EUROGRAPHICS 2013 / I. Navazo, P. Poulin
(Guest Editors)

Volume 32 (2013), Number 2

View-Dependent Realtime Rendering of Procedural Facades with High Geometric Detail

Lars Krecklau, Janis Born, Leif Kobbelt

RWTH Aachen University, Germany

Abstract

We present an algorithm for realtime rendering of large-scale city models with procedurally generated facades. By using highly detailed assets like windows, doors, and decoration such city models can provide an extremely high geometric level of detail but on the downside they also consist of billions of polygons which makes it infeasible to even store them as explicit polygonal meshes. Moreover, when rendering urban scenes usually only a very small fraction of the city is actually visible which calls for effective culling mechanisms. For procedural textures there are efficient screen space techniques that evaluate, e.g., a split grammar on a per-pixel basis in the fragment shader and thus render a textured facade in a view dependent manner. We take this idea further by introducing 3D geometric detail in addition to flat textures. Our approach is a two-pass procedure that first renders a flat procedural facade. During rasterization the fragment shader triggers the instantiation of a detailed asset whenever a geometric facade element is potentially visible. The set of instantiated detail models are then rendered in a second pass. The major challenges arise from the fact that geometric details belonging to a facade can be visible even if the base polygon of the facade itself is not visible. Hence we propose measures to conservatively estimate visibility without introducing excessive redundancy. We further extend our technique by a simple level of detail mechanism that switches to baked textures (of the assets) depending on the distance to the camera. We demonstrate that our technique achieves realtime frame rates for large-scale city models with massive detail on current commodity graphics hardware.

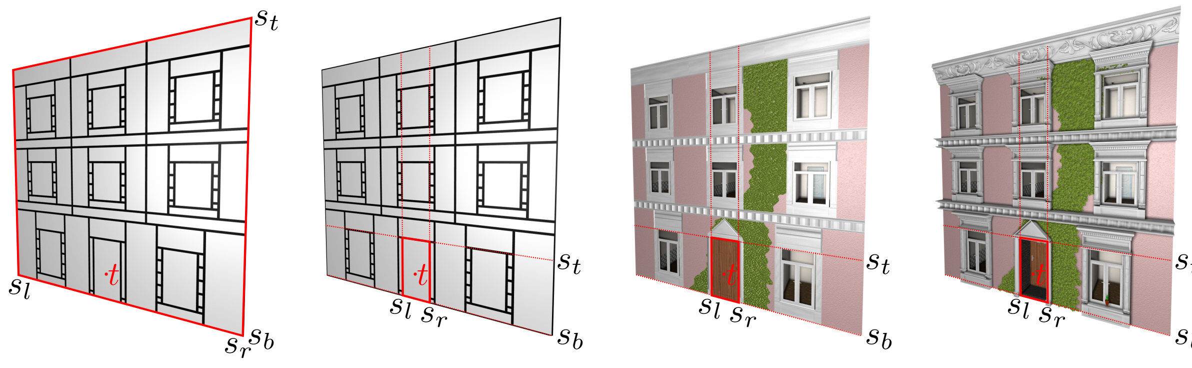
Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Languages—I.3.2 [Computer Graphics]: Graphics Systems—I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—

1. Introduction

Renderings of cityscapes appear in a variety of real-time graphics applications such as computer games, three-dimensional street maps, urban planning, or traffic simulation. Users expect such renderings to look visually convincing and to offer rich detail and variety. Even though in theory the massive amount of rendered elements could be hierarchically structured to achieve real-time performance, the excessive amount of required memory stays a major problem. Instead of optimizing the rendering of massive data, procedural descriptions have become a popular solution to encode the complexity of urban environments into a set of code snippets. Consequently, the complete model never needs to be fully generated and the memory footprint stays small.

Although the memory consumption is heavily reduced by procedural techniques, generating objects on-the-fly introduces additional computation. We utilize the GPU for this purpose which provides hundreds of cores in a SIMD architecture, such that many evaluations of the procedural description can be processed in parallel. Especially for large scale city models, Haegler et al. [HWA*10] have recently shown visually pleasing real-time renderings by rasterizing a coarse proxy of the scene and evaluating a high quality procedural description of the facades on a per-pixel basis (cf. Figure 1a - Figure 1c). Due to the application of a screen space technique, the rendering performance is majorly dependent on the screen resolution and not on the complexity of the scene which is a great advantage assuming that city models will even grow larger in the future.

A per-pixel evaluation of procedural facade textures tackles a lot of memory and performance issues, however, the coarse proxy does not provide any real geometric details.



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

Figure 1: The per pixel evaluation starts with a scope $s = (s_l, s_b, s_r, s_t)$ of the full facade dimension at texture coordinates $t = (t_x, t_y)$ (a). After tracing down several subdivision steps, which will successively reduce the size of s, the evaluation reaches a scope that is associated with a terminal operation (b). At this point we can either sample from a texture (c) or instantiate a new geometric item (d). Note that the texture as well as the geometry will be scaled to perfectly fit into the scope.

We overcome this limitation by enriching the rendering with newly instantiated polygonal items directly in screen space. Instead of sampling from a texture during the per-pixel evaluation, the procedural description is allowed to instantiate new geometric elements (cf. Figure 1d).

Unfortunately, this is not a trivial task for two major reasons. On the one hand, a texture can cover many pixels which might lead to instantiating the same geometric item many times. Therefore, utilizing the parallel processing power of the GPU demands for sophisticated caching mechanisms that minimize locking by performing as many operations asynchronously as possible and avoid expensive atomic operations. On the other hand, our technique will produce elements that protrude the coarse proxy and hence, we need an efficient way to conservatively determine all potentially visible elements. For example, objects that are defined on back facing facades but reaching into the visible region of the viewer still need to be instantiated properly.

This paper shows a novel solution for the realtime rendering of detailed city models with focus on these key aspects:

Pixel Driven Geometry Instantiation Having a set of rules for the procedural definition of facade textures we present two new operators for the instantiation of geometric items. When a procedural facade texture is evaluated on a per-pixel basis, it majorly traces down a subdivision structure that is defined by a *split grammar* until a region is found in which a texture is sampled. Instead of only using a 2D projection of windows, doors, cornices, and other architectural items, the first operator will fit a geometric item into the rectangular domain with a defined displacement along the normal of the facade. Furthermore, we show the concept of *attach textures* which are also defined in the rectangular domains to place a number of geometric items in a pseudo chaotic fashion in order to visualize fine details like gravel stones, grass, or ivy.

Visibility Determination The most problematic issue for our screen space techniques are global visibility issues like partly occluded geometric details. We present an efficient solution to tackle these problems by introducing only a very small overhead in the rendering pipeline.

Two Pass Rasterization Many previous approaches enhance textures by raytracing techniques such as relief mapping [POC05]. The rasterization only serves as a fast way to render a coarse bounding volume that initializes the raytracer to display the fine geometric surface. Evaluating a split grammar on a per-pixel basis corresponds to this process, however, placing geometry on the facade raises two issues. First, the geometry might protrude the coarse proxy of the city model and second, the raytracing of polygonal models becomes inefficient without embedding the geometry into a hierarchical data structure. We solve this by an additional rasterization step, i.e. the evaluation of the facade texture will produce an additional buffer of geometric items that are rendered in a second rasterization step.

2. Related Work

A common paradigm of procedural architecture is to start with the coarse volumetric representation of a building which is then recursively broken down into smaller subshapes. Finally, all resulting sub-shapes are replaced by geometric primitives. The concept of iteratively refining shapes has been formalized using *shape grammars* by Stiny [Sti80]. A specialization of this approach which is more suitable for algorithmic evaluation are *split grammars* [WWSR03]. Müller et al. extended split grammars by context sensitivity and have coined commonly used formalisms with their *CGA Shape* grammar [MWH*06]. This approach has become particularly famous for the reconstruction and the interactive creation of single facades or buildings providing a high level of detail [MZWG07, LWW08, MWW12].

For large scale city models, a highly detailed impression of the facades can be achieved by a per-pixel evaluation [HWA*10, KK11]. Similar to CGA Shape, a set of subdivision operations (i.e. *split* and *repeat*) is used to describe the structure of the facade, however, the textures are not explicitly generated but implicitly queried on a per-pixel basis. The terminal elements in these approaches are still textures and thus, the visual expression becomes negatively affected if the viewer has a very sharp angle to a facade since it suddenly appears to be flat. We overcome this limitation by instantiating new geometric items during the evaluation of a facade texture and rasterizing these elements in a succeeding rendering pass.

Marvie et al. have also shown a way to display real 3D geometry by a renderer that is entirely based on ray casting on the GPU [MGHS11]. The idea is to render a coarse coverage polygon for each facade and then perform ray casting through the facade space in the fragment shader. They combine the simulation of geometric detail using the concepts of interior mapping for inner rooms [vD08], relief mapping for structural details of the facade materials [POC05], and geometry images for real geometric items that are placed on the facade [GGH02, dTWL08]. Most importantly, the geometry images are evaluated by brute force testing all ray/triangle intersections, which is fairly inefficient even for low-polygon models. In contrast, we perform another rasterization stage which performs much better on current GPUs, even for geometric models with a high polygon count.

Since our approach utilizes the fragment shader of the GPU, many pixels might evaluate the same rules and therefore also want to instantiate the same geometric details. We prevent this duplicated instantiation by a cache that is inspired by GPU hashing techniques as presented by García et al. [GLHL11] and Reiner et al. [RLD*12]. While their caching method is used to perform expensive calculations only once such that subsequent evaluations can reuse the cached value, we take this idea to guarantee that only one fragment (in the set of fragments that want to instantiate the same geometry) is actually allowed to provide the needed information for the succeeding rasterization stage.

3. Per-Pixel Facade Textures

For the approach presented in our paper we build upon existing concepts for generating procedural facade textures on a per-pixel basis such as those presented by Haegler et al. [HWA*10] or Krecklau et al. [KK11]. We will not go into detail with these techniques, however, a brief summary of the most important ideas and operators is supportive to be able to follow our main algorithm.

Procedural facade textures are defined by a set of *rules* of the form $\rho \rightarrow \omega(p_1, \dots, p_n)$ where ρ is a *rule name* and ω is an *operator type* from an application-defined set. The number n of *operator parameters* $p_1, \dots, p_n$ depends on the operator type. Scalar parameters are mostly used to define certain sizes (e.g. how large is a certain part defined by a split operation) whereas rule names as parameters are used to associate a certain element, that results from an operator, with a new rule. Therefore, from a the theoretical view of a procedural grammar description, operators providing rule names as parameters can be seen as *non-terminals* (e.g. `SplitX`, `RepeatY`) since a further evaluation is needed in that pixel while operators without rule name parameters can be seen as *terminals* (e.g. `SampleTexture`, `SetColor`).

During the process of the evaluation all operators have access to a set of globally defined attributes per pixel. The *scope* $s = (s_l, s_b, s_r, s_t) \in \mathbb{R}^4$ defines an axis aligned bounding rectangle on the facade which is set to $s = (0, 0, w, h)$ at the beginning of the evaluation with w and h defining the absolute size of the facade in world space. The texture coordinates $t = (t_x, t_y) \in \mathbb{R}^2$ result from the corresponding pixel within the domain s. During the evaluation, the scope is broken down into smaller elements by the split operations until a terminal operator is found which sets the color $c = (c_r, c_g, c_b) \in [0, 1]^3$ of the pixel (cf. Figure 1).

4. Pixel Driven Geometry Instances

The major idea of our approach is to trigger the instantiation of geometric objects on the procedurally generated facades directly in screen space. Given the concept of per-pixel facade textures we introduce the new operators `AttachScope` and `AttachTexture` to allow for attaching detailed geometry based on the current *attachment scope* or based on a manually designed *attachment texture*, respectively.

Similar to previous work, the input to our system is a coarse polygonal mesh which consists of extruded floor plans where each facade is represented by a planar rectangle. Our rendering pipeline first performs a z prepass on the model to reduce the overdraw for the expensive per-pixel evaluation of the grammar. This is especially advisable for street level camera views where a lot of buildings occlude each other. In the second pass, the model is rasterized again such that the grammar evaluation is performed once per fragment. In contrast to previous work, this pass will not only calculate a color per pixel but also fill an *attachment buffer* with information about the currently visible attachment geometry. Based on the depth, the system will automatically choose between using the baked texture of the geometry or putting all needed information into the attachment buffer to render the attached geometry as a polygonal model in the next step. In the final pass, the attachment buffer is used to rasterize the attached geometry.

Our pipeline avoids raytracing techniques as often as possible, since the current graphics hardware performs much faster in rasterization. Therefore, we still achieve realtime performance (i.e. more than 24 frames per second) for huge scenes providing a high amount of detailed geometric items (see our accompanying video).

5. Attachment Scopes

A straightforward extension to the per-pixel facade textures is to provide the following new operator:

$$\texttt{AttachScope}(g, s_z, f_{\text{near}}, f_{\text{far}})$$

It takes a polygonal geometry g and scales it such that its width and height perfectly fit into the current scope which is here referred to as *attachment scope*. Before the rendering of the city is performed, a designer needs to create a collection of polygonal models (like windows, doors, or cornices) that should be used as facade details. Hence, the parameter g is a zero-based ID value that specifies one element in this predefined set of *detail object types*. The actual instances generated by `AttachScope` are then called *detail objects*.

Conceptually, the idea is similar to sampling a texture that is fit into the current scope except that the user defines an additional scalar parameter s_z for the depth, i.e. the scale factor along the facade normal. Algorithmically, however, this extension requires a sophisticated strategy, because attaching a new geometry can not simply be done on a per-pixel basis and needs to be placed on hold for a later rasterization step.

The last two parameters f_{near} and f_{far} are used to blend between the polygonal geometry and its corresponding baked texture. Based on these values, a visibility value v is calculated for every detail object. Given the distance d of a detail object to the camera near plane, v is defined as follows:

$$v = \text{clamp}\left(\frac{f_{\text{far}} - d}{f_{\text{far}} - f_{\text{near}}}, [0,1]\right)$$

If $v = 1$ ($d < f_{\text{near}}$), the detail object is fully visible and should be displayed. For $0 < v < 1$, the detail object is faded out gradually using alpha blending. The value of v is used as its alpha value. If $v = 0$ ($d > f_{\text{far}}$), the detail object is too far away from the viewer and should no longer be rendered. As detail objects fade out, they are replaced by a 2D texture image of their projection into the facade.

5.1. Detail Object Instantiation

Since fragment shaders cannot issue the rasterization of new primitives, the `AttachScope` operator will store a list of detail object instances in an auxiliary buffer of floating point values, which we refer to as the *attachment buffer*. More precisely, we store the world space transformation of the detail object and the fadeout value which are utilized in subsequent render passes.

The attachment buffer is logically divided into equally sized sections (one per detail object type) where each section has a capacity of $n_{\max}$ detail objects. For each detail object, we need to store its world space transformation (12 values) and the current fadeout alpha value (1 value) summing up in a stride length of $n_{\text{stride}} = 13$. Due to this structure, we find the detail object with index i of type j at position $p(i,j) = (j \cdot n_{\max} + i) \cdot n_{\text{stride}}$ in the attachment buffer.

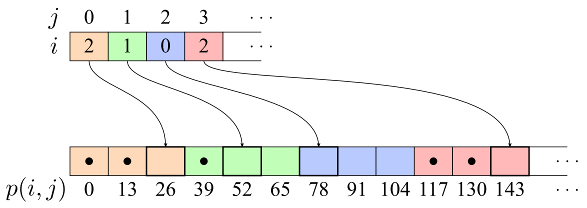


Figure 2: Layout of the attachment counter buffer (top) and attachment buffer (bottom, occupied slots marked with black dots). The next free slot in the attachment buffer for a detail object of type j is found using its counter value i. In this example, we use a section size of $n_{\max} = 3$ detail objects and a stride length of $n_{\text{stride}} = 13$.

Assuming that the detail objects are defined within a unit cube ($[-1,1]^3$), the affine transformation is calculated as follows. We first obtain the local transformation $\mathbf{F_l} \in \mathbb{R}^{4\times 4}$ within the facade by using the center and size of the current scope $s = (s_l, s_b, s_r, s_t) \in \mathbb{R}^4$ and the scaling factor s_z:

$$\mathbf{F_l} = \begin{pmatrix} \frac{s_r - s_l}{2} & 0 & 0 & \frac{s_r + s_l}{2} \\ 0 & \frac{s_t - s_b}{2} & 0 & \frac{s_t + s_b}{2} \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The local coordinate system $\mathbf{F_t} \in \mathbb{R}^{4\times 4}$ of the facade is defined by its tangent $\mathbf{t}$, bitangent $\mathbf{b}$, normal $\mathbf{n}$ and the position $\mathbf{o}$ of its lower left corner as follows:

$$\mathbf{F_t} = \begin{pmatrix} \mathbf{t} & \mathbf{b} & \mathbf{n} & \mathbf{o} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The final affine transformation that needs to be stored is then given by the upper 3×4 matrix of $\mathbf{M} = \mathbf{F_t} \cdot \mathbf{F_l}$.

In addition to the attachment buffer, we maintain an *attachment counter buffer*. For every detail object type, it stores an integer value representing the number of detail objects that have been instantiated. Given a counter value i, the next free location for detail object type j in the attachment buffer is $p(i,j)$ as illustrated in Figure 2.

The attachment buffer and attachment counter buffer are bound as image uniforms in the facade evaluation fragment shader to allow for read and write access, both asynchronously and atomically. Instantiating a detail object from within the fragment shader then consists of these steps:

1. Atomically increment the counter for the respective detail object type j by one. In OpenGL, this is achieved by an `imageAtomicAdd` operation on the attachment counter buffer at position j, where `imageAtomicAdd` returns the original value i of the counter before incrementing.
2. Write the detail object data to the attachment buffer at position $p(i,j)$. Since the detail object atomically obtained a unique position in the buffer, no two detail objects write their data to the same buffer position and hence, writing can be done asynchronously.

The data for each detail object type is tightly packed at the beginning of each section in the attachment buffer. Hence, we use this data for instanced rendering, i.e. given the counter value i for the object type j, we directly render i instances of the same model. Each individual detail object accesses the previously stored transformation in the vertex shader to transform it into world space. A handy implementation detail is the usage of `glDrawArraysIndirect` in OpenGL, which issues a draw call for instanced rendering from a buffer without the detour of downloading the data.

5.2. Duplication Prevention

Whenever one fragment of an attachment scope becomes visible, a detail object should be instantiated. However, if every fragment would proceed in this way, an unmanageable amount of duplicated detail objects would be created.

We prevent duplicates by a caching mechanism, allowing each fragment of an attachment scope to query whether a detail object has already been instantiated. Basically, a unique *detail object ID* is assigned to each detail object which is a simple quantization $id(\mathbf{p})$ of its corresponding world space position $\mathbf{p} \in \mathbb{R}^3$. The *detail object cache* is a buffer storing the ID values of detail objects that have already been instantiated. When a frame is rendered, the detail object cache is cleared at the beginning. During the rendering of the procedural facades, each fragment shader evaluating an `AttachScope` operator first calculates the ID value $id(\mathbf{p})$ of its detail object. Subsequently, the fragment performs a lookup in the detail object cache. If the ID is present, the detail object has already been created and no further processing is necessary. Otherwise, the fragment stores the ID value of the detail object in the cache and then proceeds with writing its data to the attachment buffer atomically.

Following Cormen et al. [CLRS09], our detail object cache is realized as a hash table where collisions are resolved using double hashing. Given a detail object ID i, the position of its ID in the hash table after p probings is given by the primary hashing function $hash(i,p) = (i + p \cdot hash'(i)) \bmod H$, where $hash'$ is a secondary hashing function and H is the size of the detail object cache. For the secondary hashing function $hash'$ we use $hash'(i) = 1 + (i \bmod m)$, where $m < H$. The size H is chosen to be a prime number, thus guaranteeing that each probing sequence will have covered the entire hash table after H probings. As suggested by Cormen et al., we define m to be slightly smaller than H to maximize the number of different probing sequences.

Algorithm 1 describes how each fragment decides whether it is responsible for creating a detail object or not. For the maximal number H of probings that can be performed, the algorithm first samples at the hashed detail object ID pos and stores the result in a variable val (lines 1-4). If the value of val equals the detail object ID, the algorithm terminates with false, since the object has already been created by another pixel thread (line 5). Note that this simple check can be done asynchronously for each pixel for a fast rejection. If the value of val is 0, we have found a potential candidate that creates the detail object (line 6). In this case, we perform an atomic operation (*cacheAtomicCompareSwap*) on the cache at pos that writes the current ID only if the cache value is currently 0. The original cache value is returned and stored in a variable res (line 7). If the value of res is 0, we can safely return true to allow the detail object instantiation for the current pixel (line 8). If res equals the detail object ID, we terminate with false for the same reason as before, i.e. the object has already been created by another pixel thread (line 9). Otherwise, the algorithm has read another detail object ID from the cache and we have to perform the next probing.

Algorithm 1 Detail object caching

```
1:  $i \leftarrow id(\mathbf{p})$
2:  for $probe = 0$ to $H-1$ do
3:      $pos \leftarrow hash(i, probe)$
4:      $val \leftarrow cacheAt(pos)$
5:      if $val = i$ then return false
6:      if $val = 0$ then
7:          $res \leftarrow \text{cacheAtomicCompareSwap}(pos, 0, i)$
8:          if $res = 0$ then return true
9:          if $res = i$ then return false
10:     end if
11: end for
12: return false // failure
```

We read from the cache twice, since several pixels might have passed the fast rejection. With the atomic operation we can guarantee that only one pixel performs the detail object instantiation. However, atomic operations stall the GPU which decreases the rendering performance. Performing an asynchronous fast rejection mechanism in advance helps to avoid the slow atomic operations as often as possible.

5.3. Visibility Artifacts

Screen space techniques provide an automatic level of detail, however, they can be affected by certain visibility artifacts which have not been addressed so far. There are three major issues, namely *attachment scopes on back faces*, *attachment scope occlusion*, and *attachment scopes outside screen space* which result in unwanted popping artifacts, i.e., objects that suddently disappear although they should still be visible. The remainder of this section will present our solutions to prevent popping in most common situations.

5.3.1. Attachment Scopes on Back Faces

If a facade faces away from the viewer (e.g. at building corners), it is not drawn due to back face culling. Therefore, all detail objects that might protrude from the facade are not visible when the viewer is behind the facade plane, although they might be partially visible (cf. viewpoint 3 in Figure 3b).

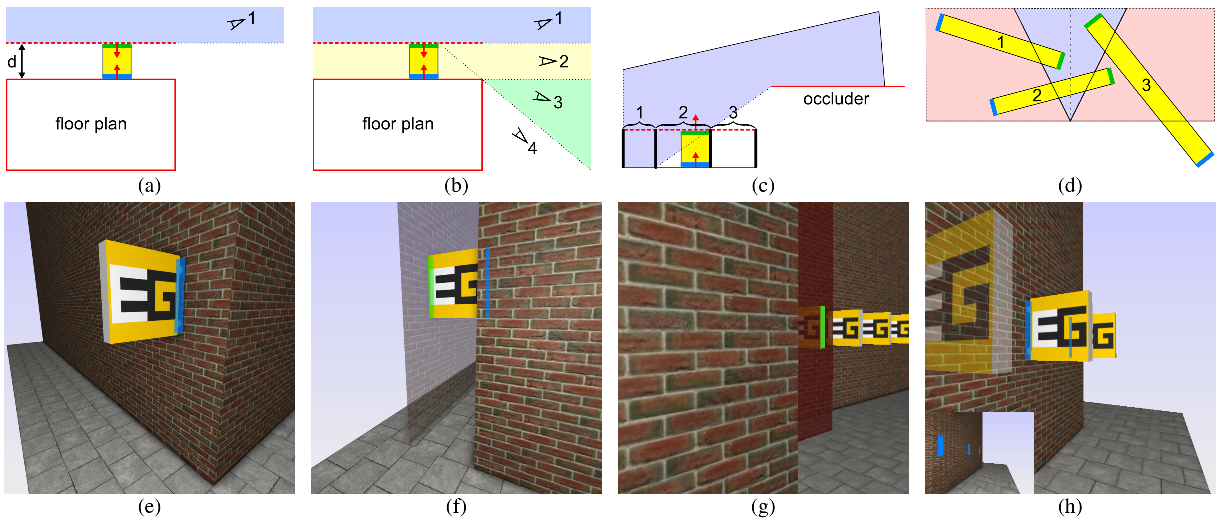


Figure 3: (a,c): Top-downs view showing the original polygons (red), auxiliary polygons (red, dashed), scopes (blue), auxiliary scopes (green), and the spanned bounding boxes (yellow). (a,e): Typical situation where the original scope is visible and the auxiliary polygon is not rendered due to back-face culling. (b,f): When moving around a convex building, either the original scope (1), the auxiliary scope (3), or both (2) are visible. Otherwise (4), the whole bounding box is occluded. (c,g): Common occlusions are handled by partially rendering a front facing auxiliary polygon (2). Most pixels of the auxiliary polygon are either not drawn due to the Z buffer (3) or discarded if their counterpart on the original polygon is visible (1). (d,h): The original field of view (blue) is extended to 180°. Hence, either the scope (1) or the auxiliary scope (2) is rendered, even if the original polygon is not in the viewport (3).

We solve this problem by rendering *auxiliary polygons* for all facades. Given a facade polygon, its auxiliary polygon is obtained by a translation offset d along its normal vector and inverting its orientation (cf. Figure 3a). Auxiliary polygons are rendered like ordinary facades with the exception that any detail object created is positioned as if it was created by the actual facade. Furthermore, auxiliary polygons discard any texture sampling as terminals and just take into account the attach operations such that the polygons themselves become invisible to the viewer.

The offset distance d is determined such that all bounding boxes of the detail objects fit between the original and the auxiliary facade plane. The value of d for a specific facade is easily obtained by a conservative estimate which is the maximum of all s_z parameters of `AttachScope` operators which are applied to that facade.

The presented strategy provides a solution for rendering detail objects on back facing facades while minimizing unnecessary evaluations of the facade shader. If the viewer's distance to a facade plane is greater than d (e.g. viewpoint 1 in Figure 3b), which is most often the case, the auxiliary polygon is never drawn due to back face culling. Furthermore, we can exploit the early Z testing to minimize fragment shader evaluations by rendering auxiliary polygons after all regular facade polygons. In the common scenario of building corners, only a narrow strip of additional fragments is generated for a visible auxiliary polygon (cf. Figure 3f).

5.3.2. Attachment Scope Occlusion

Attachment scopes on the original polygons might be occluded and thus, detail objects might not be instantiated, even if they protrude into the visible area (cf. Figure 3c).

Since the detail objects are contained within the bounding box that is spanned between the attachment scope on the original polygon and its counterpart on the auxiliary polygon, we could simply draw the auxiliary polygon from both sides, however, this would introduce an significant overdraw of the expensive grammar evaluation. On closer consideration, most attachment scopes are already visible on the original polygon such that the pixels from the auxiliary polygon are not needed. Inspired by shadow mapping, we discard all pixels from the auxiliary polygon, if the corresponding pixel on the original polygon is visible from the current viewport. Due to our Z prepass, this is easily implemented in the fragment shader. We first project the world position of the pixel back to the original polygon by the inverse offset $-d$. If the depth value at the original position is larger than the value stored in the Z buffer, i.e. the pixel on the original polygon is occluded, we have to evaluate the grammar. Otherwise, the pixel on the original polygon has been rendered already and we can skip the grammar evaluation. Thus, the overdraw is limited to slim stripes at building corners (cf. Figure 3g).

5.3.3. Attachment Scopes Outside Screen Space

A general problem with screen space methods is, that no information about objects outside the screen borders is available. This leads to the sudden disappearing of detail objects, as soon as their attachment scope leaves the screen.

We solve this issue by an additional off-screen render pass of the scene with an increased field of view. With a field of view of 180° degree, i.e. a degenerated view frustum spanning a whole half space, and a sufficiently high resolution we could guarantee, that the detail objects are instantiated properly, since either the attachment scope on the original facade or the attachment scope on the auxiliary polygon is visible (cf. Figure 3d). In practice, however, we have to cover the 180° with two view frustum matrices which introduces two additional passes. An empirical test of our rendering pipeline has shown that a slightly increased field of view (115° for rendered images with 90°) was sufficient and thus, we save one rendering pass.

Furthermore, it turned out, that the additional render pass only needs a very small resolution. Assuming, that the extent of detail objects is quite limited, they can only protrude into the visible area, if their corresponding attachment scope is quite close to the viewer's position. Consequently, the resolution of this off-screen pass can be very low, since the attachment scopes are close to the viewer and a single pixel already causes an object instantiation. In our experiments, a resolution of 160×120 pixels was sufficient for this purpose.

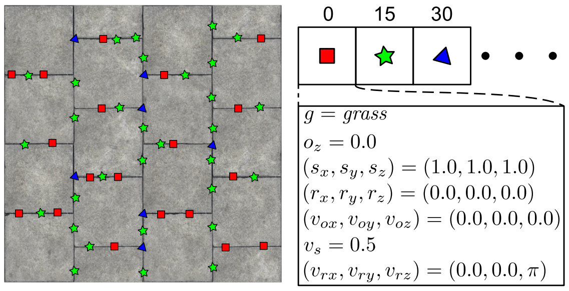
6. Attachment Textures

Instead of using the whole scope for the instantiation of a single detail object (as done by `AttachScope`), we allow for more artistic freedom by introducing the `AttachTexture` operator. Fine details like ivy, gravel stones, or moss have to be placed controlled but in a pseudo chaotic fashion, e.g. moss should only be placed in mortar regions of a brick texture, but the used geometry should be varied in size and orientation to hide the repetitive reuse of a single geometry.

Given any 2D texture, a corresponding *attachment texture* defines for certain pixels, if a detail object should be generated in this pixel during the rendering. Instead of simply using the detail object type in a pixel we refer to an *attribute group* (represented as a 1D *attribute texture*) which assigns a set of attributes to that pixel (cf. Figure 4).

An attribute group thereby defines a unique geometry and a transformation for a controlled instantiation of detail objects on a given texture. A pseudo chaotic appearance is then achieved by the definition of variances for the values.

The attachment texture is represented by a 2D texture where each color index c of a pixel signifies whether a detail object should be placed at that position and to which attribute group it belongs. We use the color index $c = 0$ to signify that no detail object is present. A color index of $c > 0$ refers to a position in the 1D attribute texture thereby assigning an



$g = grass$
$o_z = 0.0$
$(s_x, s_y, s_z) = (1.0, 1.0, 1.0)$
$(r_x, r_y, r_z) = (0.0, 0.0, 0.0)$
$(v_{ox}, v_{oy}, v_{oz}) = (0.0, 0.0, 0.0)$
$v_s = 0.5$
$(v_{rx}, v_{ry}, v_{rz}) = (0.0, 0.0, \pi)$

Figure 4: An artist can easily associate certain pixels with an attribute group which is implemented as a 1D texture. We use a set of 15 attributes per group to specify the geometry, transformations (normal offset, anisotropic scale, rotation), and variances (offset, isotropic scale, rotation).

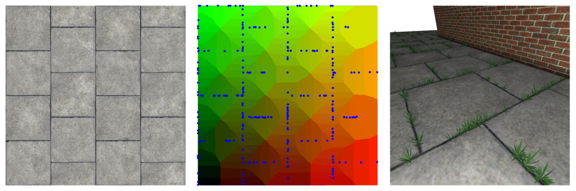


Figure 5: A boardwalk texture (a) is layered with an attachment texture placing tufts of grass on the gaps in the pavement. The attachment texture is turned into a Voronoi diagram texture (b) where the red and green color channels of each pixel represent the texture coordinates of the nearest attachment pixel in the attachment texture. The detailed grass geometry is only rendered in regions close to the viewer (c).

attribute group to that pixel. In practice, the attributes are packed into the color channels of a pixel (typically *RGBA*) to reduce the number of sampling operations that have to be performed during rendering. Hence, we only need 4 pixels in the 1D attribute texture to store our 15 attributes.

6.1. Detail Object Instantiation

In our grammar, the user can apply attachment textures by using the following operator:

$$\texttt{AttachTexture}(t, p, (m_1, m_2, m_3, m_4), f_{\text{near}}, f_{\text{far}})$$

The first parameter t identifies an attachment texture by a unique ID. Analogously to the texture sampling for terminals as proposed by Krecklau et al. [KK11], the 4-tuple $(m_1, m_2, m_3, m_4) \in \mathbb{R}^4$ specifies how the texture will be sampled with respect to the parameter $p \in \{Fit, Rep\}$. If $p = Fit$, the 4-tuple defines a rectangular region in the texture that will be fit into the current scope. Otherwise, if $p = Rep$, the first two parameters (m_1, m_2) define an offset whereas the second two parameters (m_3, m_4) define an anisotropic scaling to sample from a tileable texture that is repeated infinitely often. The fadeout distances f_{near} and f_{far} are defined similar like in the `AttachScope` operator.

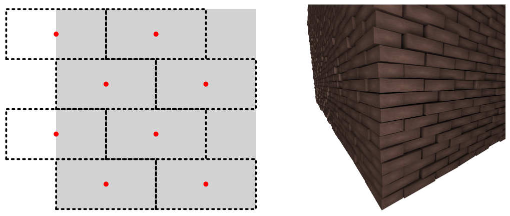


Figure 6: (a): Attachment texture for a brick wall. (b): During the rendering, irregularities are added by applying random variation to the offset and rotation of the bricks.

The AttachTexture operator defines multiple detail objects within the same scope. Our idea is that each fragment in the scope of an AttachTexture operator should try to create the detail object associated with the closest attachment pixel in the attachment texture. Finding the closest attachment pixel for each fragment corresponds to a subdivision of the scope into Voronoi cells. Each Voronoi cell then represents the area in which fragments try to produce one particular detail object. We pre-generate a *Voronoi diagram texture* for every attachment texture. Each pixel of this Voronoi diagram texture stores three values in its r,g,b channels, namely $c_x, c_y \in [0,1]$, the normalized texture coordinates of the closest attachment pixel, and $i \in \mathbb{N}$, the attribute group index of that attachment pixel (cf. Figure 5).

In contrast to the AttachScope operator where each sampled pixel during the evaluation of the grammar tries to instantiate the same detail object, the AttachTexture operator performs a lookup in the Voronoi diagram texture to retrieve all needed information, i.e. c_x and c_y to calculate the affine transformation and i to perform another lookup in the corresponding attribute texture to obtain the detail object type g. Note that all the transformation and their corresponding variances from the attribute group are taken into account when the affine transformation is calculated.

7. Results

7.1. Brickwork

As a first example, we enhanced the rendering of masonry by attaching real geometry for each individual brick (cf. Figure 6). We use the Repeat operators to generate a proper tiling such that the bricks interlock at the building corners:

```
   S ⇒ RepeatY(2,Row)
 Row ⇒ RepeatX(2,Tile)
Tile ⇒ AttachTexture(bricks,Fit,(0,0,1,1),5,10)
```

This creates tilings with an approximate absolute size of 2×2 units while lining up exactly with the facade borders. With the fadeout distances, the 3D brick geometry is only instantiated close to the viewer. Otherwise, their 2D projection is used. We improve the appearance by adding random variation to the transformation parameters of the bricks.

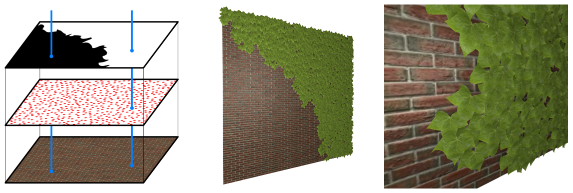


Figure 7: (a,b): A stack of layers with a SkipMask operator on top controls the distribution of ivy leaves. Where the mask texture is black, the underlying attachment texture is skipped and no leaves are instantiated. (c): Detail view.

7.2. Ivy

Facades overgrown with ivy leaves can be created in a similar fashion as brick walls. In addition to the previous example, we make use of the Layer and SkipMask operators as proposed by Krecklau et al. [KK11] to instantiate only a subset of detail objects from a given attachment texture (cf. Figure 7). Conceptually, these operations enable a convenient way to combine different structures in an artistic fashion, i.e. an artist can use a mask to combine layers on a per-pixel basis. Even though we have stated so far that both attach operators stop the evaluation, they actually only stop the evaluation in the current layer. Thus, if there are other layers on the stack, a further evaluation takes place. This is a necessary feature in most situations, since the loaded geometry does not need to occlude the whole scope such that the underlying facade is still partly visible:

```
      S ⇒ Layer(Wall,Ivy,IvyMask)
IvyMask ⇒ SkipMask      (mask, Fit,(0,0,1,1),0, 1)
    Ivy ⇒ AttachTexture(ivy,  Rep,(0,0,2,2),5,10)
   Wall ⇒ SampleTexture(brick,Rep,(0,0,2,2))
```

The first rule creates a stack of three layers, which is evaluated top-down (cf. Figure 7a). The topmost layer uses the IvyMask rule, so that the SkipMask operator is applied to decide whether the layer below should be evaluated (white region) or skipped (black region). Only for the white region, the Ivy rule is evaluated and attaches the detail geometry. Finally, the bottommost layer is evaluated which samples the color from a tileable brick texture. The mappings of the mask texture and the attachment texture of the ivy are independent, i.e. the macroscopic shape of the overgrowth, as defined by the mask, stretches to fit each facade (Fit) while the attach texture of the ivy is tileable and repeated (Rep). Thus, the ivy geometry is not stretched, but more instances are used.

8. Historical Facades

A common feature in European cities are residential buildings from the 19th and early 20th century with richly ornamented facades. The design of these facades is often dominated by a symmetric, axis-aligned layout of windows framed by decorative cornices and pilasters. This regular structure makes them especially suitable for a representation using procedural split grammars (cf. Figure 8).

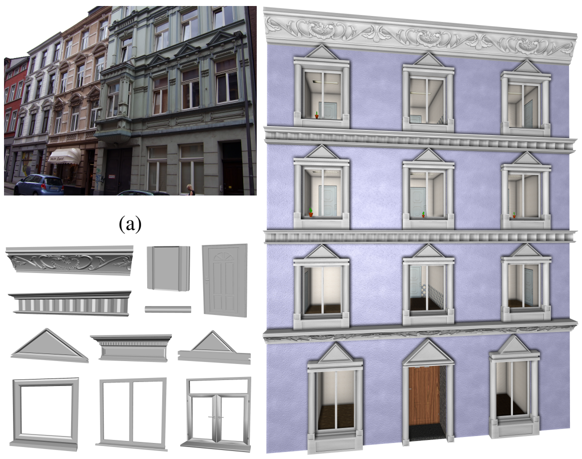


Figure 8: (a): Reference image of the facade style that we want to model. (b): Detail geometries that are used for the facades. (c): Example of a composed facade.

We use two main layers to reproduce the appearance of the facades. The lower layer creates the stucco on the facade which is sampled from a texture. On the upper layer, we create the layout of the doors and windows as a series of Split and Repeat operations. We first divide the facade into the first floor with an entrance doorway and the remaining upper floors which are of equal height. Cornices are placed between floors as separators. The remaining spaces in the first and upper floors are filled up with windows, which are framed by pilasters and decorative pediments. We randomly instantiate decorative elements to increase the overall variation. A snippet to compose a window is shown below:

```
     Window ⇒ SplitY((Abs,  1,Apron),
                     (Rel,  1,WindowMid),
                     (Abs,  1,Pilaster))
  WindowMid ⇒ SplitX((Abs, .5,Column),
                     (Rel,  1,WindowTile),
                     (Abs, .5,Column))
 WindowTile ⇒ Layer(Interior,WindowFrame)
WindowFrame ⇒ AttachScope(frame,    1,30,40)
   Pilaster ⇒ AttachScope(pilaster,1,30,40)
      Apron ⇒ AttachScope(apron,    1,30,40)
     Column ⇒ AttachScope(column,   1,30,40)
```

Like previous work, we use interior mapping for the rendering of the rooms behind the window frames [KK11, vD08].

9. Discussion

Performance We evaluate the performance of our renderer using the historical facade grammar on a large virtual city model. Statistics are taken on a GeForce GTX 470 rendering at a screen resolution of 1024×768 while showing the city model from four different viewpoints (cf. Figure 9,Table 1). Following the heuristics described in Cormen et al. [CLRS09], we use an attachment cache with a size of the prime number $H = 76801$. In the overhead view, the viewer is so far away from all facades that all visible attachment scopes are beyond their fadeout distance, thus there are no cache lookups or detail object instantiations. In the street and facade detail view, the number of visible detail objects is relatively low. This number raises quickly once a patch of ivy is viewed up close since the detail object cache fills up and the average length of the probing sequences increases.

Table 1: Statistics of our renderer showing the average frames per second (FPS), the number of placed detail objects (DO) and cache lookups (CL) as well as the average number of collisions (AC) for different views of a given city model (cf. Figure 9).

View	FPS	DO	CL	AC
Overhead view	44	0	0	n / a
Street view	55	514	245686	0.006
Facade detail	71	58	475167	0.0
Ivy detail	28	6937	610563	0.018

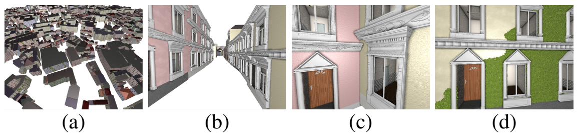


Figure 9: Different views of a city used to evaluate the rendering performance (cf. Table 1). (a): Overhead view. (b): Street view. (c): Facade detail. (d): Ivy detail.

Comparing the set of optimizations we apply for the rendering, i.e. the Z prepass, the early rejection by first reading from the cache asynchronously, and the indirect draw method to avoid downloading the buffers, our experiments imply the following conclusion. The average performance is measured for a fly-through in our city model including all four different viewpoints. Our implementation is seen as a reference with around 48 FPS in average. Without the indirect draw method, we still get ∼47 FPS, since the attachment counter buffer is so small that the download does not become a bottleneck. In contrast, the performance drops to ∼37 FPS without the Z prepass. Especially in street level views, a lot of expensive overdraw decreases the performance even to less than ∼21 FPS. The biggest performance boost emerges from the early rejection, i.e. if we just apply atomic operations the pipeline is stalled and drops to ∼17 FPS in average and even to ∼2 FPS in close up situations.

Raising the resolution to Full HD (1920×1080), our fully optimized implementation still has 28 FPS in average. Although the resolution is 2.6 times higher, the FPS in the low resolution is only 1.7 times higher. The reason for this is the fact, that the number of costy atomic operations stays the same. Due to the early rejection our technique scales well with increased resolution and even for higher resolutions our method is way faster than the current state-of-the-art (Marvie et al. had 7 FPS at 1280x720 with slightly faster hardware (GTX 480) [MGHS11]).

Visibility Artifacts When the viewer is looking onto a facade at a very sharp angle, it might happen that certain scopes become too narrow such that the attached geometry starts to flicker. The fadeout distances have to be chosen carefully, i.e. the texture should be used as soon as the corresponding scopes become too small. In future work, we want to elaborate on the automatic determination of appropriate parameters reducing the amount of manual tuning.

In very rare situations, the attachment scope on the original and the auxiliary polygon are occluded. For common city entities, like street lights, this is easily solved by rendering the additional entities in the second path. For the coarse city model, the artifact only becomes visible in very artificial situations, e.g. if the instanced geometry has a large extent and the viewer faces a narrow corridor such that both scopes are occluded (see accompanying video). Assuming a moving viewer a possible solution would be to cache the instantiated detail objects for a certain time interval.

For the attachment textures we currently use the Voronoi texture to obtain the closest attachment point which is eligible for pseudo-chaotic structures like grass or ivy. More structured patterns, like a brick wall, might better project the geometry into the attachment texture to assign the corresponding region. Obviously, both methods make sense and thus, we want to leave the choice to the artist in the future.

10. Future Work

Context Sensitivity One problem is the repetitive nature of split grammars resulting in a rather unrealistic impression for large scale cities. Based on the underlying texture, the procedural description could also place some kind of volumetric dirt proxies that influence the rendering of their surrounding which would enhance the overall integrity of the scene. We believe that weathering effects such as presented by Chen et al. [CXW*05] are easily visualized in this way.

Interactive Editing Since we are able to render large city models with detailed geometry, a modeling session can be performed directly in place such that the artist can always see the integration of the currently modeled item. This is especially important for the previously mentioned dirt proxies that actually rely on the respective context.

11. Conclusion

In this paper we show a rendering technique for the realtime visualization of highly detailed virtual city models. Using a very coarse polygonal scene of the city model, a procedural description is evaluated on a per-pixel basis to provide high quality renderings composed of architectural elements like windows, doors, ornaments, or wall structures. During this process, an intermediate buffer is filled with transformation matrices to render detailed geometry of the architectural elements on top of the previous image. Since our approach is a screen space technique, we automatically instantiate the geometric details only in regions that are currently visible to the viewer and thus get an inherent level-of-detail handling for free. By using the concept of attachment textures, we are even able to place geometric details in a pseudo-chaotic way that is not bound to the typically rectangular structure of a facade. Hence, our technique could be easily integrated into existing rendering engines that are not specifically targeted to city visualization in order to enrich the visual output by instantiating a lot of geometric items directly in screen space.

References

[CLRS09] Cormen T. H., Leiserson C. E., Rivest R. L., Stein C.: *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009. 5, 9

[CXW*05] Chen Y., Xia L., Wong T.-T., Tong X., Bao H., Guo B., Shum H.-Y.: Visual simulation of weathering by γ-ton tracing. In *SIGGRAPH* (NY, USA, 2005), ACM. 10

[dTWL08] de Toledo R., Wang B., Lévy B.: Geometry textures and applications. *CGF 27*, 8 (2008). 3

[GGH02] Gu X., Gortler S. J., Hoppe H.: Geometry images. *TOG 21*, 3 (2002). 3

[GLHL11] García I., Lefebvre S., Hornus S., Lasram A.: Coherent parallel hashing. *TOG 30*, 6 (2011). 3

[HWA*10] Haegler S., Wonka P., Arisona S. M., Gool L. V., Müller P.: Grammar-based encoding of facades. *CGF 29*, 4 (2010). 1, 3

[KK11] Krecklau L., Kobbelt L.: Realtime compositing of procedural facade textures on the gpu. In *3D-ARCH* (2011), ISPRS. 3, 7, 8, 9

[LWW08] Lipp M., Wonka P., Wimmer M.: Interactive visual editing of grammars for procedural architecture. In *SIGGRAPH* (NY, USA, 2008), ACM. 2

[MGHS11] Marvie J.-E., Gautron P., Hirtzlin P., Sourimant G.: Render-time procedural per-pixel geometry generation. In *Graphics Interface* (2011), Canadian Human-Computer Communications Society. 3, 9

[MWH*06] Müller P., Wonka P., Haegler S., Ulmer A., Gool L. V.: Procedural modeling of buildings. In *SIGGRAPH* (NY, USA, 2006), ACM. 2

[MWW12] Musialski P., Wimmer M., Wonka P.: Interactive coherence-based façade modeling. *CGF (Eurographics) 31*, 2pt3 (2012). 2

[MZWG07] Müller P., Zeng G., Wonka P., Gool L. V.: Image-based procedural modeling of facades. In *SIGGRAPH* (NY, USA, 2007), ACM. 2

[POC05] Policarpo F., Oliveira M. M., Comba J. L. D.: Real-time relief mapping on arbitrary polygonal surfaces. In *I3D* (NY, USA, 2005), ACM. 2, 3

[RLD*12] Reiner T., Lefebvre S., Diener L., García I., Jobard B., Dachsbacher C.: A runtime cache for interactive procedural modeling. *Computers & Graphics 36*, 5 (2012). 3

[Sti80] Stiny G.: Introduction to shape and shape grammars. *Environment and Planning B 7* (1980). 2

[vD08] van Dongen J.: Interior mapping - a new technique for rendering realistic buildings. *Computer Graphics International* (2008). 3, 9

[WWSR03] Wonka P., Wimmer M., Sillion F., Ribarsky W.: Instant architecture. In *SIGGRAPH* (NY, USA, 2003), ACM. 2