have been exploited to achieve flexibility in our technique:

- The query mechanism selects components sharing an identical pattern of labels or attributes. The same operations are then applied on the arbitrary number of selected components.
- The inheritance of component attributes offers a flexible scheme to semantically augment components.
- Placing regions on components allows us to delay some design (e.g., the connection of components) to a more appropriate stage of the modeling.
- Constraints permit to adjust the location of boundaries when creating new components, such as to avoid walls perpendicular to windows, or to align corridors.

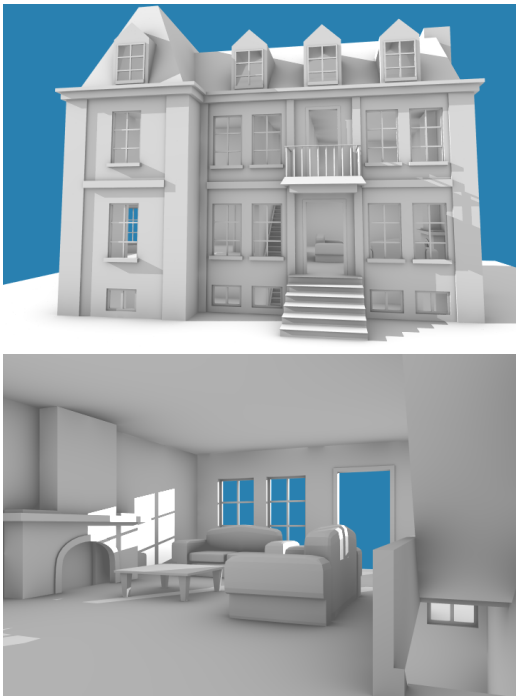

Figure 11: House and view of its interior.

For future work, we would like to exploit other types of semantics in our data to improve rendering efficiency, or even to achieve on-the-fly generation of visible building sections. Another promising but challenging research direction is to develop an intuitive graphical user interface for our system, thus making it more amenable to non-programmers.

Acknowledgements

The authors would like to thank Neil Stewart, Victor Ostromoukhov, Philippe Beaudoin, George Drettakis, Eric Blanchard, Anthony Pajot, and all the LIGUM members for their help. The anonymous reviewers made several constructive comments over the years. Financial support was generously provided by FQRNT, NSERC, and GRAND.

References

[1] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Town, Buildings, Construction*. Oxford University Press, 1977.

[2] D. G. Aliaga, P. A. Rosen, and D. R. Bekins. Style grammars for interactive visualization of architecture. *IEEE Trans. on Visualization and Computer Graphics*, 13(4):786–797, 2007.

[3] B. Bradley. Towards the procedural generation of urban building interiors. M.Sc. thesis, Game Programming, University of Hull, 2005.

[4] CityEngine. www.procedural.com/cityengine, Dec. 2010.

[5] P. E. Debevec, C. J. Taylor, and J. Malik. Modeling and rendering architecture from photographs: a hybrid geometry- and image-based approach. In *SIGGRAPH '96*, pages 11–20, 1996.

[6] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg. *Handbook of graph grammars and computing by graph transformation: applications, languages, and tools*, volume 2. World Scientific Publishing, 1999.

[7] E. Hahn, P. Bose, and A. Whitehead. Persistent realtime building interior generation. In *SANDBOX '06: Proc. ACM SIGGRAPH symposium on Videogames*, pages 179–186, 2006.

[8] M. Harada, A. Witkin, and D. Baraff. Interactive physically-based manipulation of discrete/continuous models. In *SIGGRAPH '95*, pages 199–208, 1995.

[9] S. Havemann. *Generative Mesh Modeling*. PhD thesis, TU, Braunschweig, 2005.

[10] J. Heisserman. Generative geometric design. *IEEE Computer Graphics and Applications*, 14(2):37–45, 1994.

[11] S. Horna, G. Damiand, D. Meneveaux, and Y. Bertrand. Building 3D indoor scenes topology from 2D architectural plans. In *Conference on Computer Graphics Theory and Applications*, number 1, 2007.

[12] J. Jo and J. Gero. Space layout planning using an evolutionary approach. *Artificial Intelligence in Engineering*, 12:149–162, 1998.

[13] D. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[14] L. Krecklau, D. Pavic, and L. Kobbelt. Generalized use of non-terminal symbols for procedural modeling. *Computer Graphics Forum*, 29(8):2291—2303, 2010.

[15] M. Larive and V. Gaildrat. Wall grammar for building generation. In *GRAPHITE '06*, pages 429–437, 2006.

[16] L. Leblanc. www.iro.umontreal.ca/labs/infographie/papers/Leblanc-2011-CMCB, Mar. 2011.

[17] L. Leblanc, J. Houle, and P. Poulin. Modeling with blocks. In *Proc. Computer Graphics International '11*, June 2011.

[18] J. Legakis, J. Dorsey, and S. Gortler. Feature-based cellular texturing for architectural models. In *SIGGRAPH '01*, pages 309–316, 2001.

[19] M. Lipp, P. Wonka, and M. Wimmer. Interactive visual editing of grammars for procedural architecture. In *SIGGRAPH '08*, pages 102:1–10, 2008.

[20] J.-E. Marvie, J. Perret, and K. Bouatouch. The FL-system: a functional L-system for procedural geometric modeling. *The Visual Computer*, 21(5):329–339, June 2005.

[21] P. Merrell, E. Schkufza, and V. Koltun. Computer-generated residential building layouts. In *SIGGRAPH Asia '10*, pages 181:1–12, 2010.

[22] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. V. Gool. Procedural modeling of buildings. In *SIGGRAPH '06*, pages 614–623, 2006.

[23] P. Müller, G. Zeng, P. Wonka, and L. V. Gool. Image-based procedural modeling of facades. In *SIGGRAPH '07*, pages 85:1–9, 2007.

[24] L. Nan, A. Sharf, H. Zhang, D. Cohen-Or, and B. Chen. Smartboxes for interactive urban reconstruction. In *SIGGRAPH '10*, pages 93:1–10, 2010.

[25] Y. I. H. Parish and P. Müller. Procedural modeling of cities. In *SIGGRAPH '01*, pages 301–308, 2001.

[26] P. Prusinkiewicz and A. Lindenmayer. *The algorithmic beauty of plants*. Springer-Verlag, 1990.

[27] J. J. Shah and M. T. Rogers. Assembly modeling as an extension of feature-based design. *Research in Engineering Design*, 5(3):218–237, 1993.

[28] M. Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1996.

[29] G. Stiny. Introduction to shape and shape grammars. *Environment and Planning B: Planning and Design*, 7:343–351, 1980.

[30] G. Stiny and W. Mitchell. The palladian grammar. *Environment and Planning B: Planning and Design*, 5(1):5–18, 1978.

[31] S. T. Teoh. Generalized descriptions for the procedural modeling of ancient east asian buildings. In *International Symposium on Computational Aesthetics in Graphics, Visualization, and Imaging*, May 2009.

[32] T. Tutenel, R. Bidarra, R. M. Smelik, and K. J. de Kraker. Rule-based layout solving and its application to procedural interior generation. In *3AMIGAS: Proc. CASA workshop on 3D Advanced Media in Gaming and Simulation*, pages 15–24, 2009.

[33] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky. Instant architecture. In *SIGGRAPH '03*, pages 669–677, 2003.

[34] J. Xiao, T. Fang, P. Tan, P. Zhao, E. Ofek, and L. Quan. Image-based façade modeling. In *SIGGRAPH Asia '08*, pages 161:1–10, 2008.

[35] X. Yin, P. Wonka, and A. Razdan. Generating 3D building models from architectural drawings : A survey. *IEEE Computer Graphics and Applications*, 29(1):20–30, 2009.

[36] L. Yong, X. Congfu, P. Zhigeng, and P. Yunhe. Semantic modeling project: building vernacular house of southeast china. In *VRCAI '04: Proc. ACM SIGGRAPH international conference on Virtual Reality continuum and its applications in industry*, pages 412–418, 2004.