International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, Volume XXXVIII-5/W16, 2011
ISPRS Trento 2011 Workshop, 2-4 March 2011, Trento, Italy

REALTIME COMPOSITING OF PROCEDURAL FACADE TEXTURES ON THE GPU

Lars Krecklau and Leif Kobbelt

Institute of Computer Science 8, RWTH Aachen University
52074 Aachen, Germany
{krecklau|kobbelt}@cs.rwth-aachen.de

KEY WORDS: procedural modeling, shaders, real-time rendering, facades, city visualization

ABSTRACT:

The real time rendering of complex virtual city models has become more important in the last few years for many practical applications like realistic navigation or urban planning. For maximum rendering performance, the complexity of the geometry or textures can be reduced by decreasing the resolution until the data set can fully reside on the memory of the graphics card. This typically results in a low quality of the virtual city model. Alternatively, a streaming algorithm can load the high quality data set from the hard drive. However, this approach requires a large amount of persistent storage providing several gigabytes of static data. We present a system that uses a texture atlas containing atomic tiles like windows, doors or wall patterns, and that combines those elements on-the-fly directly on the graphics card. The presented approach benefits from a sophisticated randomization approach that produces lots of different facades while the grammar description itself remains small. By using a ray casting apporach, we are able to trace through transparent windows revealing procedurally generated rooms which further contributes to the realism of the rendering. The presented method enables real time rendering of city models with a high level of detail for facades while still relying on a small memory footprint.

1 INTRODUCTION

1.1 Motivation

Visualizing a highly detailed 3D city model in real time is a challenging task with lots of application scenarios in the industry like video games, 3D navigation or urban planning. The mass of data describing the model requires a large amount of storage that can grow beyond dozens of gigabytes, if the result has to be of a high quality. As a highly detailed polygonal model of each building is hart to obtain, a low resolution mesh using textures is typically generated from the footprints which are often freely accessible (OpenStreetMap, 2010). As long as real world facade images are available, those can be streamed directly to the graphics card using a mega texture approach (Mittring, 2008), however, high quality textures are rarely available and areal images are a poor alternative as they only provide a low resolution. Furthermore, converting the image data to proper textures is a challenging task itself due to varying lighting conditions or occlusions by obstacles like trees or traffic signs.

Procedural techniques, on the other hand, are capable to encapsulate the structure of a facade into a small set of rules which only consumes a few bytes. Assuming that the variety of windows, doors and decorations can be represented by a small set of texture assets, we are able to compose complex facade structures in real time on the GPU (cf. Figure 1). This approach enables a very high resolution which even allows realistic close-ups. We combine several procedurally generated layers on-the-fly to avoid the repetitive appearance of a facade which typically results from a rule based system.

We further enhance the visualization by tracing through the windows in order to render a variety of plausible rooms behind them. Obtaining this data from the real world would be impossible for at least two reasons. On the one hand, there are technical limitations, since several images of a single window from different viewing directions are needed to reconstruct the room behind. Especially for the upper floors it would be very hard to capture the floor of the rooms. On the other hand, there are privacy constraints as people usually do not want to have their rooms being captured for a virtual model. These privacy constraints even hold for detailed images of facades, because people are claiming about being spied by criminals. Therefore, systems like Google Street View (GoogleStreetView, 2010) have to blur certain areas to preserve the privacy. In this scenario, a procedural facade texture is a convenient alternative since it could reflect the semantic elements of a specific facade, but does not reveal the real details.

Figure 1: Rendering of a procedural facade texture applied to a single quad (upper image). Close ups of the facade demonstrate the high quality of our textures that even give the impression of 3D rooms behind the windows (middle row). A selection of atomic texture elements, which were used to compose this example facade, are shown in the bottom row.

Another advantage of using procedural facade textures on a given low resolution polygonal city model is the reuse of existing grammars. Unimportant residential areas can be textured randomly with different existing facade styles to ensure a plausible visualization without any additional cost.

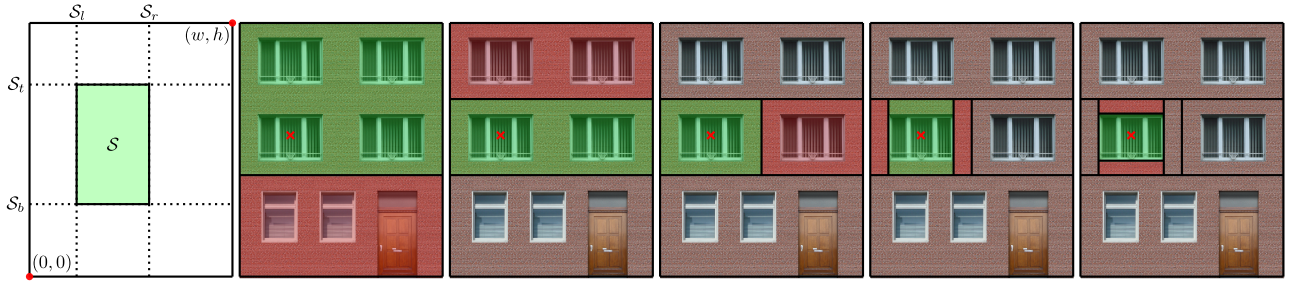

Figure 2: Taking the size of the whole facade as the start scope $\mathcal{S} = (0, 0, w, h)$, we recursively check for each operator of the grammar to which element a certain pixel of the final image belongs (shown in green). This will iteratively update the scope $\mathcal{S} = (\mathcal{S}_l, \mathcal{S}_b, \mathcal{S}_r, \mathcal{S}_t)$ until we hit a terminal symbol that samples a texture.

1.2 Related Work

A common approach for the rendering of large data sets, which do not fit into the system memory, is to stream the data from a permanent but slow data repository to the volatile but fast graphics card memory. One possible representation of the scenes are voxel octrees which scale very well and are easy to implement due to their regular structure (Laine and Karras, 2010, Crassin et al., 2009). If a coarse polygonal mesh is provided to reflect the most important features of a scene, a similar concept can be applied to stream textures (Mittring, 2008). In this paper, we also build upon on a low resolution polygonal mesh, but instead of using parts of huge and fixed texture we combine a wide variety of facade textures by combining different atomic elements like windows, doors or cornices.

In computer graphics, procedural approaches became well established for the automatic generation of plants. Lindenmayer and Prusinkiewicz introduced L-Systems as a formalism to describe the growth process of plants (Prusinkiewicz and Lindenmayer, 1996). L-Systems are basically simple string rewriting systems which produce a sequence of characters that are later on interpreted by a LOGO-style turtle for visualization purposes. The expressive power of L-Systems was also enhanced by using parameterized rules or stochastic variations during the evaluation of the grammar (Prusinkiewicz et al., 1997). Since street networks have a similar structure as plants, Parish et al. transfered the formalism of L-Systems to the modeling of cities (Parish and Müller, 2001) using the concept of self-sensitivity, i.e. if the turtle hits another line that was previously generated a crossing will be created. The output of their system is a large set of building lots which can be used to generate a low resolution polygonal mesh for artificial cities. Our system is then able of rendering the facade details on-the-fly as part of the rendering pipeline.

The most established procedural method for the generation of man-made structures like architecture is the decomposition of shapes into smaller shapes introducing more details in each step (Wonka et al., 2003). The concept of shape replacement instead of using a string rewriting system was first pioneered by Stiny presenting the idea of shape grammars (Stiny, 1975, Stiny, 1980). Larive et al. introduced a simple grammar system to produce 2.5D facades (Larive and Gaildrat, 2006) whereas Müller et al. applied the idea of shape grammars to the creation of more complex buildings by introducing the *CGA shape* modeling language (Müller et al., 2006). The basic modeling strategy of their appoach is to manually design a mass model which represents the coarse layout of the building. The details are then automatically generated by applying split rules to a scope, which is a bounding box containing a specific shape. The scope is associated with a certain rule containing operators that are applied to the scope like transformations or subdivision. Lipp et al. created an interactive version of this system to make procedural modeling accessible for artists without any programming knowledge (Lipp et al., 2008).

Krecklau et al. further enhanced the concept of *CGA shape* and introduced the unified procedural modeling language G^2 which handles multiple non-terminal classes in order to combine the philosophy of generating plants and archtiecture (Krecklau et al., 2010). There are several other approaches using expensive computations to produce realistic procedural models such as generating plausible courses of roads (Galin et al., 2010), physically stable masonry buildings (Whiting et al., 2009) or complex interconnected structures (Krecklau and Kobbelt, 2011). Unfortunately, all the previously mentioned methods are not feasible for content generation on-the-fly due to their complexity.

Recently, Haegler et al. presented *F-shade*, a real time approach for procedural facade textures that are evaluated directly on the graphics card using CUDA (Haegler et al., 2010). Their approach basically relies on a simplified instruction set of *CGA shape* that is capable of being evaluated on a per-pixel basis. Our work enhances their method by allowing a stochastic rule selection during the evaluation based on noise functions as it is typically done for generating terrains (Ebert et al., 2002). Furthermore, our system was inspired by the idea of interior mapping that enables the creation of procedurally generated 3D rooms that lie behind the windows (van Dongen, 2008). Each face of the room, i.e. the ceiling, the floor and the walls, is associated with a certain rule to produce a large set of unique rooms. In combination with a sophisticated randomization, a small rule set is already sufficient for the generation of a wide variety of different facades.

In contrast to the use of stochastic rules, real world images could be used to get unique semantic descriptions of every single facade as a compact representation (Müller et al., 2007). However, the retrieval of grammars from real world data is out of scope for this paper, since we concentrate on the efficient rendering of massive city scenes that are synthetically generated rather than reconstructing a real world city.

2 SYSTEM OVERVIEW

In general, a context-free formal grammar is defined as a 4-tuple $(\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{N}_{start})$, where $\mathcal{N}$ is a set of non-terminal symbols, $\mathcal{T}$ is a set of terminal symbols, P is a set of production rules of the form $\mathcal{N} \rightarrow (\mathcal{N} \cup \mathcal{T})^*$ and $\mathcal{N}_{start} \in \mathcal{N}$ is the start symbol. Whenever there is a non-terminal symbol that has a matching left-hand side in one of the production rules, that rule can be applied and the symbol will be replaced by the right-hand side of the rule.

Generating procedural facade textures on-the-fly relies on the idea of evaluating a grammar on a per-pixel basis. Similar to *F-shade* we use the term *scope* to denote a rectangle $\mathcal{S} = (\mathcal{S}_l, \mathcal{S}_b, \mathcal{S}_r, \mathcal{S}_t)$

GPU Representation													
Pixel Offset	0				1				···	m			
Channel	r	g	b	a	r	g	b	a	···	r	g	b	a
Repeat{X|Y}(v,R)	1|2	v	R^x	R^y									
Split{X|Y}(s_1,v_1,R_1,···,s_n,v_n,R_n)	3|4	n	R	A	$s_1 * v_1$	R_1^x	R_1^y		···	$s_n * v_n$	R_n^x	R_n^y	
Rand(l, v_1,R_1,···,v_n,R_n)	5	n	l		v_1	R_1^x	R_1^y		···	$\sum_{i=1}^{n} v_i$	R_n^x	R_n^y	
RandUni(l, R_1,···,R_n)	6	n	l		R_1^x	R_1^y	R_2^x	R_2^y	···	R_{n-1}^x	R_{n-1}^y	R_n^x	R_n^y
RandStack(R)	7	R^x	R^y										
Layer(R_1,···,R_n)	8	n			R_1^x	R_1^y	R_2^x	R_2^y	···	R_{n-1}^x	R_{n-1}^y	R_n^x	R_n^y
Skip(l)	9	l											
SkipMask(m,l_0,l_1)	10	m	l_0	l_1									
Intrusion(v,R_1,R_2,R_3,R_4,R_5)	11	v	R_1^x	R_1^y	R_2^x	R_2^y	R_3^x	R_3^y		R_4^x	R_4^y	R_5^x	R_5^y
Mat(u,m_1,···,m_5)	12	u	m_1		m_2	m_3	m_4	m_5					
MatAlpha(u,m_1,···,m_5,a_1,···,a_5)	13	u	m_1	a_1	m_2	m_3	m_4	m_5		a_2	a_3	a_4	a_5

Figure 3: Definition of operators provided by our system (cf. Sec. 3) and their representation in a texture on the GPU (cf. Sec. 4).

that is associated with a certain rule for the evaluation of the grammar (cf. Figure 2). For simplicity, we will address the bottom left corner of the scope with $\mathcal{S}_{lb} = (\mathcal{S}_l, \mathcal{S}_b)$ and the top right corner with $\mathcal{S}_{rt} = (\mathcal{S}_r, \mathcal{S}_t)$. Given the absolute size $s = (w, h)$ of the facade defining the start scope $\mathcal{S}_s = (0, 0, w, h)$ and the texture coordinates $t = (u \in [0, w], v \in [0, h])$ at the current pixel position, the grammar has to determine the terminal operator that lies at position t. The traversal of the grammar is dependent on the operators that are visited during the evaluation. Figure 2 depicts a simple technique to generate the details using subdivision along the local x and y axes, however, the advanced traversal of the grammar using a stochastical rule selection, multiple layers, or 3D rooms will be explained in Section 3.

Evaluation of the grammar per pixel is quite expensive since it can be compared to a ray tracer involving many conditions to determine a specific cell within an irregularly subdivided space. Therefore, we apply a deferred rendering technique that first uses the hardware accelerated rasterization to render the basic information that initializes a ray tracer per pixel. The details on storing the grammar in an efficient way on the GPU and the single steps of our rendering pipeline are described in Section 4.

3 GRAMMAR FEATURES

The syntax of our grammar is a simplified version of *CGA shape* balancing the expressive power and the complexity of the formalism in order to provide a real-time evaluation per pixel. Similar to *F-shade*, our rules are just mappings from a string to a specific operator with fixed parameter values, which are formally defined as follows:

$$\texttt{RuleName} \rightarrow \texttt{OperatorName(Parameter}_1\texttt{, } \cdots\texttt{, Parameter}_n\texttt{);}$$

A parameter is either a floating point value or the name of a rule to further proceed the resulting elements. Although it might sound very restrictive to use non-parametric rules, we will show in our discussion that this is not a limiting factor at all. However, stochastic variation is an important design issue for procedural facades in order to avoid the repetitive nature of rule based facades. In contrast to *F-shade*, we provide two operators that rely on a noise function to generate facade textures with randomized details and another operator that puts the current random seed on a stack. Recalling a random seed from the stack allows for the creation of random groups, i.e. all elements within a group are handled in the same way.

We heavily enrich the visual quality of our facades by tracing a ray through the windows in order to render a rectangular shaped room that lies behind it. Basically, we were inspired by the idea of interior mapping (van Dongen, 2008), but due to our grammar approach we are able to design a large set of different rooms by evaluating our grammar on the sides of the room.

Similar to *F-shade*, we allow the creation of several procedurally generated layers to easily compose background structures as walls with foreground details as windows, doors or cornices. By combining different backgrounds and foregrounds, the textures become rich in variety due to the combinatorial complexity. In contrast to *F-shade*, we present another composition philosophy to reduce the evaluation overhead for facade textures with complex subdivision patterns.

3.1 Operators

Our system provides several types of operators for the arrangement and stochastic variation of elements, the use of layers, the ray casting of rooms and the assignment of color values to the corresponding pixel (cf. Figure 3). The set of non-terminal symbols that are currently defined in the grammar is addressed as $\mathcal{N}$ in the remainder of this section. The scope to which an operator is applied will be addressed as $\mathcal{S}$, where $\mathcal{S}_w$ and $\mathcal{S}_h$ are its width and height, respectively.

For the arrangement of elements, we basically apply the *Split* and *Repeat* operators of *CGA shape*. In contrast to *F-shade*, we make use of relative and absolute parts in the definition of the *Split* operator as it is done in *CGA shape* (cf. Figure 4). This makes the operator more dynamic for differently sized facades, e.g. if we have several houses with a different height that should all have a ground floor with a height of 3 meters. Formally, the *Split* operator takes an arbitrary number n of 3-tuples $(s_i \in \{-1, 1\}, v_i \in \mathbb{R}_{>0}, R_i \in \mathcal{N})$ for $i \in \{1, \cdots, n\}$. The first parameter s_i defines the size policy, i.e. $abs \mathrel{\widehat{=}} -1$ for a part with an absolute size and $rel \mathrel{\widehat{=}} 1$ for a part with a relative size. The second parameter v_i defines the size value whereas the third parameter R_i specifies the target rule that is applied to the new element. The actual width w_i of an element i using the split operator along the x axis is calculated according to equation 1.

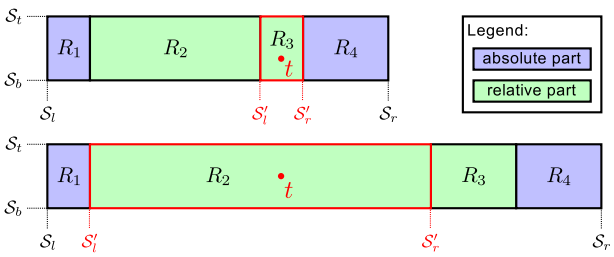

Figure 4: The *Split* operator subdivides the current scope along a certain axis. An absolute size policy will always produce an element of a fixed size (blue) whereas a relative size policy will calculate the actual size depending on the defined ratios between all relative elements (green). The element that will be further evaluated by the grammar is determined by successively checking which interval of the split is hit by the texture coordinates t.

$$w_i = \begin{cases} v_i, & s_i = -1 \\ \frac{\mathcal{S}_w + \sum_{i=1}^{n} \min\{s_i * v_i, 0\}}{\sum_{i=1}^{n} \max\{s_i * v_i, 0\}} * v_i, & s_i = 1 \end{cases} \quad (1)$$

The texture coordinates $t = (u \in [0, w], v \in [0, h])$ at the current pixel position are checked against the resulting intevals in order to determine the scope and the rule for the next iteration of the grammar evaluation.

The *Repeat* operator uses two parameters, i.e. $v \in \mathbb{R}_{>0}$ and $R \in \mathcal{N}$, to subdivide the scope into elements that all have an approximate size of v (cf. Figure 5). In detail, the actual width w of all elements that are generated by a subdivision along the x axis is $\mathcal{S}_w / \lceil \mathcal{S}_w / v \rceil$. By using the texture coordinates $t = (u \in [0, w], v \in [0, h])$ we directly calculate the scope that will be further processed by rule R.

For the stochastic variation, we introduce two operators, namely *Rand* for a non-uniform distribution and *RandUni* for a uniform distribution. We make this distinction, because in most cases we only need a uniform distribution which can be optimized to reduce the evaluation time. Formally, the operator *Rand* takes an arbitrary number n of 2-tuples $(v_i \in (0, 1], R_i \in \mathcal{N})$ for $i \in \{1, \cdots, n\}$ with $\sum_{i=1}^{n} v_i = 1$. During the evaluation, we take a random number $r \in [0, 1)$ and calculate the accumulated values $a_i = \sum_{j=1}^{i} v_j$ for $i \in \{1, \cdots, n\}$. Defining $a_0 = 0$, we can easily determine the target rule R_k such that $r \in [a_{k-1}, a_k)$. For the operator *RandUni*, which just takes a list of n target rules $R_i \in \mathcal{N}$ for $i \in \{1, \cdots, n\}$, we can directly calculate the parameter $k = \lfloor r * n \rfloor + 1$ using a random number $r \in [0, 1)$ and proceed with the target rule R_k.

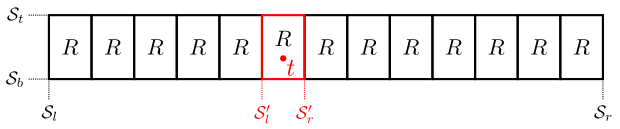

Figure 5: The *Repeat* operator applies a subdivision into elements of the same size. This enables us to directly calculate the scope which is hit by the texture coordinates t.

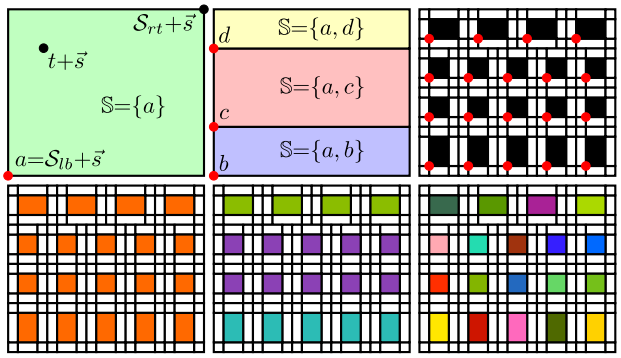

Figure 6: In the upper row, we show the application of several split rules. During the subdivision, we put several random seeds (a, b, c, d) onto the stack $\mathbb{S}$. In the bottom row, we apply the *Rand* operator to all black scopes that are shown in the upper right facade. In this example, each target rule that can be chosen by *Rand* will apply an operator which fills the whole scope with a unique color. Using 2 (bottom left facade) or 1 (bottom middle facade) as first parameter will group the elements according to the random seed on the current stack. If the first parameter of *Rand* is 0, every element uses another target rule since the bottom left corner varies from scope to scope (bottom right facade).

Each facade is initialized with a random seed vector $\vec{s} = (s_x, s_y)$ which is encoded into the start scope $\mathcal{S}_s$ by applying a translation, i.e. $\mathcal{S}_{lb} = (0, 0) + \vec{s}$ and $\mathcal{S}_{rt} = s + \vec{s}$ where $s = (w, h)$ is the full size of the facade. The texture coordinates t also have to be updated according to $\vec{s}$ in order to get the same relative position within the scope. By convention, we generate the random value r by sampling a noise texture at position $\mathcal{S}_{lb}$. All subdivision operators will thereby automatically create scopes that calculate different random values r, however, there are several situation in which a previously calculated random seed has to be used in order to create random groups, e.g. if a rule randomly selects a window shape, but all windows in a specific floor should get the same shape. This could be solved by copying the whole structure and only exchanging the rule that generates the specific window shape in each copy. Instead, we introduce the concept of putting the random seed on a stack $\mathbb{S}$. The operator *RandStack* just proceeds with the target rule $R \in \mathcal{N}$ after the bottom left corner of the current scope has been put onto the stack, i.e. $\mathbb{S} = \mathbb{S} \cup \mathcal{S}_{lb}$. The first parameter $l \in \mathbb{N}$ in both randomization operators therefore defines that the $l-th$ last element that was put into $\mathbb{S}$ will be used as the random seed. If $l = 0$, we will just use $\mathcal{S}_{lb}$ from the current scope. Figure 6 depicts our modeling strategy by utilizing random groups as an elegant solution to control the stochastic variation.

Another concept to enhance the visual quality of the facade texture is the use of layers. The *Layer* operator creates n overlayed copies of the current scope and associates a certain target rule R_i for $i \in \{1, \cdots, n\}$ with each copy. Similar to a volume ray caster (Crassin et al., 2009), we proceed the evaluation with the upper most layer, i.e. R_n, and trace through the layers summing up any color, normal or specular values according to the alpha value that holds in each layer. Since this is an iterative approach, we can stop the evaluation if we have reached an alpha value of 1.0 (cf. Figure 7). Therefore, we save computation time, since we do not necessarily traverse all layers. Formally, we update the final color value C and alpha value A by adding the color value c_i and alpha value a_i of the current layer i in the following way:

$$C \quad += \quad (1.0 - A) * (a_i * c_i) \quad (2)$$
$$A \quad += \quad (1.0 - A) * a_i \quad (3)$$

Layers can be cascaded, i.e. the grammar contains another *Layer* operator within any layer. The scope $\mathcal{S}$ and the texture coordinates t of every layer i are translated by $i * \vec{s}$ in order to produce new random values in each layer.

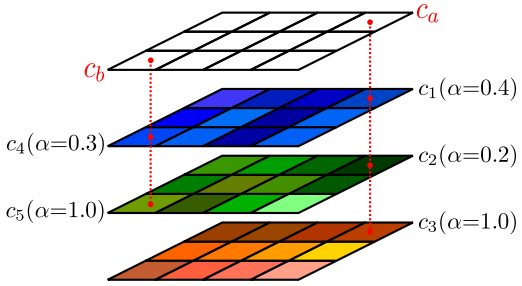

Figure 7: The *Layer* operator produces several layers that are independently evaluated. Once a terminal symbol is found, i.e. a texture is sampled, we check, if the alpha value of that pixel is already 1.0 representing a solid color. In this case, the evaluation of the grammar stops and returns the corresponding color. If the alpha value is smaller 1.0, we have to further evaluate the underlying layers to calculate the final pixel color.

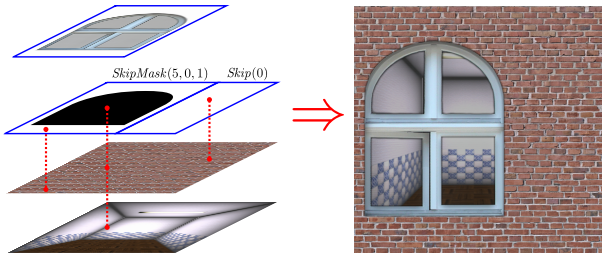

Figure 8: Skipping of underlying layers is an intuitive way for cutting holes into the wall in order to see the room behind. The *SkipMask* operator decides how many layers are skipped depending on the sampled bit value whereas the *Skip* operator skips the defined number of layers for the whole scope. In the example, we create another layer over the hole that samples a window texture to produce the final tile that is shown on the right side.

An important design issue for the creation of facade textures is the idea of masking to skip certain layers (cf. Figure 8). The operator *SkipMask* selects a bit mask $m \in \mathbb{N}_{>0}$ from the texture array. Depending on the pixel value of the bit mask, i.e. 0 or 1, we skip $l_0 \in \mathbb{N}$ or $l_1 \in \mathbb{N}$ layers, respectively. This concept enables an easy way of defining holes in a wall to see the room behind it. In detail, we just create a layer for the wall, the rooms and the bit mask that decides for each pixel, if we actually see the wall or the room behind it. For simplicity and a higher performance during the evaluation, we also provide an operator *Skip* that just skips $l \in \mathbb{N}_{\geq 0}$ layers for the whole scope.

Rooms or intruded windows are created with a similar principle as the component operator of *CGA shape*. The *Intrusion* operator takes an intrusion amount $v \in \mathbb{R}_{>0}$ as first parameter followed by 5 target rules $R_1, \cdots, R_5 \in \mathcal{N}$ for the three walls, the floor and the ceiling. The viewing ray $\vec{r}$ that hits the scope at the texture coordinates t is used to intersect with a box that is spanned by the current scope and the intrusion amount v. The intersected face becomes the new scope using the intersection point t' as new texture coordinates. All operators can now be applied as before acting on one of the faces (cf. Figure 9). By transforming $\vec{r}$ into the space of the new scope, e.g. a 90° rotation along the x axis if the bottom face is hit, it is even possible to recursively apply intrusions. Note, that the scopes $\mathcal{S}'_i$ and the texture coordinates t'_i of every face i, which is associated with rule R_i, are translated by $\mathcal{S}_{lb} + i * \vec{s}$ in order to determine new random values on each face.

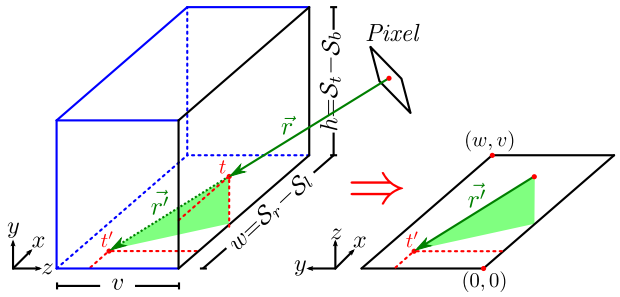

Figure 9: By tracing along the viewing ray that hits the current scope at the texture coordinates t we determine the hit face of a box that lies behind the scope. This yields a new scope, e.g. $\mathcal{S} = (0, 0, w, v)$ for the bottom face where $w = \mathcal{S}_r - \mathcal{S}_l$ and v is the intrusion amount. The texture coordinates t' are now defined with respect to the new scope. This enables us to further evaluate the grammar on any face within the box as before.

As a final step in the grammar evaluation, we can either sample a texture array or define a constant value to influence the color, the normal or the specularity of the material. All channels can be combined with an alpha map to change the strength of the influence. The first parameter of the *Mat* operator defines its usage $u \in \{1, 2, 3\}$, i.e. $color \mathrel{\widehat{=}} 1$, $normal \mathrel{\widehat{=}} 2$ or $specular \mathrel{\widehat{=}} 3$. The other 5 parameters $m_1 \in \mathbb{Z}$ and $m_2, \cdots, m_5 \in [0.0, 1.0]$ are now dependent on each other in the following way. If $m_1 = 0$, we only use the parameters $m_2, \cdots, m_5$ to define a constant color. If $m_1 > 0$, m_1 selects a texture from the array and $m_2, \cdots, m_5$ define a rectangular part from the texture that should be scaled to fit the current scope. If $m_1 < 0$, $-m_1$ selects a texture from the array, m_2, m_3 define an offset and m_4, m_5 define a scaling to apply a sampling from a seamlessly tileable texture that is repeated infinitely often. The *MatAlpha* operator works analogously, except that it multiplies the resulting value from the first sampling with a certain factor that is taken from another texture sampling which is defined by the last 5 parameters. If the alpha value is smaller 1.0 the grammar has to evaluate the layer below the current one (cf. Figure 7).

4 GPU REPRESENTATION

All grammars are stored in one large 2 dimensional 4 channel 16 bit float texture to represent the operators and their parameters. Figure 3 gives an overview of all operators and their corresponding layout within the texture. Each rule has a unique position $R = (R^x, R^y)$ within this texture, so that any operator that needs to further proceed its resulting elements (e.g. *Repeat*) has to occupy two channels to define the corresponding position of the target rule. The first channel of each operator location is always a constant that specifies, which operator code has to be executed. Most operators like *Repeat* or *Mat* have a fixed amount p of parameters resulting in a fixed number $t = \lceil p/4 \rceil$ of texture lookups to retrieve the parameter values. Note, that we always retrieve 4 parameters for one texture lookup since we utilize a 4 channel texture. Some operators have a dynamic amount of parameters, so that the number of texture lookups depends on the algorithm that is executed. The maximal number n of subsequent texture lookups is always stored in the grammar as a static termination criteria for that operator. The *Split* operator, e.g., has to check successively, if a pixel lies in the current split interval or if we have to retrieve the subsequent parameter tuple that defines the next split interval. Since this operator also relies on the global sums R and A of all relative and absolute values, respectively, we precalculate these sums and store the results directly in the grammar. Therefore, we do not necessarily have to lookup all parameter tuples, if the current pixel lies in one of the first split intervals. The same is true for the *Rand* operator where we can stop its execution once we have found a tuple with an accumulted sum that is higher than the random value r, which is taken from a noise texture. Since all pixels in the same scope have to retrieve the same random value r, we sample the noise texture at $\mathcal{S}_{lb}$ which is the lower left corner of the current scope. In most situations, a uniform distribution of target rules is sufficient which can be achieved with the *RandUni* operator. In this case, we directly calculate the offset in the grammar that contains the target rule to reduce the number of texture lookups. Note, that the chosen layout of the grammar texture may waste a small amount of channels in some situations, e.g. the alpha channel of each *Split* or *Rand* tuple, to avoid complicated branchings in the code that would result in a lower evaluation performance.

For an efficient evaluation of the grammar at real-time rates we apply a deferred shading technique. Seen from a certain camera perspective, several polygons will overlap resulting in multiple

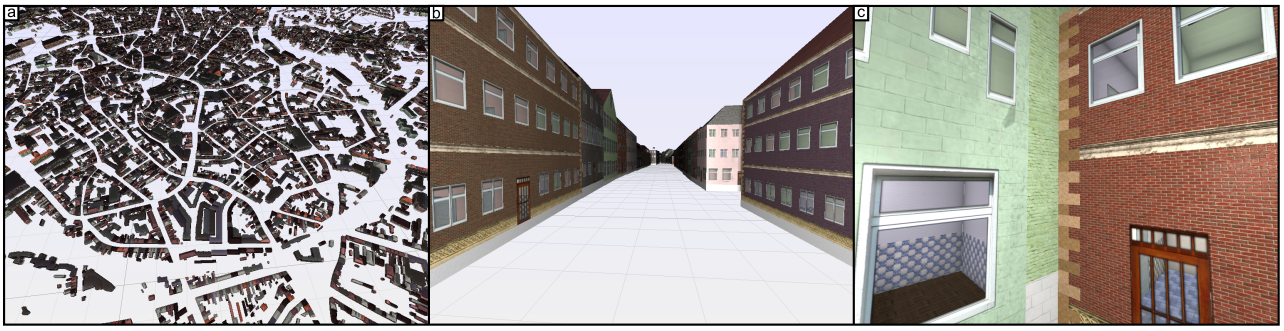

Figure 10: This image depicts the full power of our system using a texture set with only 32 elements. The evaluated grammars contain all presented operators including randomization, several layers, masking and intrusion. For our time statistics (cf. Figure 11), we have chosen different views, i.e. distance (a), street view (b) and close-up (c).

executions of a fragment shader for a single pixel. Therefore, we first apply a computational cheap fragment program that renders the low resolution polygonal mesh of the city model and only stores basic information for each pixel. Afterwards, we reuse the given information in order to execute the computational expensive shaders per pixel including the grammar evaluation, screen space ambient occlusion and phong lighting.

The fragment shader for the grammar evaluation utilizes the texture coordinates, the scope size of the complete facade, the start rule, the random seed, the normal and the tangent. Therefore, the low resolution polygonal mesh also has to provide this information as attributes in the vertex buffer object (VBO). Except for the texture coordinates all the information can be shared among the vertices of a triangle, however, for technical reasons we have to attach this information redundantly for each vertex in the VBO. The texture coordinates represent the absolute corner positions of the facade which are interpolated to yield the actual position inside the facade for a single pixel. Since we are able to apply a normal map with the material operator, we create the corresponding tangent space coordinate system using the normal and the tangent. The texture coordinates, scope size, start rule and random seed then initialize the grammar evaluation which yields the final color of the current pixel.

5 RESULTS

We have applied our method to render a virtual city model containing 20 thousand buildings. We created different scenarios to measure the frames per second depending on the grammar complexity and the camera view. The average timings using a NVIDIA GeForce 470 GTX GPU and a resolution of 1024×768 are shown in Figure 11. Using simple grammars (cf. Figure 13.a), that do not utilize randomization or intrusion, our system has a comparable rendering performance to *F-shade*. For the creation

	Simple	Random	Full
Distance	79 / 12	66 / 15	18 / 55
Street view	130 / 7	110 / 9	39 / 25
Close-up	82 / 12	70 / 14	25 / 40

Figure 11: This table contains some time statistics (frames per second / ms per frame) of our system with respect to the different views of Figure 10. In the simple version, we just apply several subdivisions to each facade utilizing a static set of textures as atomic elements (cf. Figure 13.a). For the random version, we additionally apply randomization operators to get a wide variety of combined walls and windows (cf. Figure 13.b). In the full version, we take advantage of all presented concepts including randomization, layers, masking and intrusion (cf. Figure 10).

```
S->Layer(Wall, Assets)
Wall->RandUni(0, Wall1, Wall2, Wall3, ···)
Wall1->Mat(Color, Rep:wall1, 0.0, 0.0, 1.0, 1.0)
Wall2->Mat(Color, Rep:wall2, 0.0, 0.0, 1.0, 1.0)
Wall3->Mat(Color, Rep:wall3, 0.0, 0.0, 1.0, 1.0)
⋮
Assets->SplitY(Abs, 3.0, BottomFloor,
               Rel, 1.0, Middle,
               Abs, 2.5, TopFloor)
BottomFloor->RandStack(BottomFloorSeedStored)
BottomFloorSeedStored->RepeatX(2.5, WindowTile)
Middle->RandStack(MiddleSeedStored)
MiddleSeedStored->RepeatY(2.8, MiddleFloor)
MiddleFloor->RepeatX(2, WindowTile)
TopFloor->RandStack(TopFloorSeedStored)
TopFloorSeedStored->RepeatX(3, WindowTile)
WindowTile->SplitX(Rel, 1.0, Skip,
                   Abs, 1.0, WindowTileColumn,
                   Rel, 1.0, Skip)
WindowTileColumn->SplitY(Rel, 1.0, Skip,
                         Abs, 1.3, WindowMat,
                         Rel, 1.0, Skip)
WindowMat->RandUni(1, WindowMat1, WindowMat2, WindowMat3, ···)
WindowMat1->Mat(Color, Fit:window1, 0.0, 0.0, 1.0, 1.0)
WindowMat2->Mat(Color, Fit:window2, 0.0, 0.0, 1.0, 1.0)
WindowMat3->Mat(Color, Fit:window3, 0.0, 0.0, 1.0, 1.0)
⋮
Skip->Skip(0)
```

Figure 12: This example grammar was used for one of our performance tests using a stochastic rule selection. Possible outcomes of this particular grammar can be seen in Figure 13.b. The generated structure is similar to the one depicted in Figure 6. Therefore, the chosen color coding of the rules corresponds to the scope colors in Figure 6. Constants like *Color*, *Fit*, *Abs* or the texture names are automatically mapped when the grammar is compiled.

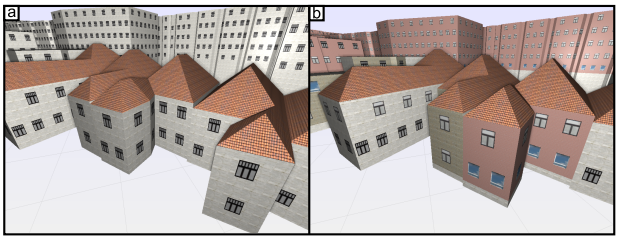

Figure 13: This image shows simple facade grammars containing only subdivisions (a) or subdivisions with randomization (b).

of a wide variety of facades while still relying on a small set of rules, we added randomization operators that contribute to the visual quality (cf. Figure 13.b), but only slightly influence the rendered frames per second. An example grammar that makes use of the randomization operator is shown in Figure 12. If we apply a complex grammar to the city model that contains all presented concepts like several layers or intrusion, our system still renders the scene at interactive frame rates (cf. Figure 11). Especially if we use a camera view that is placed on street level, which is most commonly the case for our applications, we still get frame rates of around 40 frames per second.

6 DISCUSSION

Memory Consumption — The grammars are all stored in a single texture. With a very pessimistic estimate, that an operator occupies 3 pixels on average and that each facade is encoded into 85 rules, we get approximately 256 pixels that are needed for this particular grammar. With a texture that has a resolution of 4096×4096, we are able to store 65536 unique facade descriptions consuming 128 megabytes of graphics memory. In practice, most of the rules, e.g. for generating the rooms, are randomly reused throughout the grammar which typically results in facade grammars that occupy a lot less than 256 pixels. The complete grammar of Figure 10, e.g., only contains 330 rules that occupy 741 pixels in the grammar texture consuming 5928 bytes. Due to the random variations this is already sufficient to generate a wide variety of textures for the entire city model.

However, since graphics cards even handle textures of a large size, there is no need to introduce a more complex evaluation system on the GPU that can handle parametric rules. Instead, rule sets can simply be copied to enable the instantiation of a single grammar with varying parameter values. In future work, the copying of rules could be done by a precompiler such that we get the freedom of using parametric rules without touching the evaluation system.

For the low resolution polygonal mesh we need to store several information in the VBO summing up to 68 bytes per vertex assuming 4 bytes for one float number. The sum of 68 bytes results from the vertex position (12 bytes), the texture coordinates (8 bytes), the scope size of the complete facade (8 bytes), the start rule (8 bytes), the random seed (8 bytes), the normal (12 bytes) and the tangent (12 bytes). The Aachen model, e.g., has around 1.8 million vertices which yields an overall memory consumption of around 120 megabytes. Most of the graphics card memory is needed for the texture array holding the atomic elements like windows, doors or decorations. Depending on the available memory, the resolution of the texture elements can be adjusted. In future work, the system could be also combined with a streaming approach providing all textures in different resolutions in order to choose the appropriate ones for the current view.

Usability — Currently, the rules have to be written by hand which can be compared to writing a little script. We provide namespaces so simplify the reuse of a certain grammar, i.e. rules are just copied into another namespace in order to vary certain parameters. Previous work has shown that the grammar design can also be done interactively (Lipp et al., 2008). With a similar approach, it would be possible to apply modifications to a single facade in real time while thousands of buildings are still visualized in the background.

Limitations — The presented method relies on the idea of sudividing a rectangular scope along a certain axis. While this approach works well for the creation of facades, organic structures like trees can not be handled in this way as they do not result from a subdivision of a bounding volume but from a growth process that starts in a single point. In this case, the whole grammar has to be evaluated in advance to determine the final geometry.

Future Work — Beside the other enhancements that were already mentioned throughout the paper, we would like to increase the visual quality of our visualization by applying some kind of relief mapping (Nguyen, 2007) for the atomic elements. Especially elements with a large depth, like ornaments or cornices, would benefit from this method. Evaluating a grammar on-the-fly causes aliasing artifacts, because it is a discrete decision into which element a certain pixel falls, i.e. the subdivision always returns one element for a certain pixel, although the pixel area might cover several split intervals. The errors could be prevented by an adaptive supersampling approach that samples more often around splits, however, more sophisticated methods require additional research on that topic.

7 CONCLUSION

This paper presents a method of generating procedural facade textures for high quality city visualizations on the GPU. The deferred rendering pipeline first renders important information to each pixel which initializes a fragment shader for the grammar evaluation on a per-pixel basis. By providing several layers of procedurally generated textures, a bit mask to compose these layers and a ray tracing approach to intersect with a box behind the current scope, we are able to randomly generate a variety of rooms behind the windows to enhance the realism of the visualization. Since the facade textures are generated on-the-fly they can be easily reused in most parts of the city model without enlarging the memory footprint. The balanced combination of rasterization and ray tracing as a hybrid rendering solution provides a high potential for future work where the unique geometry is stored as a low polygonal mesh and the details are generated on-the-fly without any restrictions in resolution.

ACKNOWLEDGEMENTS

This work was supported in part by NRW State within the B-IT Research School and by the DFG Cluster of Excellence on Ultra-high Speed Mobile Information and Communication (UMIC), German Research Foundation grant DFG EXC 89.

REFERENCES

Crassin, C., Neyret, F., Lefebvre, S. and Eisemann, E., 2009. Gigavoxels: ray-guided streaming for efficient and detailed voxel rendering. In: Proceedings of the 2009 symposium on Interactive 3D graphics and games, I3D '09, ACM, NY, USA, pp. 15–22.

Ebert, D. S., Musgrave, K. F., Peachey, D., Perlin, K. and Worley, S., 2002. Texturing & Modeling: A Procedural Approach, Third Edition. Morgan Kaufmann.

Galin, E., Peytavie, A., Maréchal, N. and Guérin, E., 2010. Procedural generation of roads. Computer Graphics Forum 29(2), pp. 429–438.

GoogleStreetView, 2010. http://maps.google.com.

Haegler, S., Wonka, P., Arisona, S. M., Gool, L. J. V. and Müller, P., 2010. Grammar-based encoding of facades. Computer Graphics Forum 29(4), pp. 1479–1487.

Krecklau, L. and Kobbelt, L., 2011. Procedural modeling of interconnected structures. Computer Graphics Forum.

Krecklau, L., Pavic, D. and Kobbelt, L., 2010. Generalized use of non-terminal symbols for procedural modeling. Computer Graphics Forum 29, pp. 2291–2303.

Laine, S. and Karras, T., 2010. Efficient sparse voxel octrees. In: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games, I3D '10, ACM, NY, USA, pp. 55–63.

Larive, M. and Gaildrat, V., 2006. Wall grammar for building generation. In: GRAPHITE '06, ACM, NY, USA, pp. 429–437.

Lipp, M., Wonka, P. and Wimmer, M., 2008. Interactive visual editing of grammars for procedural architecture. In: SIGGRAPH '08, ACM, NY, USA, pp. 1–10.

Mittring, M., 2008. Advanced virtual texture topics. In: SIGGRAPH 2008 classes, ACM, NY, USA, pp. 23–51.

Müller, P., Wonka, P., Haegler, S., Ulmer, A. and Gool, L. V., 2006. Procedural modeling of buildings. ACM TOG 25(3), pp. 614–623.

Müller, P., Zeng, G., Wonka, P. and Gool, L. V., 2007. Image-based procedural modeling of facades. ACM TOG 26(3), pp. 85.

Nguyen, H., 2007. GPU Gems 3. Addison-Wesley Professional. Chapter 18 - Relaxed Cone Stepping for Relief Mapping.

OpenStreetMap, 2010. http://www.openstreetmap.org.

Parish, Y. I. H. and Müller, P., 2001. Procedural modeling of cities. In: SIGGRAPH '01, ACM Press, NY, USA, pp. 301–308.

Prusinkiewicz, P. and Lindenmayer, A., 1996. The algorithmic beauty of plants. Springer-Verlag New York, Inc., NY, USA.

Prusinkiewicz, P., Hammel, M., Hanan, J. and Měch, R., 1997. Visual models of plant development. Springer-Verlag New York, Inc., NY, USA.

Stiny, G., 1975. Pictorial and Formal Aspects of Shape and Shape Grammars. Birkhauser Verlag, Basel.

Stiny, G., 1980. Introduction to shape and shape grammars. Environment and Planning B 7, pp. 343–361.

van Dongen, J., 2008. Interior mapping - a new technique for rendering realistic buildings. Computer Graphics International 2008.

Whiting, E., Ochsendorf, J. and Durand, F., 2009. Procedural modeling of structurally-sound masonry buildings. ACM TOG 28(5), pp. 112.

Wonka, P., Wimmer, M., Sillion, F. and Ribarsky, W., 2003. Instant architecture. ACM TOG 22(3), pp. 669–677.