DOI: 10.1111/j.1467-8659.2012.03201.x
Pacific Graphics 2012
C. Bregler, P. Sander, and M. Wimmer
(Guest Editors)

Volume 31 (2012), Number 7

GPU Shape Grammars

Jean-Eudes Marvie Cyprien Buron Pascal Gautron Patrice Hirtzlin Gaël Sourimant

Technicolor

Figure 1: *We introduce a novel approach to GPU-based interactive procedural generation of the constitutive elements of large-scale environments. This scene comprising 116K buildings and 561K trees is edited, generated and rendered at 7-12fps.*

Abstract

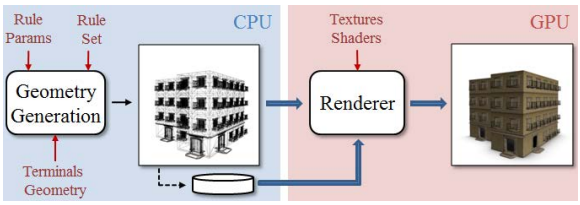
GPU Shape Grammars provide a solution for interactive procedural generation, tuning and visualization of massive environment elements for both video games and production rendering. Our technique generates detailed models without explicit geometry storage. To this end we reformulate the grammar expansion for generation of detailed models at the tesselation control and geometry shader stages. Using the geometry generation capabilities of modern graphics hardware, our technique generated massive, highly detailed models. GPU Shape Grammars integrate within a scalable framework by introducing automatic generation of levels of detail at reduced cost. We apply our solution for interactive generation and rendering of scenes containing thousands of buildings and trees.

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

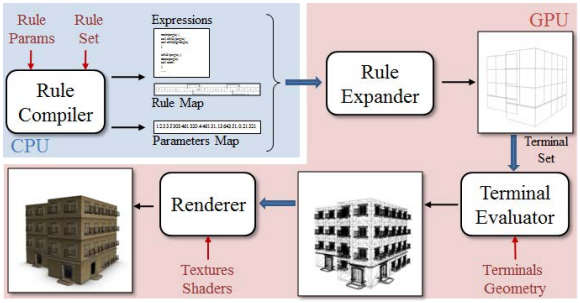
1. Introduction

Modeling the massively detailed environments used in current video games and films requires intensive artistic input. These complex geometries can sometimes be handled in real-time, to the extent of the available graphics memory. However, in movie post-production the creativity is often restrained by the lack of interactive feedback. The elements of complex scenes are often modeled independently, and assembled using rough representations to preserve interactivity. Specific preprocessings and out-of-core schemes are then required for visualization.

Procedural modeling reduces user interventions by exploiting the repetitive patterns typically present in buildings, cities and organic shapes. Instead of explicitly modeling the scene elements, the artist selects a procedure describing the construction rules for a family of objects. Using a collection of elementary (possibly high definition) shapes provided by the artist, the geometry of entire families of objects is generated automatically. However, modifying a single generation parameter may require a costly regeneration of the entire object. This issue scales with the size of the environments: cities or forests comprise many objects generated from a small set of construction rules. Modifying a rule then involves a full generation and storage of all the related models (Figure 2a), resulting in delays in the design workflow. Furthermore, the memory occupancy of many fully detailed objects quickly rises to prohibitive levels.



(a) Classical generation pipeline



(b) GPU Shape Grammars

Figure 2: *GPU Shape Grammars map procedural modeling techniques to the requirements of graphics hardware. We compile the grammars into an efficient structure for fast expansion of the rules. The expansion generates a lightweight intermediate representation of the object structures. The geometry is then generated on the fly without explicit storage.*

We take a new approach to real-time procedural modeling and rendering of complex environment elements, avoiding explicit geometry storage (Figure 2b). Our method expands construction rules at run time using hardware geometry generation. Its easy integration brings the advantages of procedural modeling within the reach of real-time engines.

In Section 3 we introduce *GPU Shape Grammars* and discuss a generic pipelined architecture for the generation of highly detailed models (Sections 4 to 6). We apply our approach to buildings and vegetation (Section 7) and propose a full-featured scheme for dynamic generation of seamless levels of detail for interactive editing and rendering of massive environments (Sections 8 and 9).

2. Technical Background

This section focuses on previous work most related to our contributions. A thorough state of the art analysis can typically be found in [WMWF07].

Prusinkiewicz and Lindenmayer [PL90] describe the use of L-Systems [Lin68] for vegetation modeling. Based on observations of real plants, the growth of each species is described by a grammar. Families of distinct plants can then be obtained by expanding the grammar using a small set of parameters describing each individual. Levels of details can also be generated [LCV03].

Procedural modeling also finds a particular use for architectural modeling [WWSR03, MPB05]. In most approaches [MWH*06, MGHS11] the high-level structure of the buildings is modeled using a set of rules and parameters, while artists provide the shapes of the constitutive elements of the facades. Each facade is then a set of assembled elements. While effective, the evaluation cost of the grammar rules might prevent interactive editing and rendering of complex models. Also, the full geometry of the generated sceneries often has to be stored prior to rendering.

The approach of Lipp *et al.* [LWW10] is based on an intermediate representation of the grammars after a derivation step. This representation expresses a generic expansion for a fixed number of iterations. Then the interpretation stage traverses it to generate the geometry. While this technique exhibits parallelism for derivation and interpretation of multiple L-System grammars, it requires full regeneration when changing the number of iterations, and storage of the intermediate structure. Conversely, our approach considers unconstrained numbers of iterations for any object, trading code optimization for generic solution, while leveraging parallelism.

Another approach to avoid the explicit storage of expanded geometry is to delay the generation until the rendering stage. Haegler *et al.* [HWA*10] and Marvie *et al.* [MGHS11] introduce specific grammar representations for fast generation. For each pixel these techniques lazily perform the grammar derivation for each visible façade element. In [HWA*10] facades are assembled from simple 2D textures representing the elements, with no conditional and stochastic rule support. The approach from Marvie *et al.* [MGHS11] is based on more general rules and ray tracing of geometry images [GGH02] representing façade elements, providing higher visual quality. These techniques effectively reduce the generation time and provide interactive performance even with massive models. However, they are heavily fragment-bound and do not support component split operations.

3. Procedural Pipeline

Existing techniques are usually based on iterative geometry refinements. In counterpart, the efficiency of the graphics pipeline comes from a highly parallel, stage-based structure carrying specialized information. We then reformulate grammar expansion for efficient procedural generation on graphics hardware. Based on Chomsky grammars [Cho65], our solution supports both growth and reduction operations and is usable for most purposes of procedural modeling. As shown in Figure 2 our approach is composed of a rule compiler, a rule expander and a terminal evaluator.

Rule Compiler This CPU-based stage extracts a generic expansion graph from the rules and converts it into a interpretable *rule map*. The run-time behavior of the rules is extracted and combined with a generic *rule map* interpreter,

yielding a grammar-specific GPU *expression-instantiated interpreter*. The grammar parameters are packed into a *parameter map* for fast access.

Rule Expander Our GPU interpreter traverses the rule map according to the input parameters. The output of the traversal is a lightweight set of terminal symbols describing the object structure.

Terminal Evaluator This stage performs GPU-based geometry generation based on the terminal set. The evaluator fetches the geometric description of each terminal and generates terminal geometry. The geometry is then directly rendered without storage.

4. Rule Compiler

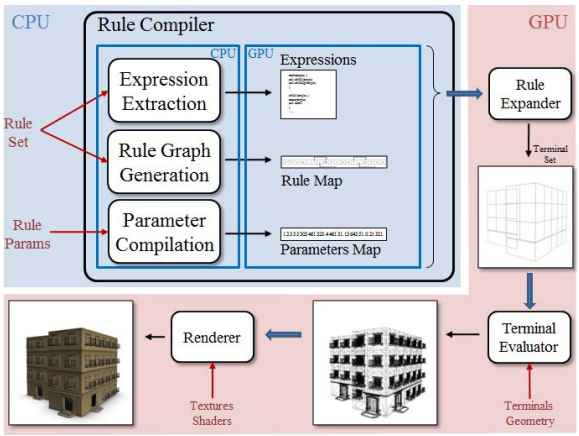


Figure 3: *The rule compiler divides the grammar into a rule map and a set of shader-based expressions, resulting in an expression-instantiated rule map interpreter.*

Any grammar rule can be written in the spirit of [MWH*06]:

$$\text{Pred} \rightsquigarrow \text{Rule}(\{Expr_j(P)\}_{j\in\mathbb{N}})\{\text{Succ}\} \qquad (1)$$

where *Pred* is the rule predecessor, *Rule* is the name of one of the supported built-in rules, and *Succ* is the set of rule successors. The behavior of the expressions $Expr_j$ is driven by the generation parameters $P = \{p_i\}_{i\in\mathbb{N}}$. We replace the rule condition of [MWH*06] by the specific rule type Cond.

Let us consider a simple recursive growth grammar (Figure 4). Using this formulation we represent a rule using four components: a predecessor, a successor set, a rule type and a set of expressions. All the possible expansions of a grammar can then be represented by a *rule graph* linking each predecessor to all its successors through a rule type identifier and a set of expressions associated to each rule.

We flatten the rule graph into a *rule map* where the successors are represented by offsets and the rule type is a simple

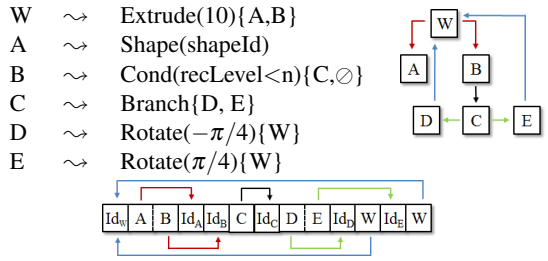


Figure 4: *A simple growth grammar (left), where W is the axiom. We represent the rules using a rule graph (right), encoded into a* rule map *(bottom).*

identifier. The expressions typically involve low-level, GPU-friendly operations. We extract these expressions from the grammar and generate an expression library in shader code for run-time evaluation.

Finally, we instantiate a stack-based generic interpreter for rule map traversal. The purpose of this interpreter is to evaluate the expressions and to apply the current rule to determine the successor set. The successors are then recursively processed until all the rules have been expanded into terminal symbols. This interpreter is statically combined with the grammar-specific expression library. This results in a high performance expression-instantiated interpreter for the input grammar, ready for execution in the rule expander (see next section).

5. On The Fly Rule Expansion

We carry out the geometry generation in two main steps: the rule expander outputs a set of independent terminals, for which the terminal evaluator generates the geometric details.

5.1. Rule Expander

The grammar and generation parameters fully describe the structure of the target object, although not in a directly renderable form. The rule expander makes intensive use of hardware geometry generation to generate a set of simple primitives associated with each terminal symbol (Figure 5).

The operations described in [MWH*06] operate on elements of various dimensions from points to volumes, where the component split operation breaks elements into elements of smaller dimension. The underlying data-parallel structure of graphics processors cannot handle such dimension changes without a significant overhead. In counterpart, this architecture favors repetitive operations performed in parallel on objects of constant dimension.

This problem is naturally solved by a simple principle: An object of dimension n can always be decomposed into elements of lower dimensions [Edm60, Lie94]. As 3D surfacic objects are composed of 0D (vertices), 1D (segments) and

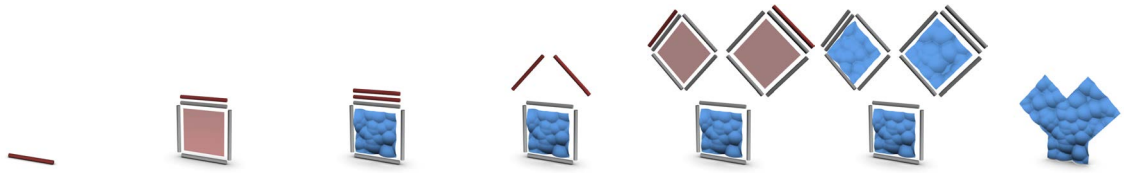


Figure 6: *Exploded view of the expansion of our example grammar. The axiom is applied on a base 1D atom (a), yielding an extruded face composed of one 2D and four 1D atoms (ie. an expansion element), and a translated 1D atom (b). The expansion element is replaced by a terminal shape, while the translated 1D atom is split (c) and rotated (d). Each branch is then extruded (e) and the new expansion elements are replaced by shapes (f). The assembly of shapes is the final geometry (g).*

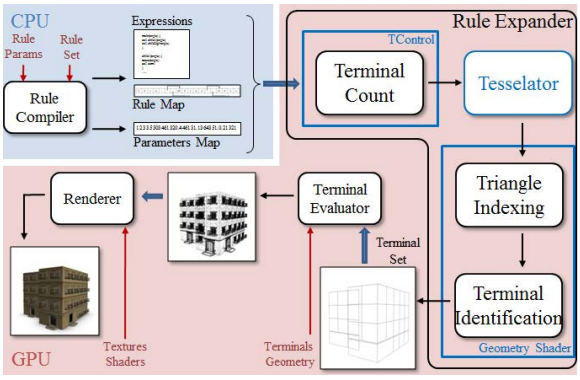


Figure 5: *The rule expander uses the tessellation control shader to expand the grammar according to the input parameters, generating a list of parameterized terminal symbols. The list size is then passed to the tesselator which generates the appropriate number of triangles. Finally, the geometry shader associates each triangle with a terminal and outputs a lightweight terminal set.*

2D (surfaces) atoms, geometric operations can be expressed as a combination of segments with potential 2D elements. For example, the extrusion a surface starts with a decomposition into segments followed by an extrusion of each segment. A surface split simply requires a decomposition in two sets of segments. Using this principle we introduce a grammar expansion method based on 1D atoms.

Expansion Context Each rule expander thread uses a context representing the local frame of the current 1D atom and a tag indicating whether the atom is part of a surface. Current contexts are managed using fixed-size stacks in GPU memory.

Segment-Based Expansion Starting with a set of input atoms we initialize an *expansion context* representing their local frames. The axiom W of the grammar is then applied, generating a set of 1D and 2D atoms (Figure 6) and updating the expansion context. Subsequent rules are then applied recursively until all paths reach terminal rules. This formulation has a direct impact on the rule representation: even when applied on 1D atoms, a notion of the higher dimensions must remain. This principle is illustrated by the extrusion rule of Figure 4. Considering a single input atom (Figure 6a), the output of this rule is twofold: First, the generated face is described by a 2D atom and a set of contour 1D atoms (Figure 6b), called expansion element. Second, the rule translates the input 1D atom by the extrusion vector $\mathbf{v}$. More formally, the extrusion is:

$$\text{Pred} \rightsquigarrow \text{Extrude}(\mathbf{v})\{\text{F\_Succ}, \text{S\_Succ}\} \tag{2}$$

where F_Succ and S_Succ are respectively applied to the generated expansion element and the translated 1D atom or previously generated expansion element. This formulation is an implicit component split operation. In Figure 4 no specific rule is applied to the face: We formalize this by introducing the concept of a *null terminal* stopping the expansion.

We introduce a *branch* rule which replicates the original atom n times (Figure 6c) and applies a successor to each generated atom (Figure 4):

$$\text{Pred} \rightsquigarrow \text{Branch}(n)\{\ \{\text{Succ}_i\ \}_{i=1\ldots n}\} \tag{3}$$

The current 1D atom can be rotated using a *rotation* rule (Figure 6d), updating the expansion context (Figure 6e):

$$\text{Pred} \rightsquigarrow \text{Rotate}(q)\{\text{Succ}\} \tag{4}$$

Finally, the Shape rule associates an expansion element with its corresponding detailed geometry:

$$\text{Pred} \rightsquigarrow \text{Shape}(geometry) \tag{5}$$

This rule discards the input atoms, and outputs a 2D surface which is converted into a terminal geometry in later stages.

Using these principle the example grammar (Figure 4) can be expanded as shown in Figure 6. Our reformulation is carried out automatically from the input grammar, making this process entirely transparent to the user.

Implementation on Graphics Hardware We implemented our technique using hardware tesselation and Shader Model 5.0 (DirectX 11). While Cuda/OpenCL languages could also be used, the specific tesselation units of the Nvidia GTX 480

processor could not be directly accessed. Our shader-based implementation then harnesses the computational power of both the computation cores and tesselation units.

Starting from input 1D atoms, the tessellation control shader executes our expression-instantiated interpreter. The resulting list of terminal symbols and associated parameters are stored within a simple read/write buffer in graphics memory (using shader image load store mechanisms). The list size is then passed to the tessellator which generates an appropriate number of triangles (up to 8192 on GTX 480).

As the output of the tessellator is a simple triangles soup we identify triangles using the barycentric coordinates of their vertices. The corresponding terminal symbols and parameters are then fetched from the previously generated list. The final output of the geometry shader is a lightweight set of 2D patches representing the placement and parameters of the terminals. This terminal set can either be streamed to the next stage for direct geometry generation and rendering, or cached to avoid per-frame regeneration of static objects.

5.2. Terminal Evaluator

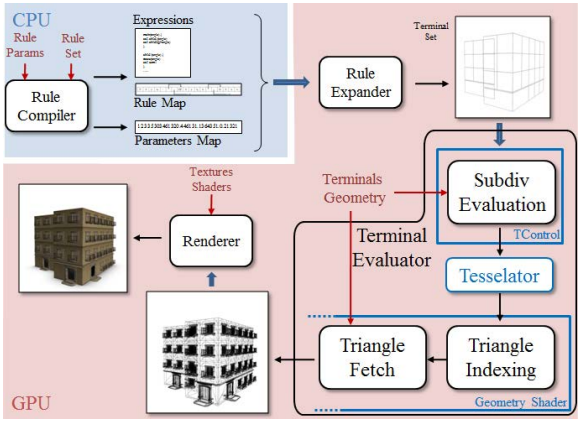


Figure 7: *The terminal shapes are substituted to the terminal set using tessellation and on the fly geometry evaluation.*

The terminal set contains the location and parameters for the terminal shapes. However, the structure of graphics hardware does not allow a direct substitution of the terminal set by the corresponding detailed geometry. Instead, we embed the terminal shapes within GPU buffers.

Each primitive of the terminal structure is then tessellated into the number of triangles corresponding to the target detailed terminal geometry. The geometry shader then extracts the corresponding shape from the geometry buffers (Figure 7). Note that other representation such as texture-guided subdivision surfaces could also be used.

The generated geometry is finally rendered within the same render pass, avoiding explicit storage of the detailed model.

6. Renderer

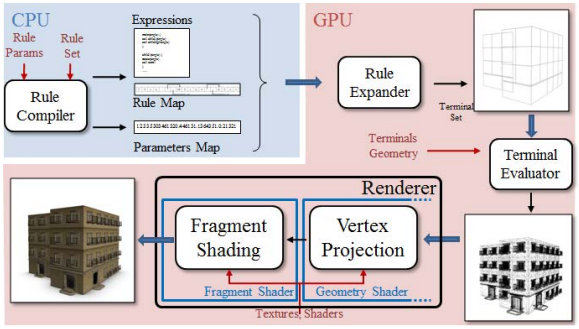


Figure 8: *Our pipeline generates a set of classical meshes ready for rendering using any existing shading technique.*

The output of our procedural generation pipeline is a simple set of textured triangle meshes (Figure 8). Procedurally generated objects can therefore be combined to other scene components within a same render pass, making our method easily integrable into existing rendering engines. For the purpose of production rendering the output of the terminal evaluator can also be stored within GPU buffers and read back to files for later rendering.

7. Applications

7.1. Architecture

Starting from a set of footprint segments, grammar expansion generally begins with a small number of growth operations generating the overall building shapes. Numerous reduction operations (*e.g.* split) then divide each facade into elements such as doors and windows. A simple example of such grammar is provided below using a syntax following our formulation based on 1D atoms:

$$
\begin{array}{lcl}
W & \rightsquigarrow & \text{Extrude(buildingHeight)}\{F, \oslash\} \\
F & \rightsquigarrow & \text{Split("y", floorHeight, } \sim)\{GF, FL_R\} \\
GF & \rightsquigarrow & \text{Split("y", floorHeight-0.1, 0.1)}\{G, T^L\} \\
G & \rightsquigarrow & \text{Split("x", doorWidth, } \sim)\{T^D, GW_R\} \\
GW_R & \rightsquigarrow & \text{Repeat("x", windowWidth)}\{T^W\} \\
FL_R & \rightsquigarrow & \text{Repeat("y", floorHeight)}\{FW_R\} \\
FW_R & \rightsquigarrow & \text{Repeat("x", windowWidth)}\{T^W\} \\
T^L & \rightsquigarrow & \text{Shape(ledge)} \\
T^D & \rightsquigarrow & \text{Shape(door)} \\
T^W & \rightsquigarrow & \text{Shape(window)}
\end{array}
$$

where rules F, GF, G and FL_R represent the facade, ground floor including top ledge, ground floor elements and the other floors. T^L, T^D and T^W are the terminal symbols of the grammar, linking to the shapes of the ledge, door and windows.

The split and repeat operations [MWH*06] can be easily implemented in shader code. The terminal shapes can be arbitrarily chosen and encoded into GPU buffers, yielding a wide range of building appearances (Figure 9).

Figure 9: *Variety of buildings following our sample grammar. Changing the geometry of the terminal symbols allows for further style variations.*

7.2. Vegetation

Compared to architecture, plant evolution induces numerous recursive growth operations, keeping reductions marginal. This expansion scheme seamlessly fits within our pipeline, which supports extrusion and implicit component split.

The tree model[†] shown in Figure 10 is generated from the grammar on page 60 of [PL90], adapted to our framework:

W	$\rightsquigarrow$	Extrude($trnkLen$,$twist_0$){T^T, Grow }
Grow	$\rightsquigarrow$	Cond($rec < n$){ BranchSplit, T^B }
BranchSplit	$\rightsquigarrow$	Branch(3){ Main, Sub1, Sub2}
Main	$\rightsquigarrow$	Rotate($q1$){ RotBranch }
RotBranch	$\rightsquigarrow$	Extrude($brchLen$,$twist_1$){T^B, Leaves}
Leaves	$\rightsquigarrow$	Branch(2){Leaf, Grow}
Leaf	$\rightsquigarrow$	Rotate(q_2){ T^L }
Sub1	$\rightsquigarrow$	Rotate(q_3){ Grow }
Sub2	$\rightsquigarrow$	Rotate(q_4){ Grow }
T^T	$\rightsquigarrow$	Shape(trunkGeom)
T^B	$\rightsquigarrow$	Shape(branchGeom)
T^L	$\rightsquigarrow$	CreateQuad(leafVertices)

This grammar follows the principle of Figure 6, including recursive rules simulating the growth of smaller branches from the trunk. This recursion is represented within the rule graph and evaluated for any level at run-time by our expression-instantiated interpreter. Each tree is based on a single input 1D atom along with a parameters map.

8. Scaling Up

We introduce object batching and LOD management for interactive generation and rendering of massive environments.

8.1. Object Batching

Starting with 1D atom contexts and a set of parameters our pipeline generates a terminal set describing the location and parameters of each element. We leverage the parallel architecture of graphics hardware by grouping the 1D atom contexts into a single buffer, and packing parameters into a *parameters map*. The terminal set for an entire set of objects with various parameters is then generated using a single

[†] Terminal geometry and textures courtesy of `http://xfrog.com/`

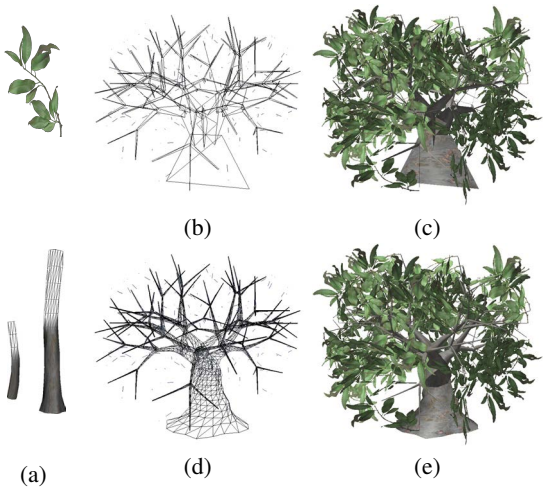


Figure 10: *Starting with an input 1D atom and a small set of terminal shapes (a), the grammar is expanded into 240 terminal patches (b), on which textures can be mapped to generate a coarse LOD (c). Those terminals are then replaced by their corresponding geometry, yielding a fully detailed model (d,e) generated from end to end in 15.3ms (64.8fps). Once the terminal set is cached render speed reaches 530fps.*

draw call (Figure 11a). The evaluation and rendering of the terminal shapes is performed by rendering the entire terminal set. Consequently, multiple objects can be generated and rendered at the same cost as a single object (Figure 11b).

8.2. LOD Generation

Our technique for dynamic generation of levels of detail (LOD) comes from this observation: starting from a basic representation, each generation step creates additional details. Lower LODs can then be obtained by truncating the generation process. We devise 5 LODs for buildings: LOD 4 is fully detailed, while levels 3 and 2 only require a terminal set. LOD 0 and 1 only use the base geometry of the objects, *ie* the result of the first growth rule (Figure 12).

The higher LODs substitute geometry by image-based impostors [JWP05]. A classical approach consists in generating an impostor per object. Instead, as procedural models reuse a limited number of terminal shapes we generate and assemble per-terminal impostors. Those impostors are generated automatically by GPU evaluation and rendering of terminal shapes simultaneously from several views. Using this principle, LOD3 preserves parallax through view interpolation, while more distant objects only use the center view (LOD2). Those LODs use the same terminal set and can be inexpensively chosen and blended for each terminal.

Lower LODs avoid the generation of a terminal set. Instead, we generate a simplified *extruded set* representing the

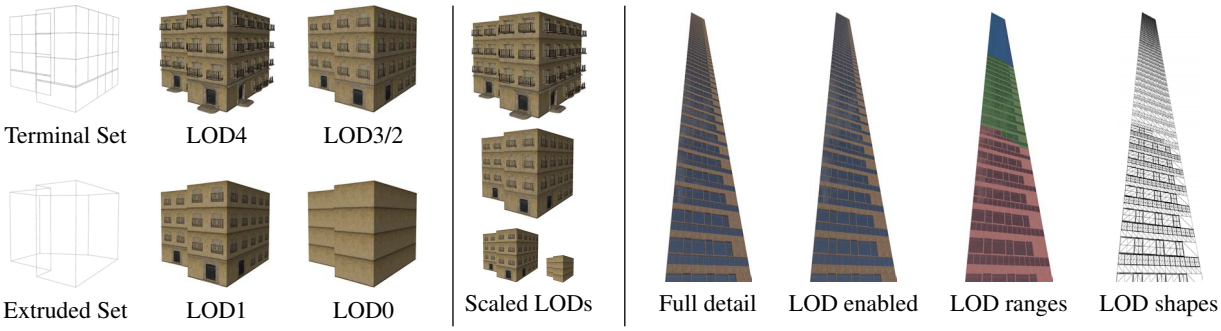


Figure 13: *LODs 2 to 4 use the entire terminal set: full terminal geometry (LOD4) or image-based impostors (LOD3/2). The LODs of terminals can be heterogeneous within a single facade (right). The coarser levels rely on a simplified terminal set: LOD1 applies per-pixel grammar expansion, while farthest objects are simply textured (LOD0).*

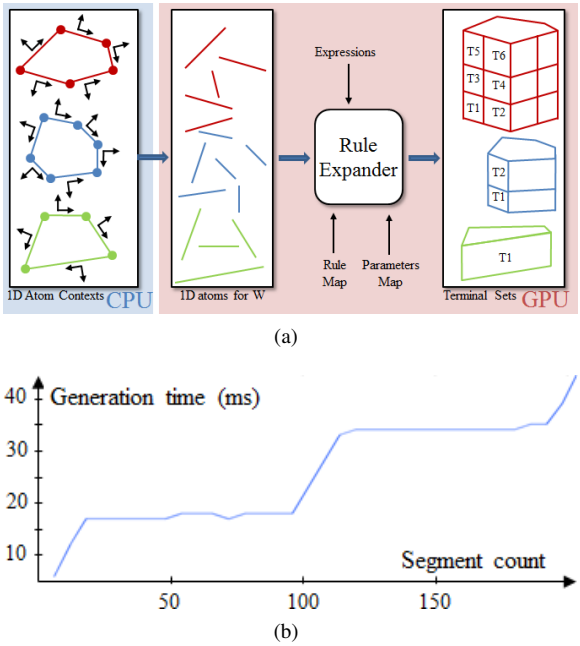


Figure 11: *Input footprints of buildings are divided into 1D atoms and expanded in parallel (a), yielding sublinear complexities (b). Single-atom trees also benefit from object batching.*

overall shape of the object. To this end, the grammar designer tags the grammar to indicate how to generate the *extruded set*. However, the presence of terminals remain visible even from long distances. For LOD1 the grammar is then lazily expanded for each pixel of the base volume, as in [MGHS11]. For each visible terminal the central view of the terminal impostor is sampled to ensure consistency with the higher levels. Finally, objects covering very few pixels are replaced by a base textured volume in LOD0 (Figure 13).

Discrete LODs usually exhibit "popping" level changes. In our approach LODs are smoothly blended for seamless transitions. Between LOD0 and LOD1 we simply blend the default texture with per-pixel grammar expansion. LODs 1 and 2 are visually identical, and hence not prone to popping. The transition to LOD3 is then performed by blending the views contained in the impostors.

The last level involves the terminal geometries, potentially generating protruding features. We avoid popping between LODs 3 and 4 by progressively flattening (*geofading*) the geometry as the screen coverage of the terminal decreases.

9. Results

We apply GPU Shape Grammars to the interactive generation and tuning of buildings and vegetation. The images and timings were obtained at a resolution of 1280×720 using an Intel Xeon 3.36GHz CPU and a Nvidia GeForce GTX 480.

The skyscrapers scene (Figure 14), containing 25K terminals (1.25M polygons), is generated in 21ms. The level of detail of each terminal depends on the viewing distance, yielding a rendering speed of 21.2fps with on the fly generation, and 38.4fps by storing the terminal set (1.9MB) within graphics memory. If stored the full scene geometry would take approximately 115MB. As our method features building-

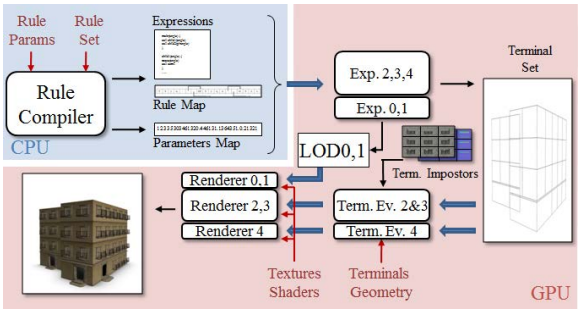


Figure 12: *The management of levels of detail is integrated within our pipeline for interactive generation and visualization of large-scale sceneries.*

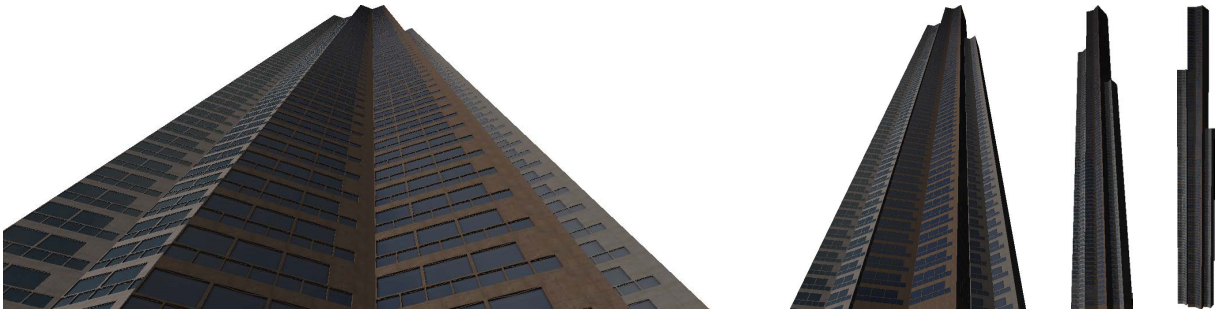


Figure 14: *Skyscrapers scene, from closeup to distant views. The towers comprise up to 209 floors, generated in 21ms and rendered at a minimum of 38.4fps using GPU Shape Grammars.*

grained parallelization, 4 similar scenes (100K terminals total) are generated in 26ms.

Our massive city scene comprises 116573 buildings and 561280 trees of 7 different species (Figure 1). The buildings are generated using two distinct grammars: the business district buildings use 28 rules, 3 of which being conditional. The other buildings are generated using 40 distinct rules, including 8 stochastic rules. The buildings generated using this second grammar use 12 unique terminal shapes. The impostors for each shape are generated at a resolution of 256×256, yielding a total memory footprint of 25MB.

The city is divided into 780 object batches. When LODs 2-4 are needed the terminal set of the batch is generated and stored in graphics memory (5MB per batch on average). The farther cells are rendered using LODs 0-1. The ray-traced grammars of LOD1 are visually equivalent to LOD2, providing smooth transitions. The scene is rendered at 7-15fps with on the fly generation of the terminal sets.

The representation of the detailed geometry for the entire city would take approximately 2.3TB, exceeding by far the available graphics memory. Using GPU Shape Grammars the entire representation for the scene is reduced to an average of 900MB. Besides real-time navigation, the accompanying video shows that the generation parameters of any arbitrarily complex set of buildings can be edited in place interactively. This capability finds a particular use for the production of massive assets for movie post-production. The detailed geometry can then be read back to generate final images using a production renderer.

10. Discussion

GPU Shape Grammars open a new way of thinking about procedural generation by introducing a unified pipeline avoiding the storage of detailed geometry. We discuss future challenges: snapping, occlusion and roof generation.

Snapping and Occlusion These operations adapt the generation of procedural models to their environments [MWH*06]. Typical example are the alignment of the window levels (snapping) and the avoidance of openings in occluded building facades. While those operations are costly in CPU-based generation schemes, our segment-based formulation for parallel grammar expansion forbids the application of such techniques within a single render pass. However, the terminal set can be read back and analyzed in a post-process to remove occluded terminals and snap terminal locations.

Roof Generation A unified solution for real-time roof generation of complex buildings is highly challenging [KW11]. Moreover, our approach does not preserve the overall roof footprints, precluding the computation of straight skeletons [ADAG95] on the GPU. We thus generate roof structures using this algorithm on CPU, caching the output within GPU buffers. Future work will consider processing roof skeletons using GPU Shape Grammars to generate features such as roofing tiles, chimneys and attic windows.

11. Conclusion

We introduced GPU Shape Grammars, a generic solution for real-time generation, tuning and rendering of procedural models. Based on an expression-instantiated rule interpreter coupled to parallel segment-based grammar expansion, our approach generates geometry on the fly within graphics hardware. The generated models are then directly streamed across the graphics pipeline, avoiding the storage of the fully detailed models. The generation quality can be finely tuned using multiple levels of detail automatically deduced from the grammar parameters.

We applied GPU Shape Grammars to the real-time generation, tuning and rendering of massive urban environments. The fast rendering capabilities of our method find a particular interest in interactive applications. Also, the ability for interactive tuning and rendering of arbitrarily massive models makes the GPU Shape Grammars approach a highly valuable tools within the visual effect industry.

References

[ADAG95] AICHHOLZER O., D.ALBERTS, AURENHAMMER F., GÄRTNER B.: A novel type of skeleton for polygons. *Journal of Universal Computer Science 1*, 12 (1995), 752–761. 8

[Cho65] CHOMSKY N.: *Aspects of the Theory of Syntax*. MIT Press, 1965. 2

[Edm60] EDMONDS J.: A combinatorial representation for polyhedral surfaces. *American Mathematical Society Notices 7* (1960), 646. 3

[GGH02] GU X., GORTLER S. J., HOPPE H.: Geometry images. In *Proceedings of SIGGRAPH* (2002), pp. 355–361. 2

[HWA*10] HAEGLER S., WONKA P., ARISONA S. M., GOOL L. J. V., MÜLLER P.: Grammar-based encoding of facades. *Computer Graphics Forum 29*, 4 (2010), 1479–1487. 2

[JWP05] JESCHKE S., WIMMER M., PURGATHOFER W.: Image-based representations for accelerated rendering of complex scenes. In *EUROGRAPHICS State of the Art Reports* (2005), pp. 1–20. 6

[KW11] KELLY T., WONKA P.: Interactive architectural modeling with procedural extrusions. In *Proceedings of SIGGRAPH* (2011), vol. 30, pp. 14:1–14:15. 8

[LCV03] LLUCH J., CAMAHORT E., VIVÓ R.: Procedural multiresolution for plant and tree rendering. In *Proceedings of AFRIGRAPH* (2003), pp. 31–38. 2

[Lie94] LIENHARDT P.: *n*-dimensional generalized combinatorial maps and cellular quasi-manifolds. *Intl Journal of Computational Geometry and Applications 4*, 3 (1994), 275–324. 3

[Lin68] LINDENMAYER A.: Mathematical models for cellular interactions in development parts i & ii. filaments with one-sided inputs. *Journal of Theoretical Biology 18*, 3 (1968), 280 – 299. 2

[LWW10] LIPP M., WONKA P., WIMMER M.: Parallel generation of multiple l-systems. *Computer and Graphics 34*, 5 (2010), 585–593. 2

[MGHS11] MARVIE J., GAUTRON P., HIRTZLIN P., SOURIMANT G.: Render-time procedural per-pixel geometry generation. In *Proceedings of Graphics Interface* (2011), pp. 167–174. 2, 7

[MPB05] MARVIE J., PERRET J., BOUATOUCH K.: The FL-system: a functional L-system for procedural geometric modeling. *The Visual Computer 1*, 5 (2005), 329–339. 2

[MWH*06] MULLER P., WONKA P., HAEGLER S., ULMER A., GOOL L.: Procedural modeling of buildings. In *Proceedings of SIGGRAPH* (2006), pp. 614–623. 2, 3, 5, 8

[PL90] PRUSINKIEWICZ P., LINDENMAYER A.: *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990. 2, 6

[WMWF07] WONKA P., MÜLLER P., WATSON B., FULLER A.: Urban design and procedural modeling. In *SIGGRAPH courses* (2007). 2

[WWSR03] WONKA P., WIMMER M., SILLION F., RIBARSKY W.: Instant architecture. In *Proceedings of SIGGRAPH* (2003), pp. 669–677. 2