Render-Time Procedural Per-Pixel Geometry Generation

Jean-Eudes Marvie* Pascal Gautron† Patrice Hirtzlin‡ Gaël Sourimant§

Technicolor Research and Innovation

(a) Procedural city, ∼ 10000 facades, 7fps (b) Macroscopic details (c) Microscopic details (d) On custom mesh

Figure 1: Per-pixel generation of procedural facade details: we propose a compact and generic solution for generating and rendering procedural building models featuring both macroscopic (b) and microscopic (c) details. Our method can be used to generate and render massive cities (a), as well as custom-shaped buildings (d).

ABSTRACT

We introduce *procedural geometry mapping* and *ray-dependent grammar development* for fast and scalable render-time generation of procedural geometric details on graphics hardware. By leveraging the properties of the widely used split grammars, we replace geometry generation by lazy per-pixel grammar development. This approach drastically reduces the memory costs while implicitly concentrating the computations on objects spanning large areas in image space. Starting with a building footprint, the bounding volume of each facade is projected towards the viewer. For each pixel we lazily develop the grammar describing the facade and intersect the potentially visible split rules and terminal shapes. Further geometric details are added using normal and relief mapping in terminal space. Our approach also supports the computation of per-pixel self shadowing on facades for high visual quality. We demonstrate interactive performance even when generating and tuning large cityscapes comprising thousands of facades. The method is generalized to arbitrary mesh-based shapes to provide full artistic control over the generation of the procedural elements, making it also usable outside the context of urban modeling.

Index Terms: F.4.2 [[Mathematical Logic and Formal Languages]: Grammars and Other Rewriting Systems; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism; I.6.3 [Simulation and Modeling]: Applications; J.6 [Computer-Aided Engineering]: Computer-Aided Design (CAD)

1 INTRODUCTION

Virtual worlds become more and more complex in terms of both geometry complexity and shading. However, when it comes to the generation and rendering of complex urban environments, the current challenges are numerous. In particular, the modeling and rendering of varieties of buildings is often a tedious task. Many methods have been proposed to overcome this problem by introducing procedural modeling, in which the facade of a building is described by a small set of construction rules, or grammar, and parameters. However, the output of such methods is usually a very complex mesh, resulting in significant computational costs as well as a high memory footprint. Also, any adjustment of the parameters during the design process involves regenerating the entire geometry.

We propose a method for real-time generation of procedural buildings avoiding the need for actual geometry storage. Our solution is based on a lazy development of facade grammars on graphics hardware at render time. This allows fast editing of the parameters of procedural models and provides interactive performance even on city models comprising thousands of buildings. As our approach only requires the footprint of the buildings plus few additional grammar parameters, the memory costs of our method are negligible compared to explicit geometry storage. Our contributions are twofold: first, *ray-dependent grammar development* efficiently renders buildings described by grammars on graphics hardware. Second, *procedural geometry mapping* associates the concepts of texture mapping and procedural modeling to generate facade details on footprint extrusions. We generalize this method to arbitrary bounding meshes for increased artistic control on the shapes of the buildings. The absence of actual geometry generation allows artists to visualize and modify numerous buildings simultaneously at interactive rates, thus improving the design workflow for large cityscapes.

2 RELATED WORK

The literature on procedural modeling and geometry generation features numerous documents. In this section, we focus on existing work closely related to our method, in the fields of both per-pixel geometry synthesis using graphics hardware and grammar-based generation of buildings.

2.1 Per-Pixel Geometry

The topic of geometry synthesis on graphics hardware has driven many recent studies among which we consider two categories: geometry images and displacement mapping.

Geometry images [6, 10, 17, 2, 4, 5] are generated in a preprocess to encode the shape of a complex object within a set of texture maps. At render time, a ray tracing is then performed for each fragment of a bounding volume by intersecting the complex mesh stored in the textures. This technique proves effective for rendering very complex models in which ray tracing is more efficient than rasterization. However, the generation of the geometry images is computationally expensive, hence preventing interactive generation and modification.

*e-mail:jean-eudes.marvie@technicolor.com

†e-mail:pascal.gautron@technicolor.com

‡e-mail:patrice.hirtzlin@technicolor.com

§e-mail:gael.sourimant@technicolor.com

Displacement mapping techniques are mainly used to refine coarse meshes by adding fine-scale details [18]. Originally performed per-vertex [3, 8], per-pixel approaches [16, 15] have also been introduced to avoid the need for subdividing the meshes. Furthermore, such per-pixel techniques implicitly concentrate the GPU load on objects covering large parts of the image, which tend to be visually important. However, the advantages of this feature are now counterbalanced by the geometry and tesselation shaders available on the latest graphics hardware: Geometry shaders alone introduce geometry generation at the expense of performance. Tesselation shaders overcome this problem but require complex parameterization to generate specific geometries.

2.2 Procedural Modeling

The generation of architectural models described by grammars has been devised in several forms, including L-Systems [14], FL-Systems [12] and Split grammars [19]. While those methods provide high quality results, their main drawback is the lack of intuitive design tools. This problem has been overcome with Computer Generated Architecture Shape grammars [13], for which visual editing tools have been proposed [9].

Despite some conceptual differences, the above methods exhibit common characteristics: the generated models are potentially large datasets encoding all the details of the buildings, which are then rendered using classical rendering methods. To avoid a geometrical bottleneck when rendering large cityscapes, some techniques combine visibility culling and levels of detail with on-the-fly grammar development to reduce the polygon count and maintain interactive frame rates [11]. However, another common aspect is the unidirectional generation process: once the set of polygons describing the surface of the building is generated, any further modification of the parameters requires a complete regeneration.

The Compressed Facade Displacement Maps [1, 7] aim at rendering complex cityscapes at interactive frame rates. The details of the facades are encoded within displacement maps, which are then rendered using per-pixel displacement mapping. While related to our approach, this technique suffers from the same drawbacks as the geometry images: the geometry representation is synthesized during a potentially expensive preprocessing step, even on graphics hardware [7]. Again, any subsequent modification of the facade parameters requires regenerating the displacement maps.

Besides the need for precomputations, another common denominator of those approaches is a high memory consumption: the entire geometry of the generated buildings has to be cached in main or graphics memory prior to rendering. Even though this storage is not problematic when rendering a single building, this may become an issue for large cityscapes. Based on these observations, our method preserves the procedural representation until fragment shading so that no actual geometry gets generated nor stored. The facade details are rendered by developing facade grammars on-the-fly, directly on graphics hardware.

3 Facade Grammar

This section provides the key aspects of procedural modeling of realistic facades. A facade is generally composed of repeated macroscopic elements such as windows or doors (Figure 1b). The arrangement and structure of such elements can be efficiently described by a set of rules, or grammar, along with associated parameters.

Our method is exposed throughout the paper using the following split grammar [19] to generate the facade details illustrated in Figure 1, using Müller *et al.*'s notation [13]:

1. $F \rightarrow \text{Split}(\text{"}y\text{"}, floorHeight, \sim floorHeight)\{GF, FL^*\}$
2. $GF \rightarrow \text{Split}(\text{"}y\text{"}, 0.5, floorHeight - 0.5)\{L_f, G\}$
3. $G \rightarrow \text{Split}(\text{"}x\text{"}, doorWidth, \sim windowWidth)\{D_f, W_f^*\}$
4. $FL \rightarrow \text{Split}(\text{"}x\text{"}, \sim windowWidth)\{W_f^*\}$

where F, GF, G and FL respectively represent the facade, ground floor including top ledge, ground floor elements and the other floors. L_f, D_f and W_f are the functions describing the first floor ledge, the door and the windows. This example grammar is kept simple for the clarity of the exposition: the application to more complex split grammars is discussed in Section 8.

4 Overview

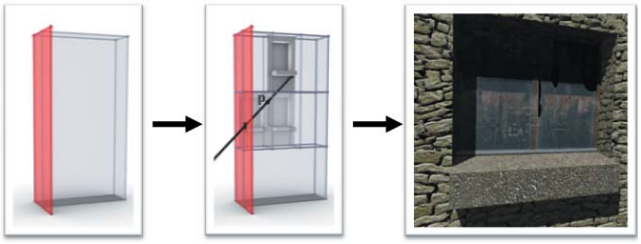


Figure 2: Main steps of our render-time modeling method: from left to right, extraction of the facade coverage, ray-dependent grammar development and ray-terminal intersection, relief mapping and self shadowing in terminal space.

Our method for render-time procedural modeling comprises four key aspects. Three of those are depicted in Figure 4. Starting with the footprint of a facade, we first generate a polygon covering the projection of the facade extents onto the screen (Section 5). This facade coverage polygon is a virtual entry gate towards the procedural structure of the facade.

Then, for each fragment of this polygon, we perform a lazy development of the facade grammar to determine the potentially visible shapes (Section 6.1). The facade point visible through each fragment is determined in facade space by testing the view ray against the rules and terminal symbols of the grammar.

The view ray steps through those terminal symbols by computing ray-shape intersections: we devise the use of parametric surface sets and geometry images for efficient shape representation on graphics hardware (Section 6.2). Further details are achieved using normal and relief mapping in terminal space to represent the smallest features of the surfaces (Section 6.3). This lazy development is extended for rendering fine self shadowing details on procedural elements (Section 6.4). Our method provides continuous LOD management (Section 7) for interactive performance and negligible memory cost even when rendering and mass-editing large cityscapes (Section 9).

Our approach is very efficient on buildings described by footprints, which is particularly useful for automatic interactive generation of large urban environments. For more artistic control, we generalize our method for intuitive interactive generation and tuning of facade details on arbitrary meshes based on the concept of texture mapping (Section 10). Specific results and issues regarding this generalization are discussed in Section 10.3.

5 Extraction of the Facade Coverage

Starting with an extrusion of the building footprint, our method generates procedural details which may stick out, such as window sills. This section addresses the alteration of the polygons describing each facade to ensure a proper coverage of the protruding elements in screen space. This process is performed in *Facade Space*, in two main steps: the computation of the bounding volume of the facade and the projection of a viewpoint-facing coverage polygon.

(a) (b) (c) (d) (e)

Figure 3: For a given facade of an extruded building footprint, we define its *facade space* whose axes are aligned with the facade (a). Then, using the global parameters of the grammar, we compute the bounding volume of the facade (b). This volume is projected onto the axis-aligned plane in facade space most facing the viewpoint. The vertices of the facade are displaced to encompass the projected volume (c). For another facade we choose another projection plane to render potentially visible protrusions depending on the relative locations of the viewpoint and the facade (d). Each facade is processed independently, yielding a set of overlapping coverage polygons (e). This overlap allows our method to solve visibility issues when a view ray traverses the bounding volumes of several facades.

5.1 Facade Space

Our technique is based on the computation of intersections between the viewing rays and the rules describing the facade. In a way similar to relief mapping approaches, we perform such computations in the tangent space of the surface. In the context of buildings generated from a footprint, the tangent space ($\mathbf{t_F}$, $\mathbf{b_F}$, $\mathbf{n_F}$) is constant over the facade, and is described by the facade normal $\mathbf{n}$ and a vertical binormal $\mathbf{b}$ (Figure 3a). By convention, we define the origin of the facade space at the bottom left of the facade.

5.2 Computation of the Facade Volume

The bounding volume of the facade is directly extracted from the building footprint and the parameters of the grammar: while the footprint provides the facade width, the parameters of the grammar include the height of the building as well as the overall thickness of the facade including the procedural elements. As those elements may protrude or recess from the facade, the thickness is the sum of the maximum protrusion and recess distances yielding the bounding volume illustrated in Figure 3b. Based on this volume definition, our algorithm determines a coverage of the volume on the screen.

5.3 Facade Coverage Polygon

Using the above definition of the facade volume, we focus on generating a simple geometry covering the volume. This geometry then undergoes a rasterization process, generating fragments over the facade along with information regarding the location on the facade. A first approach consists in extruding the facade to build a bounding box of the facade. While effective in the context of footprint-based generation, this approach quickly becomes intractable when generating facades on arbitrary meshes: such boxes would become truncated pyramids whose coherency along the surface of an animated mesh cannot be easily ensured. Instead, we propose a simple transformation of the vertices of the facade to create a polygon covering the entire projected facade: we first select the orientation of the polygon according to relative locations of the facade and the viewpoint, then compute the actual location of its vertices.

5.3.1 Orientation Selection

The orientation of the coverage polygon ensures a proper generation of the fragments covered by the facade: based on the location of the viewpoint, our algorithm chooses the best axis-aligned orientation in facade space. Depending on the location of the viewpoint with respect to the facade, we align the polygon so that it is perpendicular to the major axis of the viewing direction in facade space (Figures 3c and 3d).

5.3.2 Facade Projection

Once the orientation of the coverage polygon has been determined, we adjust the location of the vertices so that the entire facade volume is covered in screen space. As shown in Figure 3c, we project the bounding volume of the facade onto the axis-aligned plane in facade space most facing the viewpoint. This projection is not only performed in world space, but also in facade space to retain the location of the points with respect to the facade. The 4 vertices of the coverage polygon are then displaced to fit the projection of the bounding volume. As illustrated by Figure 3e, the coverage polygons of neighboring facades tend to overlap as some elements of a background facade may be visible through the volume bounding the foreground elements. While some pixels may be ray-traced more than once, final visibility is solved using classical depth buffering.

In addition to their position and normal in world space, the vertices of the facade also embed their position on the facade plane: $(0,0)$, $(1,0)$, $(1,1)$ and $(0,1)$ respectively. Those values express whether each vertex has to be displaced to the minimum or to the maximum extent of the coverage polygon as shown in Figure 3c.

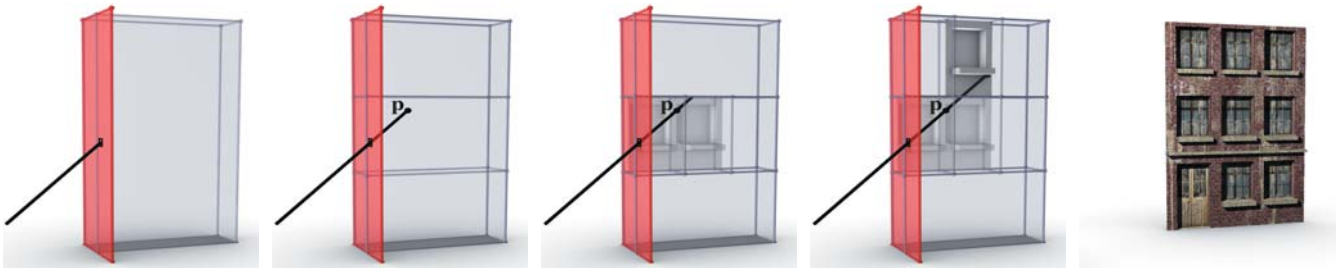
6 Per-Pixel Geometry Generation

For each fragment of the facade coverage polygon we introduce *ray-dependent grammar development* for efficient intersection of rays with the procedurally-generated elements. The intersection of the rays with the terminal functions of the grammar is then discussed in the cases of parametric surface sets and geometry images. Fine-scale details and self shadowing are then added for high quality rendering.

6.1 Ray-Dependent Grammar Development

A split grammar describes the structure of a procedural facade by dividing them successively along the axis of facade space, creating a hierarchy of blocks. For each split operation, the generated blocks are represented by intervals hence providing the widths of the blocks along the split axis. The size of the elements at each level of the grammar being known, the grammar itself can be considered as an implicit hierarchical space partitioning.

Figure 4 represents a simple procedural building generated using the grammar developed in Figure 5. When a ray enters the facade at a point $\mathbf{p}$, we perform a lazy search for the closest terminal symbol: the axiom F corresponds to a split into three parts of known height along the vertical axis. As the vertical component of $\mathbf{p}$ does not lie within the height interval of the ground floor GF, this rule is not developed. As $\mathbf{p}$ belongs to the interval of the first FL rule, we develop the rule to obtain a first terminal symbol W_f. The intersection function of W_f is then evaluated for the considered ray starting



(a) (b) (c) (d) (e)

Figure 4: For each fragment of the coverage plane, we consider the corresponding view ray towards the facade (a). The bounding volume of the facade is then intersected, yielding an entry point **p**. Following Algorithm 1, we identify the child rule FL which bounding volume contains **p** using the split parameters of the facade (b). As the children of FL are shape rules, we lazily test the view ray against their bounding volumes and perform the actual shape intersections only if needed (c). As no intersection is found within the volume of this FL rule, the algorithm steps to the next FL rule (c). We then identify the intersected shape rule and compute the ray-shape intersection (d). Once applied to each pixel of the image I, the entire procedural facade is rendered without any explicit generation of geometry (e).

at **p**. As no intersection if found we iterate over the next terminal symbol along the horizontal axis. In this example, the second terminal is neither intersected and the ray leaves the vertical interval of the current FL rule. The same process is then applied to the next FL rule, yielding the final intersection point. The traversal order and the intersection calculations are illustrated in Figure 5: only the shaded elements and split operations are developed to obtain the intersection point.

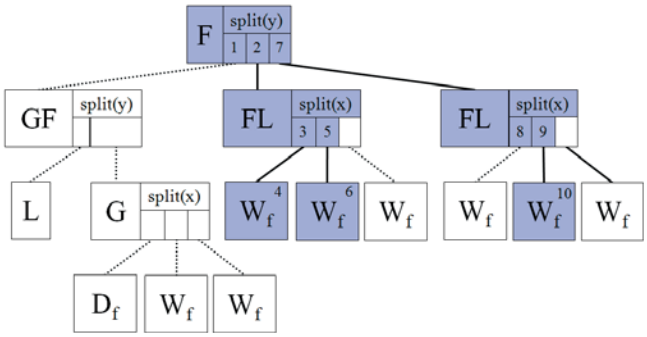


Figure 5: Given a view ray, the elements of the split grammar are developed only if they are potentially intersected: the grammar becomes an implicit acceleration structure for efficient ray tracing. The numbers represent the order of traversal for intersecting the ray of Figure 4 with the facade grammar.

The generalization of this example is provided in Algorithm 1: in the first step we intersect the view ray with the bounding volume of the facade (Figure 4b). If an intersection is found, we step through the facade by performing a conditional depth-first traversal of the grammar development tree: for each split rule, we first determine the child rule whose bounding box contains the current point on the ray (Figure 4c). If this child is a shape rule, we compute its actual intersection with the ray: if the ray intersects the shape, the traversal stops and we return the corresponding hit point. Otherwise, the current point is moved forward to the exit point of the bounding volume of the rule. If the child is a split rule, we iterate the child identification and recursively call the intersection calculations. Upon exiting the bounding volume of the rule, we determine the next intersected rule based on the ray direction and start over until the ray exits or intersects the facade (Figure 4d).

6.2 Ray-Terminal Intersection

Each terminal function intersects the view ray with its embedded definition of the procedural element, and returns the distance to the nearest intersection if any. In this paper we develop two types of terminal functions: parametric surface sets and geometry images.

6.2.1 Parametric Surface Set

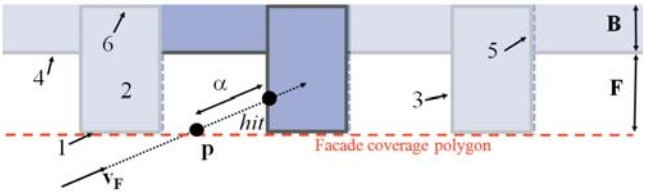


Figure 6: Intersection between the eye ray and the procedural elements in facade space. The numbers correspond to the labels of the parametric surfaces of a window block, as shown in Figure 7.

The grammar describing the facade elements is composed of rules and terminal symbols. For a terminal symbol such as a window block, we define a set of functions describing the surfaces of the element: in Figure 6, each block can be defined as a set of parametric surfaces. The general form of the equations representing the terminal symbols is:

$$\begin{array}{rcl} x_F & = & f_1(t_1,t_2) \\ y_F & = & f_2(t_1,t_2) \\ z_F & = & f_3(t_1,t_2) \end{array} \quad \text{with} \quad \left\{ \begin{array}{l} (f_1,f_2,f_3) \text{ continuous functions} \\ (t_1,t_2) \text{ real variables} \end{array} \right. \tag{1}$$

A location along the view ray in facade space is given by:

$$\mathbf{p_F} = \mathbf{a_F} + \alpha \mathbf{v_F} \tag{2}$$

Where $\mathbf{a_F}$ is the intersection of the ray with the coverage polygon in facade space, α is a positive scalar and $\mathbf{v_F}$ is the viewing direction in facade space. The intersection function for each terminal symbol determines t_1, t_2 and α in the following system of equations:

$$\begin{array}{rcl} \alpha \mathbf{v_F}.x + \mathbf{a_F}.x & = & f_1(t_1,t_2) \\ \alpha \mathbf{v_F}.y + \mathbf{a_F}.y & = & f_2(t_1,t_2) \\ \alpha \mathbf{v_F}.z + \mathbf{a_F}.z & = & f_3(t_1,t_2) \end{array} \tag{3}$$

More precisely, let us consider the top surface of the window sill within a window block using the notations of Figure 7. The generic

Algorithm 1 Ray-Dependent Grammar Development

```
bool intersectFacade(in Facade f, in Ray r, out Hit hit )
  Pos p
  if getBBoxIntersection( f, r, p ) then
    intersectRule( f.getAxiom(), p, r.getDir(), hit )
  end if

bool intersectRule( in Rule rule, inout Pos p, in Dir d, out Hit hit )
  if rule.isSplitRule() then
    return intersectSplit( rule, p, d, hit )
  else
    // Compute ray-terminal intersection, see section 6.2.
    return intersectShape( rule, p, d, hit )
  end if

bool intersectSplit( in Rule rule, inout Pos p, in Dir d, out Hit hit )
  Int index = rule.getSplitIndex( p )
  Rule childRule = rule.getChildRule( index )
  if intersectRule( childRule, p, d, hit ) then
    return true
  else
    Float distance = childRule.getTraversalDistance( p, d )
    p = p + distance . d
    Int direction = sign( projection( d, rule.getSplitAxis ) );
    Bool intersection = false
    while p ∈ rule.getBBox() && not( intersection ) do
      index = index + direction;
      childRule = rule.getChildRule( index )
      if intersectRule( childRule, p, d, hit ) then
        intersection = true
      else
        distance = childRule.getTraversalDistance( p, d )
        p = p + distance . d
      end if
    end while
    return intersection
  end if
```

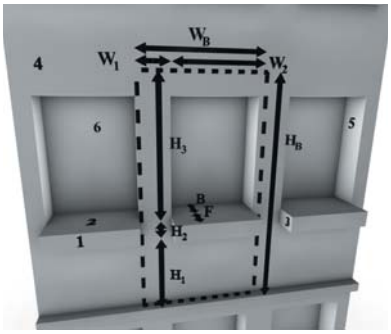


Figure 7: A window block of our example grammar is represented by a set of values describing the width, height and depth of each of its components. The bottom-left of the considered block is located at (X_B, Y_B) on the facade plane, derived from the parent split rule.

system of Equation 1 becomes:

$$\begin{array}{rcl} x_F &=& X_B\,\mathbf{W}_B + \mathbf{W}_1 + t_1 \\ y_F &=& Y_B\,\mathbf{H}_B + \mathbf{H}_1 + \mathbf{H}_2 \\ z_F &=& -t_2 \\ 0 \le t_1 &\le& \mathbf{W}_2 \\ 0 \le t_2 &\le& \mathbf{B}+\mathbf{F} \end{array} \qquad (4)$$

where X_B and Y_B are the origin of the window block along $\mathbf{t_F}$ and $\mathbf{b_F}$. Similar equations describe the remainder of the surfaces of the element, hence providing a simple framework for ray-procedure intersection. Further performance is achieved by ordering the surfaces with respect to their distance to each projection plane: this allows our method to test the intersection of rays with the surface set in a front-to-back order to avoid unnecessary computations.

While effective, the definition of equations describing complex geometries may become intractable without harming the creativity of the design artist. Geometry images are another way of encoding geometry details for fast per-fragment intersection calculations.

6.2.2 Geometry Images

Geometry images [6, 10, 17, 5] are images describing the structure of objects such as triangle meshes: pixels of the texture represent the coordinates of vertices as well as optional attributes such as adjacency. Unlike parametric surfaces, such textures can be easily extracted from most 3D file formats. Also, ray-triangle intersections can be efficiently implemented on graphics hardware. Therefore, using the grammar defined in Section 3, we used geometry images as terminal functions, yielding more varied building appearances as shown in Figure 8. This technique increases the power of expression, while maintaining interactive performance. Note that our implementation only considered brute-force ray tracing, in which every triangle of the element is tested for intersection. Further performance will be achieved in the future by supporting hierarchical acceleration structures embedded within the geometry image.

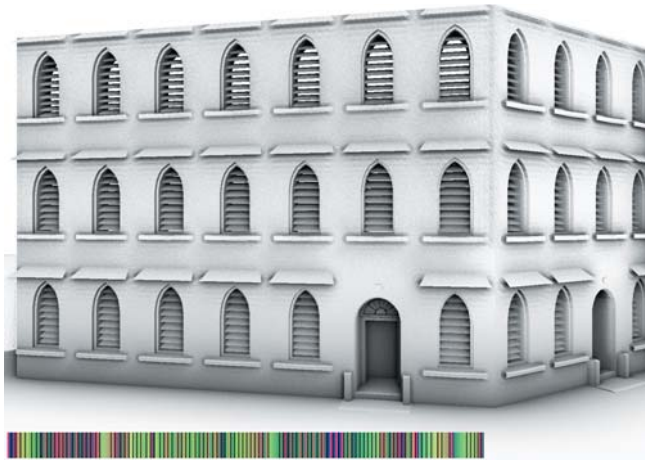


Figure 8: Procedural building using a grammar for which terminal symbols are geometry images: we only store the 1D textures representing a window block and a door block. The geometry image of a window block is shown at the bottom (image is stretched on the vertical axis for legibility). The building is rendered at $6-30$ frames per second (depending on the screen coverage) for resolution of 1024×768 using brute-force ray tracing within terminal functions. The window and door blocks contain 286 and 732 triangles respectively.

The geometry of parametric surface sets and geometry images tends to remain relatively simple, especially in the context of real-time rendering. However, such coarse geometries can then be enriched by fine details added using relief mapping techniques for improved visual quality.

6.3 Relief Mapping on Terminal Elements

The procedural elements describe the macroscopic surfaces of the facade. Further details such as brick relieves are added using relief mapping [16, 15] on the intersected terminal symbol (Figure 9). For each point within its bounding volume, each terminal symbol also defines its local *terminal space* $(\mathbf{t_T}, \mathbf{b_T}, \mathbf{n_T})$, expressed in facade space. Using geometry images the terminal space is the tangent space of the underlying geometry at the considered point. For

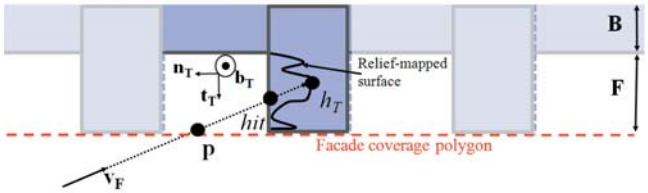


Figure 9: Procedural elements are enriched with additional details using relief mapping in terminal space.

parametric surface sets the terminal space is:

$$\mathbf{t_T} = \left(\frac{\partial f_1}{\partial t_1}, \frac{\partial f_2}{\partial t_1}, \frac{\partial f_3}{\partial t_1}\right) \quad \mathbf{b_T} = \left(\frac{\partial f_1}{\partial t_2}, \frac{\partial f_2}{\partial t_2}, \frac{\partial f_3}{\partial t_2}\right) \quad \mathbf{n_T} = \mathbf{t_T} \times \mathbf{b_T}$$

For example the terminal space of the top surface of the window sill is $\mathbf{t_T} = (1,0,0)$, $\mathbf{b_T} = (0,0,-1)$ and $\mathbf{n_T} = (0,1,0)$.

When a ray intersects a surface described by a terminal symbol, the intersection point is projected into terminal space. Relief mapping is then applied to obtain the actual intersection point on the fine geometry details, as well as the corresponding reflectance properties. The advantages of using relief mapping are multiple: this technique can be used to generate arbitrarily fine details, and is inherently scalable. Also, self shadowing can be easily added for high quality lighting on both procedural and relief-mapped details.

6.4 Self Shadowing

Our technique particularly targets interactive design and visualization of the shape and appearance of procedural buildings and cityscapes. The lighting effects may partly drive the design process, and are mandatory for the final rendering. Building upon [16, 15] and our ray-dependent grammar development, we propose a solution for real-time, per-pixel self shadowing due to point light sources. As illustrated in Figure 10, our method comprises three main steps: the computation of the entry point of light within the facade volume, the intersection with the procedural elements, and visibility tests within the relief-mapped surfaces.

Given the location of the light source with respect to the facade, we determine a light-facing plane as in Section 5, and determine the corresponding facade space. After projecting the light ray into facade space, we intersect the ray with the plane. Then, the intersection of the light ray with the procedural elements is performed as for the view ray (Section 6.2.1). If the ray intersects the same element as the view ray, we compute self shadowing within the relief-mapped details as in [16, 15]. If the considered point is lit, we compute the amount of light reflected towards the viewpoint.

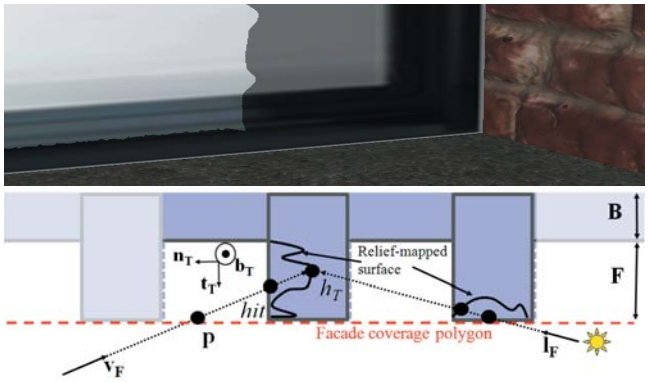


Figure 10: Self shadowing is performed by intersecting the shadow rays with both the procedural elements and the microscopic details, yielding the fine shadow details shown in the top image.

7 Levels of Detail

Our method makes intensive use of level of detail for scalable rendering of facades (Figure 11). For instance, relief-mapped details and self shadowing may not be necessary when viewing a building from a long distance. We introduce the following levels: in the coarser level (*i.e* for distant buildings), the macroscopic thickness is set to zero. The protrusions and recesses appear in the next level, allowing for macroscopic self shadowing, but with standard texture mapping of the facades. In the last two levels, we add normal mapping then relief-mapping, leading to self shadowing of microscopic details. A classical method for avoiding popping artifacts when switching discrete levels of details requires rendering the object at both levels of detail and blending the resulting images. Instead, we linearly attenuate the effects of each level with respect to the distance. For example, the transition from the coarsest level to the next is achieved by linearly increasing the thickness of the procedural elements. The addition of normal and relief mapping is performed using a smooth transition from geometry-based to normal-mapped normals. Simultaneously the overall thickness of the relief-mapped details is linearly interpolated from zero to the final thickness.

8 Discussion: Genericity

The lazy grammar development devised in the previous section can be applied to a wide range of split grammars. However, several aspects cannot be handled using our technique. In particular, our approach provides fast ray-terminal intersection based on the use of split rules as a bounding volume hierarchy. The induced limitations are twofold: first, a notion of upper bounds for the extents of the protrusions and recesses must be provided either empirically or using bottom-up traversal of the grammar. Second, the terminal shapes are not allowed to stick out of their parent bounding box: the BVH structure would lose consistency, resulting in visibility errors.

Another potential genericity issue relates to extended features of the split grammars such as the component splits. This technique applies rules to specific geometric elements (faces, edges or vertices). While generally not complying with our direct BVH representation, component splits are often used to simply differentiate facades based on their semantic: front, sides etc. Our method directly handles this particular case by tagging the facades and applying grammar elements accordingly.

9 Results

We implemented our technique using the programmable shaders available on current graphics hardware: the extraction of the facade volume is performed in a vertex shader, while a fragment shader contains the procedural rules, the intersection functions of the terminal symbols and the relief mapping functions.

For our tests, we used our sample grammar to render images with resolution 1280×720. The parameters of the grammar are typically the number of floors, the size of the protrusions F and recesses B and the number of window blocks on each floor. We generated a complete cityscape composed of 10K facades of various sizes and colors (Figure 1a). A complete geometry generation of the city including normals and texture coordinates would require approximately 400MB. Our compact representation reduces the overall storage to 650KB, comprising the grammar parameters and building footprints. As these memory requirements are nearly independent from the grammar and number of terminal shapes, more complex descriptions would benefit even further from our method.

The texture maps have a resolution of 1024×1024 and are divided into two sets: per-element textures represent the set of diffuse RGB reflectances and RGBA textures representing the normals and relieves of each type of terminal symbol. The second texture set stores per-facade or per-building information such as dirt. The city contains 58 per-element maps and 22 per-facade maps, yielding an

(a) (b) (c) (d)

Figure 11: In this procedural cityscape , ~ 10000 facades are simultaneously modified (elevation, size of protrusions and recesses) on-the-fly at 3.5fps (see video). Depending on the distance to the viewpoint, our algorithm performs smooth and inexpensive transitions between 4 main levels of detail: flat buildings (a), macroscopic geometry and shadowing (b), normal mapping (c) and relief mapping (d).

additional memory load of 280MB without compression. The overall representation of the city is then 280.7MB. Note that unlike existing methods, the per-element textures do not contain the full facade description: they only describe the fine high definition details of each terminal shape for high quality rendering on both distant and extreme closeup views. We obtain an average rendering speed of 7 frames per second on an nVidia GeForce GTX480.

10 Extension to generic meshes

In the previous sections we considered the generation of procedural facades on buildings defined by their extruded footprint. To provide complete artistic freedom we extend our approach to more complex structures such as stylized buildings by extending the concepts of facade space and facade coverage polygon over quad-based meshes.

10.1 Facade Space

In Section 5.1, the use of an extruded footprint provides a simple way of obtaining the facade space. Using arbitrary meshes, the assumption of vertical, planar facades does not hold anymore. We propose a method to create a coherent set of facade spaces all over the mesh by defining a local space for each quadrilateral of the building. To this end, we assume that each vertex of the mesh contains its normal as well as coherent 2D texture coordinates (u,v) which define the location of the vertex in facade space. For each vertex, the basis vectors of the facade space are constructed from properties of adjacent vertices using *selective adjacency extraction*.

This extraction is the computation of orthogonal vectors respectively collinear with the gradients of the texture coordinates u and v. In a first step, we select two vertices $\mathbf{p_i}$ and $\mathbf{p_j}$ in the direct neighborhood of the considered vertex $\mathbf{p}$ with texture coordinates (u,v) so that $\mathbf{pp_i}\cdot\mathbf{pp_j}$ is minimal (Figure 13). However, depending on the curvature of the mesh, $\mathbf{p_i}$ and $\mathbf{p_j}$ may not be located on the tangent plane of $\mathbf{p}$. To overcome this problem and ensure the generated tangent and binormal vectors are orthogonal to the vertex normal $\mathbf{n}$, we project $\mathbf{p_i}$ and $\mathbf{p_j}$ on the tangent plane as follows:

$$\mathbf{p_i'} = \mathbf{p_i} + (\mathbf{n}\cdot\mathbf{p} - \mathbf{n}\cdot\mathbf{p_i})\mathbf{n} \tag{5}$$

The tangent and binormal vectors are then obtained using the formulas introduced by Gath[1]:

$$\mathbf{t_F} = \left\| \frac{(v_i - v)(\mathbf{p_j'} - \mathbf{p}) - (v_j - v)(\mathbf{p_i'} - \mathbf{p})}{(u_j - u)(v_i - v) - (u_i - u)(v_j - v)} \right\| \tag{6}$$

$$\mathbf{b_F} = \left\| \frac{(u_i - u)(\mathbf{p_j'} - \mathbf{p}) - (u_j - u)(\mathbf{p_i'} - \mathbf{p})}{(u_i - u)(v_j - v) - (u_j - u)(v_i - v)} \right\| \tag{7}$$

where (u_i, v_i) are the texture coordinates of vertex $\mathbf{p_i}$ and $\|.\|$ is the normalization operator. This extraction step is performed once and for all as a preprocess, and stored as additional vertex attributes within the mesh. As long as the topology of the object remains unaltered, those attributes are used at runtime to extend the concept of facade volume to arbitrary animated quad-based meshes.

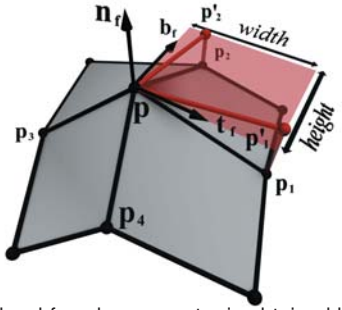


Figure 13: The local facade space at $\mathbf{p}$ is obtained by selecting two adjacent vertices $\mathbf{p_1}$ and $\mathbf{p_2}$ creating nearly orthogonal vectors and projecting those vertices onto the tangent plane of $\mathbf{p}$. The width and height of the corresponding part of the facade mapped on the quadrilateral are given by the extents of the facade-aligned bounding box containing $\mathbf{p}$, $\mathbf{p_1}$ and $\mathbf{p_2}$.

10.2 Facade Volume

The use of generic quadrilaterals instead of plain vertical facades requires an extension of the concept of facade volume. In this context we consider each quadrilateral as a separate facade whose width and height are determined in facade space as follows (Figure 13):

$$w = \max(|\mathbf{pp_i'}\cdot\mathbf{t_F}|, |\mathbf{pp_j'}\cdot\mathbf{t_F}|), \quad h = \max(|\mathbf{pp_i'}\cdot\mathbf{b_F}|, |\mathbf{pp_2'}\cdot\mathbf{b_F}|) \tag{8}$$

The choice of the location of each facade vertex is performed in the same spirit as for footprint-based generation, using the texture coordinates instead of the simple 0 and 1 values (Section 5.3.2). Finally, the other steps described in Section 6 remain identical, yielding a general solution for *procedural geometry mapping*.

10.3 Results and Discussion

Procedural details have been added to generic objects using the grammar of Section 3. Our technique preserves real-time performance and dynamic editing (Figure 12).

However, this extension of our approach suffers from the drawbacks of the techniques based on relief mapping: if the distance traversed by the ray is high, the local spaces of the entry point and the actual hit may be inconsistent especially on low-poly, highly curved meshes (Figure 14a). The pathological case illustrated in Figure 14b exhibits view-dependent, varying curvatures of the procedural elements. For the same reasons, this approximation may also generate errors in the self shadowing estimation. While artifacts are unnoticeable on most meshes, curvature compensation using higher order representation of facade space could be considered to avoid gaps or discontinuities.

[1] http://www.blacksmith-studios.dk/projects/downloads/tangent_matrix_derivation.php

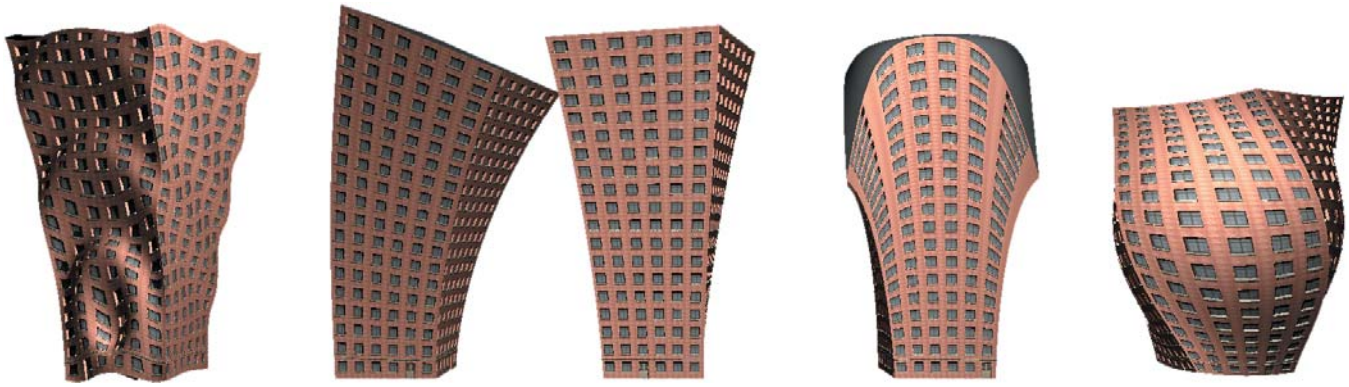
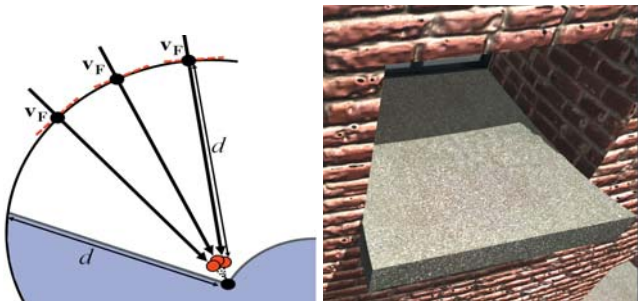


Figure 12: Procedural geometry mapping applied on generic meshes directly exported from standard modeling softwares: the parameters of the grammar as well as the texture coordinates can be adjusted in real-time. The geometry mapping can thus be interactively tuned as for texture coordinates, providing complete artistic control. Topology-preserving mesh deformation could be directly applied for real-time animation.



(a) (b)

Figure 14: On highly curved meshes featuring relatively large protrusions or recesses, facade space becomes inconsistent along the view rays. This results in view-dependent errors on intersection points shown as red dots in (a). Those errors appear as artificial distortions of straight lines in the final rendering (b).

11 Conclusion

The modeling of buildings and cityscapes generally involves complex parameterizations of rules for procedural generation, and requires large amounts of memory for rendering. We open a way towards a novel, artist-friendly approach to the design of procedural buildings avoiding the need for explicit geometry generation and storage. Based on a per-pixel partial grammar development on graphics hardware, our technique achieves interactive performance. In particular, we preserve the ability to tune the generation parameters on-the-fly even on entire cityscapes. The procedural details are generated per-pixel onto extruded footprints of buildings or on arbitrary meshes, resulting in negligible storage costs compared to previous approaches. This latter aspect makes our method particularly useful for streaming or transferring virtual worlds over low-bandwidth networks. Furthermore, the entire process runs on graphics hardware, relieving the central processor from the generation task. The design of visually pleasant buildings requires fine artistic control, which is usually not available or intuitive in existing methods for procedural generation. Procedural geometry mapping provides intuitive artistic control over both the overall shape of the buildings and the arrangement of the procedural elements. Based on texture coordinates, our method brings procedural modeling into any production workflow. Future work will particularly consider the issues related to the mapping of procedural elements on highly curved meshes, for example by replacing the tangent planes by higher order representations.

Acknowledgements

The authors wish to thank Cyril Delalandre for assembling and compositing the video accompanying this paper.

References

[1] S. Ali, J. Ye, A. Razdan, and P. Wonka. Compressed facade displacement maps. *IEEE Transactions on Visualization and Computer Graphics*, 15(2):262–273, 2009.

[2] L. Baboud and X. Décoret. Rendering geometry with relief textures. In *Proceedings of Graphics Interface*, 2006.

[3] M. Bunnell. Adaptive tessellation of subdivision surfaces with displacement mapping. *GPU Gems 2*, pages 109–122, 2005.

[4] P. Cignoni, M. Di Benedetto, F. Ganovelli, E. Gobbetti, F. Marton, and R. Scopigno. Ray-casted blockmaps for large urban models visualization. *Computer Graphics Forum*, 26(3), 2007.

[5] R. de Toledo, B. Wang, and B. Levy. Geometry textures and applications. *Computer Graphics Forum*, 27(8):2053–2065, 2008.

[6] X. Gu, S. J. Gortler, and H. Hoppe. Geometry images. In *Proceedings of SIGGRAPH*, pages 355–361, 2002.

[7] S. Haegler, P. Wonka, S. M. Arisona, L. V. Gool, and P. Muller. Grammar-based encoding of facades. In *Proceedings of Eurographics Symposium on Rendering*, pages 1479–1487, 2010.

[8] X. Huang, S. Li, and G. Wang. A GPU based interactive modeling approach to designing fine level features. In *Proceedings of Graphics Interface*, pages 305–311, 2007.

[9] M. Lipp, P. Wonka, and M. Wimmer. Interactive visual editing of grammars for procedural architecture. In *Proceedings of SIGGRAPH*, pages 1–10, 2008.

[10] F. Losasso, H. Hoppe, S. Schaefer, and J. Warren. Smooth geometry images. In *Proceedings of Eurographics Symposium on Geometry Processing*, pages 138–145, 2003.

[11] J. Marvie, J. Perret, and K. Bouatouch. Remote interactive walkthrough of city models. In *Proceedings of Pacific Graphics*, pages 389–393, 2003.

[12] J. Marvie, J. Perret, and K. Bouatouch. The fl-system: a functional l-system for procedural geometric modeling. *The Visual Computer*, 1(5):329–339, May 2005.

[13] P. Muller, P. Wonka, S. Haegler, A. Ulmer, and L. Gool. Procedural modeling of buildings. In *Proceedings of SIGGRAPH*, pages 614–623, 2006.

[14] Y. Parish and P. Muller. Procedural modeling of cities. In *Proceedings of SIGGRAPH*, pages 301–308, 2001.

[15] F. Policarpo and M. Oliveira. Relaxed cone stepping for relief mapping. *GPU Gems 3*, pages 409–428, 2007.

[16] F. Policarpo, M. Oliveira, and J. Comba. Real-time relief mapping on arbitrary polygonal surfaces. In *Proceedings of I3D*, pages 155–162, 2005.

[17] P. Sander, Z. Wood, S. Gortler, J. Snyder, and H. Hoppe. Multi-chart geometry images. In *Proceedings of Eurographics Symposium on Geometry Processing*, pages 146–155, 2003.

[18] L. Szirmay-Kalos and T. Umenhoffer. Displacement mapping on the GPU - State of the art. *Computer Graphics Forum*, 27(1):1567–1592, 2008.

[19] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky. Instant architecture. In *Proceedings of SIGGRAPH*, pages 669–677, 2003.