EUROGRAPHICS 2014 / B. Lévy and J. Kautz
(Guest Editors)

Volume 33 (2014), Number 2

On-the-fly Generation and Rendering of Infinite Cities on the GPU

Markus Steinberger[1], Michael Kenzel[1], Bernhard Kainz[1], Peter Wonka[2], and Dieter Schmalstieg[1]

[1]Graz University of Technology, Austria
[2]King Abdullah University of Science and Technology, Saudi Arabia

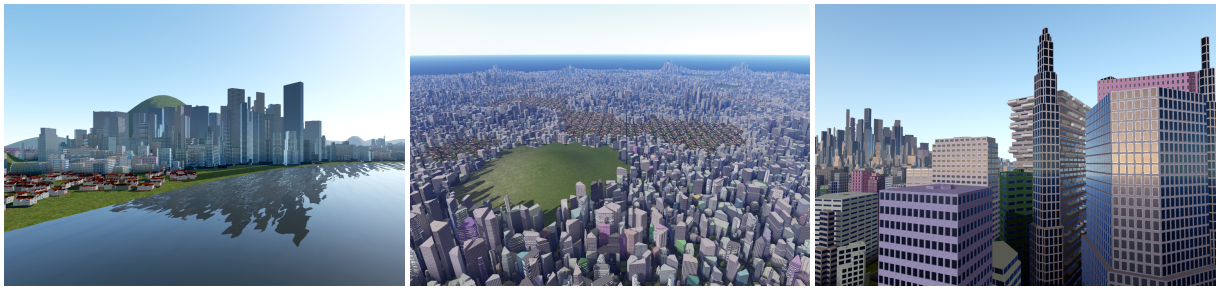


Figure 1: Infinite cities with highly detailed, context-sensitive buildings can be generated in real-time on the GPU using a parallel shape grammar. The visible 28 km^2 of the city contain up to 47 000 buildings. In full detail, these buildings would expand to 240 million rules, producing 2 billion triangles. Generating an initial view with adaptive level of detail (7 million triangles) from scratch takes 500 ms. Exploiting frame-to-frame coherence, we update the geometry for successive frames in 50 ms on a standard PC, even if the viewer moves at supersonic speed.

Abstract

In this paper, we present a new approach for shape-grammar-based generation and rendering of huge cities in real-time on the graphics processing unit (GPU). Traditional approaches rely on evaluating a shape grammar and storing the geometry produced as a preprocessing step. During rendering, the pregenerated data is then streamed to the GPU. By interweaving generation and rendering, we overcome the problems and limitations of streaming pregenerated data. Using our methods of visibility pruning *and adaptive level of detail, we are able to dynamically generate only the geometry needed to render the current view in real-time directly on the GPU. We also present a robust and efficient way to dynamically update a scene's derivation tree and geometry, enabling us to exploit frame-to-frame coherence. Our combined generation and rendering is significantly faster than all previous work. For detailed scenes, we are capable of generating geometry more rapidly than even just copying pregenerated data from main memory, enabling us to render cities with thousands of buildings at up to 100 frames per second, even with the camera moving at supersonic speed.*

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—

1. Introduction

Open world games such as *Batman: Arkham City* and *Grand Theft Auto* are massively successful, because they grant players absolute freedom in exploring huge, detailed virtual urban environments. The traditional process of creating such environments involves many person-years of work. A potential remedy can be found in procedural modeling using shape grammars. However, the process of generating a complete, detailed city the size of Manhattan, which consists of more than 100 000 buildings, can take many hours, producing billions of polygons and consume terabytes of storage. Rebuilding such a scene after parameter tweaking becomes a costly operation,

making rapid design iterations impossible. All these factors limit the practical usefulness of the procedural approach for large environments.

As the full-detail geometry of a huge environment does not fit into memory as a whole, it must be streamed from external storage for rendering. The organization and streaming of such data for real-time rendering is a complicated exercise. It could be avoided by generating only the geometry needed to render the current view on the fly [PB13]. Through the methods presented in *parallel generation of architecture* (PGA) [SKK*14], high-performance evaluation of state of the art shape grammars has become a possibility. Specifically designed for efficient, massively parallel execution on a current *graphics processing unit* (GPU), PGA can deliver grammar derivations of large cities in less than a second. However, the amount of geometry generated for huge cities by far exceeds storage capabilities of consumer graphics hardware.

In this work, we show how to extend the PGA evaluation scheme to significantly reduce the amount of geometry generated by taking into account visibility and different levels of detail. In doing so, we not only benefit from faster rendering due to less geometry, but can also greatly speed up grammar evaluation itself. The challenge herein lies in the fact that we cannot know in advance the exact shape of the geometry which will eventually be generated. Additionally, our evaluation scheme also takes advantage of frame-to-frame coherence. As the view usually does not change abruptly between two frames, the *derivation tree* [Sip06] is largely the same in successive frames. Instead of reevaluating the whole tree in each frame, we update the tree computed in previous frames. Robust and efficient handling of the dynamic data structures required to implement such an update procedure in a massively parallel environment is a nontrivial problem. In summary, our contributions are:

- We propose the method of *visibility pruning*. Instead of *culling* geometry after it has been generated, we are able to conservatively skip evaluation of rules to keep such geometry from being generated in the first place.
- We propose a method of handling adaptive level of detail during massively parallel grammar evaluation on the GPU. Considering the characteristics of shape grammar operators, *surrogate terminals* can automatically be prebaked for a given rule set, and applied during evaluation to avoid generation of visually insignificant detail.
- We present a method of dynamically adapting an existing derivation tree to a new view during parallel rule evaluation on the GPU. Using a voting scheme, our update mechanism can prioritize important generation steps to ensure grammar derivation meets real-time.
- We propose a dynamic vertex buffer and index buffer management scheme that allows geometry to be generated at any point during grammar derivation on the GPU. This buffer management forms the backbone for dynamically updating geometry across multiple frames.

2. Related work

The foundations of this work were laid in *parallel generation of architecture* (PGA) [SKK*14]. PGA itself builds on our CUDA-based implementation of the *Softshell* programming model [SKK*12] to schedule *CGA shape* [MWH*06] grammars on GPUs. Relevant prior work influencing CGA shape include *shape grammars* [Sti75], *set grammars* [Sti82], *L-systems* [PL90], and split operations for façades [WWSR03]. L-system evaluation has been implemented on the GPU using shaders [LH04], multi pass rendering [Mag09], and CUDA [LWW10]. Otherwise sequential evaluation can also be parallelized for the GPU by running it for each pixel [KBK13]. High computational redundancy, however, keeps such approaches from achieving real-time performance.

Geometry Streaming. The classical alternative to on-the-fly evaluation of shape grammars is streaming pregenerated geometry during rendering. To provide different *levels of detail* (LOD), mesh simplification algorithms such as vertex decimation [SZL92], progressive meshes [Hop96], or region merging [RR96] can be applied. Exploiting the special properties of buildings, LOD algorithms can be fine-tuned for handling cities [And05]. Mesh simplification can be performed on the GPU [JWLL06], and also be optimized for fast expansion on the GPU [HSH09]. Various out-of-core techniques can be employed in streaming pregenerated data to the GPU, including binary tree mesh partitioning [CGG*03], tetrahedra hierarchies [CGG*04], or clustered hierarchies [YSGM04]. A GPU-friendly out-of-core streaming approach for massive model rendering that also exploits frame-to-frame coherence has recently been proposed by Peng et al. [PC12]. Their take on resource management is less powerful than ours and requires the entire geometry to be available beforehand.

Visibility Culling. Especially in urban scenes, visibility culling can be of great benefit. If the geometry is static, occluder fusion can be used to compute potentially visible sets [WWS00]. Visibility can also be evaluated in parallel to rendering [WWS01], or be integrated with view-dependent rendering [ESSS01]. Using occlusion-switches [GSYM03], culling can be distributed over multiple graphics cards. If a voxel-based scene representation is available, a multiresolution framework can be built and used for culling [GM05]. The use of hardware occlusion queries has also been explored [BWPP04, MBW08]. All these approaches, however, rely on scene geometry, which is not yet available during grammar evaluation.

Generation of Cities. The generation of large urban environments typically involves multiple stages [PM01]. Heavily simplifying this model based on a fixed grid layout, a city of infinite extent can be generated for a single view on the CPU and rendered efficiently as shown by Greuter et al. [GPSL03]. Similar to our approach, their work depends on spatially-dependent pseudo-random numbers to guarantee consistent

derivations when returning to the same location. To reduce the number of derivations in each frame, buildings generated in previous frames can be kept in a simple cache [CO11]. Such a simple caching approach, however, cannot be employed in our massively parallel, GPU-based evaluation scheme with dynamic insertion of surrogate terminals. In the shader based split grammar by Marvie et al. [MBG*12], the use of a similar cache is possible only because a limited set of LOD representations is supported per building and visibility is left out of consideration. Additionally, they restrict their grammar to a simplified subset of CGA shape not supporting context-sensitivity, and evaluation is already at least an order of magnitude slower than PGA. An alternative way to incorporate LOD into grammar-based city generation is to precompute multiple versions of terminal shapes [BP13]. Because detail reduction is applied to terminals only, deep derivation trees will always generate large amounts of geometry. In contrast, our LOD approach is able to insert surrogate terminals at any point in the derivation tree.

3. Extended Evaluation Scheme

While PGA can generate a large, detailed city in a matter of seconds, the complete city will exceed graphics memory capacities. However, for rendering just a specific view, invisible geometry need not be generated and detail on distant buildings can be reduced. We introduce the notion of *visibility pruning*, *i. e.*, skipping the evaluation of rules that would generate geometry not being visible, which conceptually corresponds to pruning the derivation tree. Note the difference to *visibility culling*, where already generated geometry is discarded. To remove as many shapes with as little effort as possible, we first perform a coarse-grained *view frustum pruning*, where we skip rules at the root of entire buildings that would be constructed outside of the viewing frustum. The remaining rules are then subject to *occlusion pruning*, where we further skip rules that would generate geometry occluded by other elements of the scene. Derivation of rules that could potentially be skipped must not start before pruning has completed. In PGA, this kind of dependency is expressed by introducing a phase boundary between pruning and further derivation. Therefore, integrating these techniques into PGA implies a minimum of three evaluation phases:

1. **Building hulls.** Initially, for each building, a bounding volume called the *hull* is generated. Frustum pruning is performed on these hulls.
2. **Building specification.** The hulls inside of the view frustum are further expanded into a more detailed description of each building used to decide occlusion. We call this description the *building specification*.
3. **Building construction.** The rules remaining after occlusion pruning are evaluated to generate the final geometry. To stop evaluation of rules that would generate unnecessary detail, we adaptively insert suitable terminal shapes.

Figure 2 illustrates this pipeline, which is implemented completely on the GPU. Note that our pipeline is not limited to a three phase model, we only require a minimum of three phases to resolve the dependencies introduced by frustum pruning and occlusion pruning. Visibility pruning is exposed to the rule designer by means of tagging. If, *e. g.*, a shape should serve as a building's hull, the designer simply marks the rule generating the shape with the *hull* tag.

3.1. City Layout

As a first step in generating our virtual world, a *city layout* including streets and building lots needs to be created. We base our city layout on a regular grid of tightly packed *cells*. For each cell an initial rule is executed. This rule can either trigger the generation of a road network followed by lot definitions, or load street layouts and building footprints from GIS data. To ensure that derivations are consistent when returning to a previously visited location, every cell is supplied with its own pseudo-random seed, similar to the approach by Greuter et al. [GPSL03]. In addition, a list of manual customizations by a user could be stored for each cell. As a first, simple measure to limit the amount of generated data, we initiate rule derivation only for cells that are not further away from the camera than a certain *maximum viewing distance*.

3.2. Building hulls

During the first phase, for each building, a hull must be generated. The hull is assumed to be a conservative bounding volume—the building will not generate geometry outside of its hull—and thus be suitable for view frustum pruning. Building hulls can also be subject to queries in successive phases, *e. g.*, to determine the tallest building in an area, or to avoid the placement of balconies directly facing nearby buildings.

View frustum pruning If a shape has been tagged as a building hull, it is subject to view frustum pruning. We apply a trivial reject test: If all vertices are outside of the same side of the frustum, the building can be ignored. Although this test is not highly accurate, its simplicity usually outweighs potential gains through more accuracy. Individual vertices can be checked independently, making it a natural fit for the SIMD architecture of the GPU. If a building hull is determined not visible, the entire building is skipped for all following phases.

3.3. Building specification

During this phase, a *building specification*, *i. e.*, a geometric description of the building accurate enough to participate in occlusion pruning, is constructed. This process is similar to, and can be performed as part of, mass modeling. The building specification can be an arbitrary composition of shapes. Each shape is associated with the respective building and designated an *occluder type*. A shape can either be an

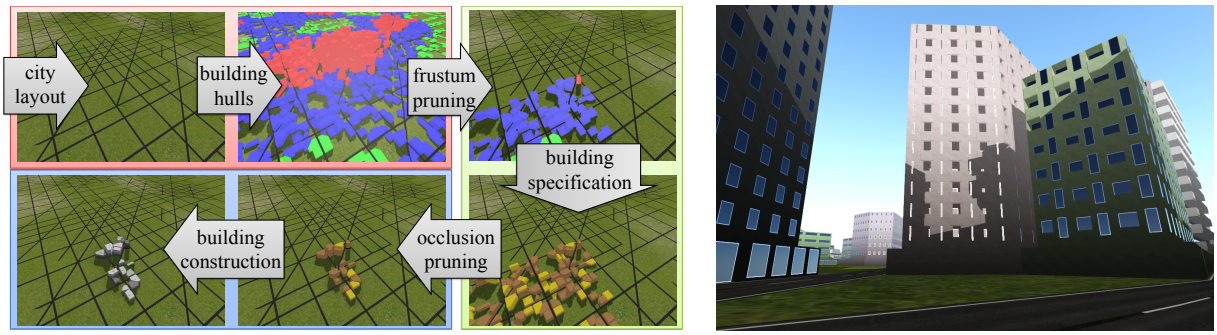


Figure 2: Integrating visibility pruning into PGA requires at least three phases (red, green, blue). Starting from the city layout, building hulls, which enclose individual buildings, are constructed and used for frustum pruning. The more detailed building specifications generated in the next phase are used for occlusion pruning. During the final stage, the geometry is constructed.

opaque, *enclosing*, or *hidden* occluder. Opaque occluders are expected to be entirely surrounded by solid geometry. Thus, shapes that are fully occluded by an opaque occluder can be discarded. Enclosing shapes form a boundary volume that will contain visible shapes. They act as a refinement of the building hull. Whenever part of an enclosing shape is visible, we conservatively assume that all its contents are going to be visible. Hidden shapes do not participate in occlusion pruning. An example is given in Figure 3.

Figure 3: Different occluder types illustrated for a suburban house: Boxes and wedges specify *opaque* parts of the building (gray). An enclosing box (yellow) is added to cover the fence. A hidden shape (red) marks the entrance way toward the door.

Occlusion pruning Especially in city walkthroughs, large portions of the urban environment can be removed by occlusion pruning. If an *opaque* or *enclosing* shape is determined to be visible, the building is constructed. If no part of the building specification is visible, the entire building is skipped.

The use of hardware occlusion queries would require every single building to be rendered to determine visibility. Since the efficiency of rule evaluation in PGA depends on uninterrupted derivation, we use a custom, hierarchical depth buffer [GK93] to perform occlusion pruning in software during derivation. The top level of our low-resolution depth buffer is divided into 8×8 tiles which are further subdivided to a maximum depth of five levels. In this configuration, 64 threads can test all shapes of a building specification in parallel, each thread traversing the subtree associated with a tile. For depth testing and updating of the depth buffer, we exploit the fact that all our shape primitives are convex. Depending on the kind of shape, we either use the convex hull computed from the projection of the shape's vertices for the depth test and update, or a circumscribed circle for the depth test and an inscribed circle for the update to determine the tiles affected by the shape. Convex hulls are computed using the parallel Jarvis' march algorithm [Jar73]. Both, opaque and enclosing shapes, are tested against the depth buffer, but only opaque shapes are written to the depth buffer.

For occlusion pruning to be effective, we have to ensure that occluders are written to the depth buffer before occluded shapes are tested. In a massively parallel derivation process, this is not automatically the case. A strict ordering could be enforced by introducing an additional evaluation phase. However, every shape would then have to go through depth buffering twice in addition to all the usual performance implications of global synchronization. Instead, we rely on an approximately optimal processing order by sorting building hulls according to their distance to the camera before starting the building specification phase. While some actually occluded buildings might unnecessarily be generated due to concurrent depth buffer updates, this less invasive approach will overall perform better in practice.

3.4. Building construction

For all buildings remaining after visibility pruning, geometry is finally generated. However, evaluating distant buildings to full detail would be wasteful, as most detail will not be noticeable and can cause aliasing artifacts. It would be desirable to stop evaluation early, generating only as much detail as visually important.

Procedural Level of Detail If further derivation would not visually enhance the rendering, suitable terminal shapes should be emitted to halt evaluation. We observe, that rule input shapes usually approximate the shapes generated during following derivations. Therefore, we propose to use automatically generated *surrogate terminals* which are based upon

these input shapes. We identify three issues that have to be addressed in such an approach: First, the input shape does not always match the shapes produced in the derivation, *e. g.*, if balconies are constructed on the tiles of a façade, the shapes do not even share the same dimensionality. Second, generated shapes may depend on random variables, and thus, a single surrogate cannot cover all possible instances. And third, there might be probabilistic selection among alternative rules.

To address these problems, we propose the following preprocessing approach to bake surrogate terminals for a given grammar: We start by generating samples of many building instances. We then traverse the derivation tree bottom up. To decide under which circumstances a parent shape can be used as surrogate for a rule, we render both, the detailed child shapes and the parent shape from various angles and viewing distances. For each viewing distance, we compute the average per-pixel difference between the potential surrogate and the detailed rule across all samples and viewing angles. We then fit a sigmoid function model approximating this view dependent per-pixel difference to allow efficient error estimation during rendering.

$$\text{tile} \rightarrow \text{Border}\{\textit{randA} : \text{wall},\ \textit{randA} : \text{wall},\ \textit{randA} : \text{wall},\ \textit{randB} : \text{wall},\ \text{window}\}$$

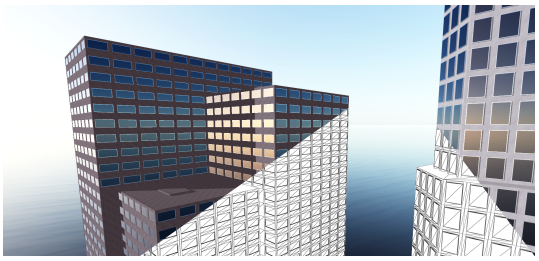
(a) Surrogate texture generation for the tile rule

$$\text{floor} \rightarrow \text{RepeatX}\{2.2 : \text{tile}\}$$

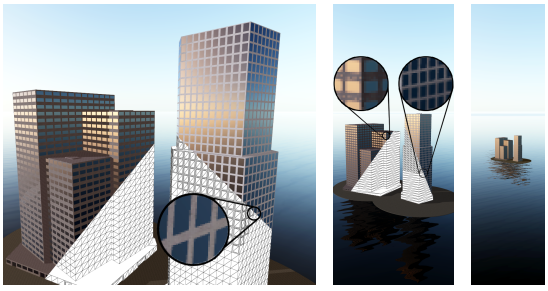
(b) Surrogate texture generation for the floor rule

Figure 4: Surrogate textures are generated from rule samples. Depending on the operators used, an archetype can be derived from the samples and rendered to generate a single texture per rule. This texture can then be warped and repeated to approximate any given instance of the rule (colored outline in the combined texture). In this way, a single surrogate texture can be used for different buildings as shown in Figure 5.

Similarly to image-based impostors [JWP05], surrogate terminals can be enhanced by applying an automatically generated texture showing an image of the more detailed shapes. By providing automatic texture generation and mapping for each shape operator, arbitrary rules can be supported. At this point, we are also able to deal with the influence of randomness on a shape's appearance. Consider the example given in Figure 4 and Figure 5: The tile rule splits a face into a randomly sized border and a window. First, we generate a



(a) 88*k* triangles (full detail)



(b) 7440 triangles (c) 1312 (d) 268

Figure 5: Different views on two towers rendered with adaptive level of detail. Both buildings can use the same surrogate texture, even though the window size and number of windows per façade differ. Also note that context-sensitive rules do not create windows where the parts of the left building meet.

number of samples for the tile rule. Knowing the characteristics of the border operator, we can derive an *archetype* from these samples and render it into a texture. This texture can then be warped to approximate any given instance of the rule. The floor rule example demonstrates the same strategy applied to the repeat operator. Again, we sample the space of rule products and generate a texture from an archetype, which can then be warped and repeated to approximate any instance of the rule. Following the same procedure, we can also store material parameters in another texture to achieve appropriate shading on different parts of the surrogate. For rules that entail probabilistic selection among multiple alternatives, we fall back to generating the surrogate texture by averaging all samples.

4. Frame-to-frame coherence

Given that the view changes only slightly from one frame to the next, major parts of the scene will be the same in both frames and, thus, could be reused. To enable such reuse, derivation trees need to be updated dynamically, taking into account visibility pruning as well as adaptive level of detail. Thus arises the need for dynamic memory management on the output buffers that receive the generated geometry. Robust and efficient dynamic data structures pose a major challenge on the GPU.

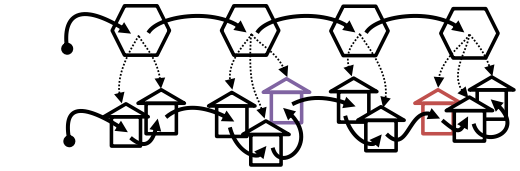
4.1. Sparse derivation tree

To keep track of objects in the scene, we use two arrays in GPU memory. The first array stores all cells within viewing distance. The second array stores all building hulls along with their building specifications for buildings located on these cells, sorted according to their distance to the camera. In each frame, based on the current and previous camera positions, we can compute which cells are no longer within viewing range and which cells have just come into viewing range. After removing cells which are no longer in viewing range (along with their buildings), we trigger the PGA evaluation scheme for all cells which came into viewing range and all building hulls stored in the array. As some buildings have already been generated in a previous frame, we extend the PGA evaluation scheme as follows:

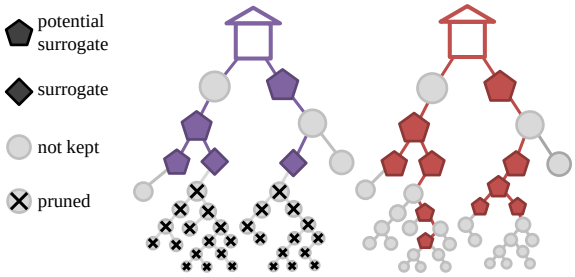
- If a previously visible building is moved out of the frustum, we remove all geometry associated with it, but keep its building hull and building specification in the second array to enable a quick rebuild.
- If a previously visible building is completely occluded, we increase a counter we keep for each building. As occlusion could be temporary, we remove the building's geometry only if it has been occluded for a certain number of frames.
- If a building comes back into sight, we start the PGA derivation at the building specification which is still present in the second array.
- If a building remains not visible, there is no need for any derivation.
- If a previously visible building is still visible, we reevaluate the building's level of detail as outlined in the following.

To enable reevaluation of a building's level of detail, we keep a sparse version of the derivation tree for all visible buildings in memory. This sparse tree contains all possible surrogate terminals and all actually inserted surrogates, as shown in Figure 6. During the PGA evaluation, instead of directly executing the building's rule set, we first traverse this tree and recompute the error estimates for all nodes. If the error estimate suggests the insertion or removal of a surrogate, a vote is cast for the reevaluation of the building. We accumulate these votes over multiple frames, and, if the sum of votes exceeds a given threshold, rebuild the building. The number of votes can also be used to prioritize the reevaluation of those buildings which are most in need of a rebuild, while delaying the rebuild for buildings which are nearly consistent with the current view to a later frame.

Our approach of rebuilding entire buildings instead of adjusting the derivation tree locally may seem inefficient. However, the alternative would require the ability to remove small amounts of geometry from the output buffers, which, over time, can lead to strong fragmentation and high memory management overheads. The proposed scheme, on the contrary, works with larger portions of memory and allows us to settle for more lightweight memory management strategies.



(a) List of cells and building hulls



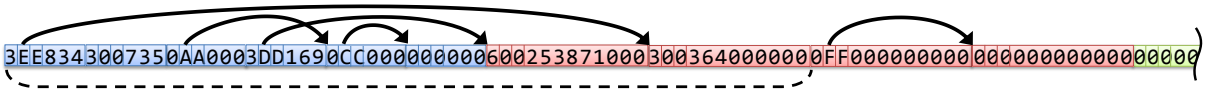
(b) Sparse derivation tree for two instances of the same building

Figure 6: To exploit frame-to-frame coherence, we keep a list of all active world cells and buildings on these cells (a). In every frame, we run through these lists in parallel and reevaluate the state of each entity. To enable a reevaluation of the pruned derivation tree for each building, we keep a sparse derivation tree for each building (b). This tree contains potential and active surrogates only. We continuously recheck these surrogates and rebuild a building if the level of detail does not fit anymore.

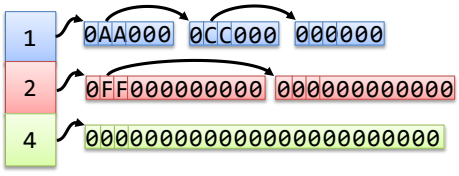
4.2. Buffer management

In our setup, we use a vertex and an index buffer to store the geometry generated by PGA. CUDA and OpenGL interoperability features enable the buffers filled by PGA to be directly used for rendering in OpenGL. Instead of directly producing geometry, terminal shapes could be rendered using hardware instancing, in which case the output buffers would receive instance data. In both cases, we face the same challenges: First, geometry of arbitrary size may have to be inserted or deleted at any point in time. Second, as we are dealing with potentially large amounts of data, we cannot afford to waste memory. Third, the necessity to move around data should be avoided to save bandwidth. Thus, the buffers storing the geometry should be directly usable for rendering. Neither previous buffer management strategies [PC12], nor general purpose GPU dynamic memory allocators [SKKS12] are able to meet these demands. We propose a novel, dynamic memory management scheme that builds its data structures directly in the index buffer, and is able to serve thousands of concurrent allocation requests efficiently.

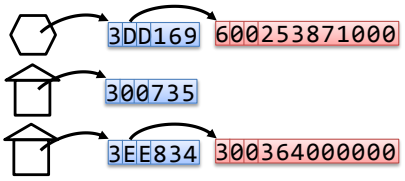
Buffer Setup. We assume vertex and index buffers are of sufficient size to hold all geometry that will be generated. Similar to the buddy memory allocator [Kno65], we partition the index buffer into *blocks*. Each block in the index buffer



(a) The index buffer subdived into blocks of different size: only the lower part of the index buffer is used for rendering (dashed).



(b) A list of free blocks is kept for each block size



(c) Each cell and building has a list of in-use blocks

Figure 7: To dynamically manage the buffers used for terminal geometry, we build a parallel buddy allocation scheme within the index buffer (a). A list of free blocks serves allocation requests (b). Memory allocations are routed over buildings (and cells), which keep a list of blocks in use (c). In every block, we use the first index as atomically operated counter capturing the number of indices in use, and the second index as pointer linking to the next list element. To ensure this meta data is skipped during rendering, we keep a copy of the second index in the third, generating a degenerated triangle.

is associated with a block in the vertex buffer, based on an estimated vertex to index ratio. Thus, we avoid the need for additional data structures to manage the vertex buffer. As our allocation algorithm allows *holes* in the buffers, we zero-initialize the index buffer to ensure unused parts are ignored during rendering.

Incremental allocation and combined free. Since we always regenerate entire buildings, instead of micromanaging small parts of geometry, we propose the use of *incremental allocation* and *combined free*. Associated with the building hull, we keep a list of blocks allocated for each building. During shape derivation, we request memory from this list. If there is no more space to be found in this list, we request a new block from the memory allocator. As we cannot foretell how much geometry will be needed for a specific building, we use an adaptive allocation strategy: We start by allocating a small block only. Whenever a block is full, we request a block twice the size of the previous block and add it to the list. Following such a strategy of doubling the allocation size, only little memory is requested for low-detail buildings, while at the same time, full-detail buildings require only a few allocations. If a building's geometry is to be removed, we run through the block list, zero all blocks, and free them. During rendering, we draw all triangles between index zero and the highest index in use. As the unused areas contain zeros only, they will be ignored.

Memory block management. For each block size used by the allocator, we keep a list of free blocks. If a new block is requested, we check if a free block of suitable size is available. If this is the case, we remove it from the list. If not, we take a block of larger size and split it. When a block is freed, it is inserted back into the respective list of free blocks. To counteract fragmentation, we scan these lists between rendering and merge neighboring blocks unless the number of blocks available for the given size is below a threshold. Furthermore, we move blocks starting at lower indices to the front. As blocks are removed from the front of the lists, this strategy ensures that holes within the buffer are filled before new memory is requested from the back,

As outlined in Figure 7, we use the first three indices in each block to store meta information to build our data structure. The first index serves as an atomically operated counter keeping track of the number of indices available. The second index is used as a pointer to the next block in the list. The third index holds the same value as the second, forming a degenerate triangle that is skipped during rendering.

5. Results

To evaluate the effect of visibility pruning and adaptive level of detail, we tested four static scenes and sequentially activated the individual techniques. To evaluate our sparse derivation tree and buffer management to exploit frame-to-frame coherence, we evaluated a continuous camera movement through an infinite city. Due to the fact that PGA itself is already orders of magnitude faster than all other grammar evaluation schemes [SKK*14], we only compare our methods to the PGA baseline. All measurements were run on the same machine with an Intel Core i7-940 Quad Core CPU (2.93 GHz) and an NVIDIA Quadro 6000 GPU.

Generating a single view. The four test scenes used for the static scene evaluation were *Suburban*, which can be seen in Figure 3, *Green Town Streetside* (Figure 2 right), *Green Town Birdseye* (Figure 2 left view without pruning), and *Infinite Airplane* (Figure 1 center), with a visibility distance of 500 m, 2000 m, 2000 m, and 8700 m, respectively. In addition to our GPU implementation, we also provide a CPU version for comparison. The characteristics of the scenes and results for all tests are shown in Table 1.

		buildings	nodes	terminals	vertices	triangles	memory	GPU	CPU	trans	render
Suburban (500m)	PGA full	868	490.9k	3.8M	102.1M	59M	425MB	1.9k	4.0k	8.1k	88.3
	LOD	868	124.8k	647.7k	14.6M	8.8M	74MB	187	655	1.2k	14.5
	LOD+FRT	311	38.8k	202.1k	1.7M	926.2k	22MB	32.8	128	51.6	6.0
	LOD+FRT+OCC	265	26.8k	150.6k	1.3M	723.5k	19MB	26.4	98.8	47.9	5.8
Green Town Streetside (2000m)	PGA full	5.9k	1.3M	6.5M	37.5M	18.8M	800MB	1.1k	2.4k	1.7k	27.3
	LOD	5.9k	43.6k	229.2k	2.4M	1.2M	23MB	81.1	304.9	81.3	4.5
	LOD+FRT	1.6k	20.7k	64.9k	610.8k	321.3k	7.2MB	26.1	97.8	11.3	3.5
	LOD+FRT+OCC	17	4.0k	7.6k	64.2k	54.9k	4.0MB	5.23	20.3	2.64	3.2
Green Town Birdseye (2000m)	PGA full	4.5k	1.2M	6.1M	35.1M	17.6M	684MB	743	2.5k	1.8k	114.5
	LOD	4.5k	34.8k	176k	2.0M	1.0M	20MB	43.4	174	42.3	6.8
	LOD+FRT	1.1k	21.3k	79.4k	788.7k	410k	10MB	28	110	16.8	6.3
	LOD+FRT+OCC	1.1k	21.2k	79.2k	786k	409.9k	10MB	32.1	121	16.3	6.3
Infinite Airplane (8700m)	PGA full	151.2k	183.4M	1.1G	7.4G	3.8G	121GB	-	13.5M	-	-
	LOD	151.2k	941.6k	6.6M	54.8M	28.6M	735MB	1.4k	5.2k	3.2k	141.2
	LOD+FRT	30.4k	501.3k	2M	15.4M	8.2M	279MB	721	2.8k	864	15.3
	LOD+FRT+OCC	30.4k	501.2k	2M	15.4M	8.2M	279MB	793	3.1k	863	15.3

Table 1: Scene characteristics, memory requirements for intermediate shapes, generation times (in ms) for our GPU and CPU implementation, transfer time (in ms) if the data generated were copied from main memory to the GPU, and rendering time (in ms). While adaptive level of detail (LOD) and frustum pruning (FRT) always increase performance, occlusion pruning (OCC) only helps if the camera is close to the ground, like in the *Green Town Streetside* scenario. Our techniques not only reduce the generation time, but also the memory requirements for intermediate symbols and rendering time. GPU generation is always faster than CPU generation, in ten out of the 16 cases even faster than copying pregenerated geometry to the GPU.

For all test scenes, we recorded a clear benefit using adaptive level of detail and frustum pruning. When applying adaptive level of detail, the performance increased by a factor of 10 to 17. As surrogate terminals affect distant buildings, the gain increased with increasing visibility distance. Due to memory restrictions, we were unable to derive the *Infinite Airplane* scenario in full detail on the GPU. However, adaptive level of detail reduces the amount of geometry by a factor of 100 and GPU generation becomes possible.

When also using frustum pruning, we could further increase the derivation speed by a factor of 1.5–5.7. The lowest speedup was achieved for *Green Town Birdseye*, in which the camera points to the city center. Thus, the most complex buildings were not pruned. Using occlusion pruning, the performance can further be increased in scenes where the camera is positioned on the ground. For instance, we achieved a speedup of five for *Green Town Streetside*. If hardly any buildings can be pruned, like in the birds-eye view, occlusion pruning actually decreases performance.

Our implementation of pruning and adaptive level of detail does not only work well on the GPU, we achieved very similar performance gains on the CPU. Still, the GPU implementations were always between two and five times faster than the corresponding CPU versions. The lowest speedups were achieved for the full derivations, which we account to the high memory usage, slowing down the memory allocator [SKKS12]. Using our techniques, the memory required for intermediate shape data and generated geometry is reduced by a factor of 22–200, which allows the generation of large cities witin the memory constraints of consumer graphics cards. The reduced amount of to-be-rendered geometry also increases rendering speed by a factor of 10–20.

Another interesting fact is that in two thirds of all cases, we were able to derive the geometry on the GPU faster than it would take to copy the data from main memory. Even when large portions of the derivation tree are pruned and, thus, the time spent on pruning becomes dominant, the evaluation process does not take more than twice as long as the memory transfer would take. This clearly shows the advantages of grammar derivations directly on the GPU. Overall, we can state that, due to their low overhead, adaptive level of detail and frustum pruning should be used in all cases. Occlusion pruning is only effective, if the camera is close to the ground.

Rendering during continuous movement. To evaluate the effect of frame-to-frame coherence, we use our *Infinite City* test case with adjustable clipping distance. In this setup, we use the population density to guide the generation of street layouts and selection of building types. The street layout is derived using a simple parallel split grammar. To connect streets of neighboring cells, we use the sibling query feature of PGA. As clipping distance we chose 1000 meters and 3500 meters, leading to an average of 3500 and 47 k buildings being within in the visible range respectively. A full detail

generation of these 3500 buildings requires 18 M rules and 150 M polygons; 47 k buildings need 240 M rules generating 2 billion polygons. The LOD versions require 1.6 M and 7 M polygons only. Our test scenario contains a walkthrough (movement speed 1.5 m/s) of the suburban area, a drive to a dense skyscraper area (20 m/s), rising over the city (50 m/s) and flying above the city at rocket speed (300 m/s–100 km/s). Selected frames from the scenario are shown in Figure 1. We tested four methods: *CPU* derives the required geometry on the CPU (using adaptive level of detail) and transfers it to graphics memory, *LOD* derives the geometry on the GPU using adaptive level of detail, *Pruning* additionally uses visibility pruning, and *Frame to Frame* uses our proposed techniques to reuse geometry generated in previous frames.

The generation times for all methods and two clipping distances are shown in Figure 8. For 1000 m clipping distance, *CPU* needs between 250 ms and 400 ms per frame. Our basic GPU implementation is about five times faster with an average of 65 ms per frame. With pruning, performance approximately doubles to 30 ms per frame. If, additionally, we exploit frame-to-frame coherence, only about 5 ms per frame are needed after the initial frame, yielding a speedup of about 60 compared to *CPU*. For 3500 m clipping distance, we achieved the following results: *CPU* 3500 ms, *LOD* 525 ms, *Pruning* 280 ms, and *Frame to Frame* 50 ms per frame. Again, our full GPU implementation was able to create the city in real-time and was about 60 times faster than *CPU*.

During both tests, we hardly noticed any influence of the movement speed on the performance of *Frame to Frame*. Only in the 3500 m clipping scenario after increasing the movement speed to 10 km/s subsequent frames no longer have significant coherence and incremental evaluation is even detrimental to performance. For 1000 m clipping *Frame to Frame* worked well even for very high speeds. A much easier way to break frame-to-frame coherence is camera rotation, which was not considered in the frame-to-frame coherence scheme. We performed two fast camera rotations during the walking phase, leading to spikes in the frame rate. Still, the system recovers quickly. Overall, our approach achieved real-time performance in both scenarios.

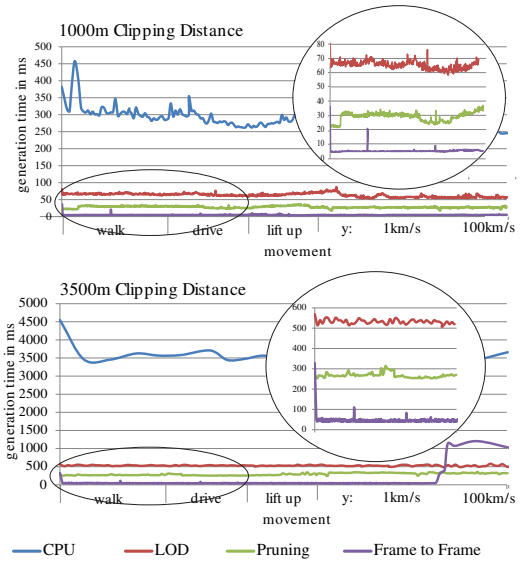


Figure 8: Comparison of generation times in an infinity city (Figure 1). Generating geometry on the CPU and streaming it to the GPU is too slow for interactive rendering. Our GPU method with frame-to-frame coherence updates the geometry needed for rendering in 5 ms (1000 m) and 50 ms (3500 m) respectively. Frame-to-frame coherence breaks if the camera is rotated fast (spikes in enlarged areas) or if the movement speed is very fast (about 10 km/s).

6. Discussion and Conclusion

We have shown that taking into account visibility during the evaluation of shape grammars is possible and can lead to vast improvements in evaluation speed, memory consumption and rendering speed. By using *frustum pruning* and *occlusion pruning*, we skip the generation of buildings that are not going to be visible. While *frustum pruning* most often reduces the number of generated buildings by 75%, *occlusion pruning* works best at street level, where it saves up to 90% of otherwise generated geometry. To further reduce the number of processed shapes and allow for highly detailed cities, we proposed an automatic approach for the generation and insertion of surrogate terminals, reducing the amount of geometry by about 90%. To reach full real-time performance, we exploit frame-to-frame coherence, using our dynamic buffer management scheme and tracking changes in the derivation trees over time. In this way, we generate and update cities with 47 000 visible buildings at 20 fps, even when the viewer is moving at supersonic speed.

Although our results are very promising, we see the need to explore visibility pruning and the generation of surrogate terminals in more detail. For instance, occlusion pruning could incorporate the depth buffer rendered in the last frame, the authoring of building specifications could be automated, and the concept of hulls and building specifications should be generalized to arbitrary objects. There are multiple benefits for the integration of our dynamic shape-grammar-based geometry into real-time applications. Our techniques enable real-time exploration of huge virtual worlds as needed for planning and simulation scenarios as well as games. They also greatly reduce the time needed to derive the geometry for a single rendering, enabling fast design iterations during content creation. Thus, we expect to see parallel shape grammar evaluation schemes executed on massively parallel processors in a variety of future applications.

Acknowledgments This research was funded by the Austrian Science Fund (FWF): P23329.

References

[And05] ANDERS K.-H.: Level of detail generation of 3d building groups by aggregation and typification. In *International Cartographic Conference* (2005). 2

[BP13] BESUIEVSKY G., PATOW G.: Customizable LoD for procedural architecture. In *Comp. Graph. Forum* (2013), vol. 32. 3

[BWPP04] BITTNER J., WIMMER M., PIRINGER H., PURGATHOFER W.: Coherent hierarchical culling: Hardware occlusion queries made useful. In *Comp. Graph. Forum* (2004), vol. 23, pp. 615–624. 2

[CGG*03] CIGNONI P., GANOVELLI F., GOBBETTI E., MARTON F., PONCHIO F., SCOPIGNO R.: Bdam - batched dynamic adaptive meshes for high performance terrain visualization. *Comp. Graph. Forum 22* (2003), 505–514. 2

[CGG*04] CIGNONI P., GANOVELLI F., GOBBETTI E., MARTON F., PONCHIO F., SCOPIGNO R.: Adaptive tetrapuzzles: efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. *ACM Trans. Graph. 23* (2004), 796–803. 2

[CO11] CULLEN B., O'SULLIVAN C.: A caching approach to real-time procedural generation of cities from gis data. 2

[ESSS01] EL-SANA J., SOKOLOVSKY N., SILVA C. T.: Integrating occlusion culling with view-dependent rendering. In *Proc. of Visualization '01* (2001), pp. 371–378. 2

[GK93] GREENE N., KASS M.: Hierarchical Z-Buffer Visibility. In *Proc. SIGGRAPH'93* (1993), pp. 231–238. 4

[GM05] GOBBETTI E., MARTON F.: Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. *ACM Trans. Graph. 24* (2005), 878–885. 2

[GPSL03] GREUTER S., PARKER J., STEWART N., LEACH G.: Real-time procedural generation of 'pseudo infinite' cities. In *Proc. GRAPHITE 03* (2003), pp. 87–ff. 2, 3

[GSYM03] GOVINDARAJU N. K., SUD A., YOON S.-E., MANOCHA D.: Interactive visibility culling in complex environments using occlusion-switches. In *Proc. I3D '03* (2003), pp. 103–112. 2

[Hop96] HOPPE H.: Progressive meshes. In *Proc. SIGGRAPH '96* (1996), pp. 99–108. 2

[HSH09] HU L., SANDER P. V., HOPPE H.: Parallel view-dependent refinement of progressive meshes. In *Proc. I3D '09* (2009), pp. 169–176. 2

[Jar73] JARVIS R.: On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters 2*, 1 (1973). 4

[JWLL06] JI J., WU E., LI S., LIU X.: View-dependent refinement of multiresolution meshes using programmable graphics hardware. *The Visual Computer 22* (2006), 424–433. 2

[JWP05] JESCHKE S., WIMMER M., PURGATHOFER W.: Image-based representations for accelerated rendering of complex scenes. *STAR reports, Eurographics 2005* (2005), 1–20. 5

[KBK13] KRECKLAU L., BORN J., KOBBELT L.: View-Dependent Realtime Rendering of Procedural Facades with High Geometric Detail . *Comp. Graph. Forum 32*, 2pt1 (2013). 2

[Kno65] KNOWLTON K. C.: A Fast Storage Allocator. *Commun. ACM 8*, 10 (1965), 623–624. 6

[LH04] LACZ P., HART J.: Procedural Geometry Synthesis on the GPU. In *Workshop on General Purpose Computing on Graphics Processors* (2004), pp. 23–23. 2

[LWW10] LIPP M., WONKA P., WIMMER M.: Parallel Generation of Multiple L-systems. *Computers & Graphics 34*, 5 (2010), 585–593. 2

[Mag09] MAGDICS M.: Real-time Generation of L-system Scene Models for Rendering and Interaction. In *Spring Conf. on Computer Graphics* (2009), Comenius Univ., pp. 77–84. 2

[MBG*12] MARVIE J.-E., BURON C., GAUTRON P., HIRTZLIN P., SOURIMANT G.: GPU Shape Grammars. *Comp. Graph. Forum 31*, 7-1 (2012), 2087–2095. 3

[MBW08] MATTAUSCH O., BITTNER J., WIMMER M.: Chc++: Coherent hierarchical culling revisited. In *Comp. Graph. Forum* (2008), vol. 27, pp. 221–230. 2

[MWH*06] MÜLLER P., WONKA P., HAEGLER S., ULMER A., GOOL L. V.: Procedural Modeling of Buildings. *ACM Trans. Graph. 25*, 3 (2006), 614–623. 2

[PB13] PATOW G., BESUIEVSKY G.: Challenges in Procedural Modeling of Buildings. In *Eurographics Workshop on Urban Data Modelling and Visualisation* (2013). 2

[PC12] PENG C., CAO Y.: A GPU-based approach for massive model rendering with frame-to-frame coherence. In *Comp. Graph. Forum* (2012), vol. 31, pp. 393–402. 2, 6

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

[PM01] PARISH Y. I. H., MÜLLER P.: Procedural modeling of cities. In *Proc. SIGGRAPH 2001* (2001), pp. 301–308. 2

[RR96] RONFARD R., ROSSIGNAC J.: Full-range approximation of triangulated polyhedra. In *Comp. Graph. Forum* (1996), vol. 15, pp. 67–76. 2

[Sip06] SIPSER M.: *Introduction to the Theory of Computation*, vol. 2. Thomson Course Technology Boston, 2006. 2

[SKK*12] STEINBERGER M., KAINZ B., KERBL B., HAUSWIESNER S., KENZEL M., SCHMALSTIEG D.: Softshell: Dynamic Scheduling on GPUs. *ACM Trans. Graph. 31* (2012). 2

[SKK*14] STEINBERGER M., KENZEL M., KAINZ B., MÜLLER J., WONKA P., SCHMALSTIEG D.: Parallel generation of architecture on the GPU. *Comp. Graph. Forum 33* (2014). 2, 7

[SKKS12] STEINBERGER M., KENZEL M., KAINZ B., SCHMALSTIEG D.: ScatterAlloc: Massively parallel dynamic memory allocation for the GPU. In *Innovative Parallel Computing* (2012). 6, 8

[Sti75] STINY G.: *Pictorial and Formal Aspects of Shape and Shape Grammars*. Birkhauser Verlag, 1975. 2

[Sti82] STINY G.: Spatial Relations and Grammars. *Environment and Planning B 9* (1982), 313–314. 2

[SZL92] SCHROEDER W. J., ZARGE J. A., LORENSEN W. E.: Decimation of triangle meshes. In *ACM SIGGRAPH Computer Graphics* (1992), vol. 26, pp. 65–70. 2

[WWS00] WONKA P., WIMMER M., SCHMALSTIEG D.: Visibility preprocessing with occluder fusion for urban walkthroughs. In *Proc. EG Workshop on Rendering Techn.* (2000), pp. 71–82. 2

[WWS01] WONKA P., WIMMER M., SILLION F.: Instant visibility. In *Comp. Graph. Forum* (2001), vol. 20, pp. 411–421. 2

[WWSR03] WONKA P., WIMMER M., SILLION F. X., RIBARSKY W.: Instant Architecture. *ACM Trans. Graph. 22* (2003), 669–677. 2

[YSGM04] YOON S.-E., SALOMON B., GAYLE R., MANOCHA D.: Quick-vdr: Interactive view-dependent rendering of massive models. In *Visualization, 2004. IEEE* (2004), pp. 131–138. 2