GPU Roof Grammars

Cyprien Buron, Jean-Eudes Marvie and Pascal Gautron

Technicolor

(a) Hipped with halfway dormer (b) Haussmann-type (c) Normandy

Figure 1: *GPU roof grammars allow for efficient modeling of complex roof structures. Note that gutters, chimneys, ridge tiles and overhangs are also generated by the system.*

Abstract

*We extend GPU shape grammars [MBG*12] to model highly detailed roofs. Starting from a consistent roof structure such as a straight skeleton computed from the building footprints, we decompose this information into local roof parameters per input segments compliant with GPU shape grammars. We also introduce* Join *and* Project *rules for a consistent description of roofs using grammars, bringing the massive parallelism of GPU shape grammars to the benefit of coherent generation of global structures.*

1. Introduction

Procedural methods provide an efficient way to model a wide variety of elements composing large sceneries, such as vegetation and buildings. CPU-based methods are generally based on the refinement of complex data structures. While this approach is valuable on single-core systems, efficient parallel approaches tend to subdivide the work into smaller independent tasks. In this way GPUs have been recently used to improve interactive generation, edition and visualization of such highly detailed models, especially in massive scenes [LWW10, HWA*10, MBG*12]. In particular Marvie *et al.* [MBG*12] decomposed initial footprints into 1D atoms (segments). While enforcing massive parallelism, global structures are lost in the process requiring ad-hoc solutions for roof generation.

Our contributions bring parallelism to grammar-based roof generation. First, a CPU component converts a global roof structure computed from building footprints into local roof information consistent with the 1D atoms of Marvie *et al.* [MBG*12]. This information typically include the target roof height and slopes. We also enrich GPU shape grammars [MBG*12] with *Join* and *Project* rules to guide roof creation using the information distributed among the 1D atoms. Following the GPU shape grammars pipeline (Figure 2), we evaluate the new rules at interpretation stage to benefit from the parallel processing capabilities of the graphics hardware.

2. Previous work

Procedural techniques are common for modeling objects that can be built from growth and reduction processes, such as vegetation and architecture. The object structure is defined by grammar rules and parameters, where terminal symbols may be substituted by shapes or textures. Various grammar types have been developped for the generation of buildings and facades on the CPU, among which L-systems [PM01], Split grammars [WWSR03], FL-systems [MPB05] and CGA shape grammars [MWH*06]. A framework for interactive edition of buildings was also proposed by [LWW08]. Kelly and Wonka [KW11] introduced an interactive method based on extrusion profiles that replace classical grammar rules. All these CPU-based methods have a high memory and computational cost for object generation, and potential bandwidth issues when transferring the geometry to the GPU for rendering.

Recent works focus on moving the generation step to the fragment shading stage of the GPU for immediate rendering, saving both memory and bandwidth [HWA*10, MGHS11]. Marvie *et al.* [MBG*12] further leverage the inherently parallel structure of GPUs by designing a complete pipeline for interactive interpretation and rendering of shape grammars. Based on 1D atoms extracted from input footprint segments, the grammar rules are first interpreted on the GPU, given a set of terminal symbols (*i.e.* structural information). Footprints are decomposed into 1D atoms to avoid working on complex structures. The geometry of each terminal is then substituted to the structural information to generate highly detailed objects interactively. However, this decomposition into independent elements precludes the generation of consistent roof structures on the GPU.

Roof construction requires a global processing of the building footprints to extract the ridges and gables. The straight skeleton algorithm [ADAG95] computes the skeleton of the polygon by sweeping the edges according to the bissector angles. This technique has been applied to the generation of complex structures such as mansard and hipped roofs, with [MWH*06] or without [LD03, KW11] grammars. However, due to the strongly sequential nature of this algorithm, most GPU methods for procedural architecture compute roofs on the CPU and transfer the geometry to the GPU for rendering [HWA*10, MBG*12].

3. Principle

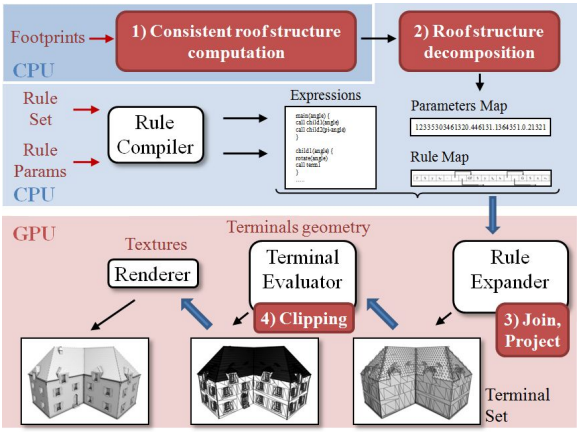


Figure 2: *Overview of our method used for GPU roof generation. New steps are indicated in red. 1) Roof structure extraction. 2) Conversion of global roof structure into 1D atoms roof information. 3) Roof sections are generated in parallel, at rule expander stage. 4) Finally terminal evaluation stage performs the clipping of roof shapes.*

GPU roof grammars are based on four steps integrated within the GPU shape grammars pipeline (Figure 2). We first determine a consistent roof structure on the CPU from an input footprint. We then decompose this information into parameters for 1D atoms (Section 3.1). Since GPU shape grammars [MBG*12] are not sufficient to model roof sections, we add *Join* (Section 3.2) and *Project* (Section 3.4) rules to the rule expansion stage. Finally, the clipping operation required by the *Join* rule is performed during terminal evaluation (Section 3.3).

3.1. Roof structure decomposition

In this step we decompose the output of the CPU-based roof structure generator. To this end we associate one roof section to each segment of the input footprint and store this information into the parameters of the 1D atoms (Figure 2). We consider two cases of 1D atom roof section: the segment either reaches another segment (ridge), or a vertex (gable). We excluded cases where the target is composed of two or more segments to avoid T-vertices. From this observation both cases can be reduced to a segment to segment operation, where a gable is obtained by duplicating the destination vertex.

3.2. Join rule

Filling a non rectangular quadrilateral with a set of shapes may be performed in two ways. The geometry can be either distorted by an extrusion to fit the quadrilateral (Figure 3a), or simply clipped by the edges (Figure 3b). To avoid distortions, we introduce the *Join* rule:

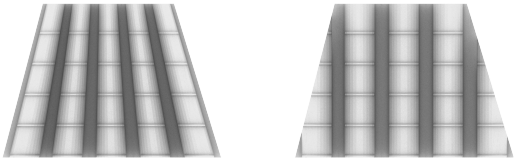
$$\text{Pred} \rightarrow \text{Join}(\text{ float heightExtrusion, vec2 v1, vec2 v2 })\ \{\text{quadClipped, quadSupport, segTop}\}$$

where *Pred* is the rule predecessor, *quadClipped* is the rule successor applied onto the clipped extruded quadrilateral (Figure 4b), *quadSupport* is the rule successor applied onto the joined face (Figure 4a) and *segTop* is the rule successor applied onto the extruded segment (Figure 4c). *heightExtrusion* is the height of the extrusion, while the target 2D segment is represented by *v1* and *v2*.

Therefore, the *Join* rule acts as a clipped extrusion rule guided by a 2D segment destination and a height. Moreover, unlike the *Extrude* rule, *Join* does not change the surface parametrization and prepares the clipping planes (Figures 4d-f). We provide built-in rule-scope and global-scope accessors to users directly into the grammar file, such as the position of the current primitive. For instance the rule-scope accessor *currentPos0()* returns the coordinates of the first vertex of the current 1D or 2D atom. Such accessors can be combined with the roof target parameters to generate specific roof structures.

3.3. Clipping

Clipping planes are automatically generated after a *Join* rule and applied to the extruded rectangular quadrilateral, oriented according to the bissector angles (Figures 4d-f).



(a) Scaled *Extrude* rule (b) *Join* rule

Figure 3: *Differences on surface parametrization.*

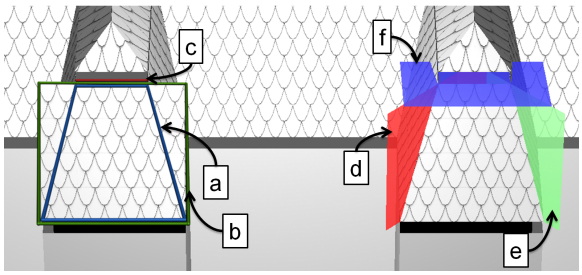


Figure 4: *The* Join *rule has three successors :* a) *a joined face,* b) *an extruded rectangular quadrilateral including the joined face, and* c) *the top segment defined by* v1 *and* v2 *at* heightExtrusion *added to starting height. Clipping planes are defined around the joined face and applied on the extruded quadrilateral. The left* d)*, right* e) *and top* f) *clipping planes are oriented according to the bissector angles.*

Adjacent terminal shapes are therefore consistently assembled. Patch clipping is partially performed at the rule expansion stage for the terminal shapes generated from a *Join* rule. Fully clipped terminal shapes are simply not generated. Partial clipping is achieved at terminal evaluation stage by applying an optimized clipping algorithm [McG11] within the geometry shader. This algorithm takes advantage of the SIMD capabilities of the graphics hardware to efficiently eliminate clipped triangles.

3.4. Project rule

During rule expansion one may want to project the current quadrilateral onto a chosen plane, for instance to create a balcony window as part of a roof section (Figure 5). While splitting the roof section, we can separate the roof part that will become the balcony. As this roof part is originally aligned with the roof surface, we project it onto the plane defined by input axes, typically the local tangent of the quadrilateral and the cross product between this local tangent and the normal of the building. The projected quadrilateral can then become the base of a balcony. We thus introduce the *Project* rule:

Pred→Project(vec3 axis1, vec3 axis2) {quadProjected}

where *Pred* is the rule predecessor and *quadProjected* is the rule successor applied onto the projected quad. The plane is defined by the vectors *axis*1 and *axis*2.

The height of projection is made available through the rule-scope built-in accessor *projectionHeigh()*. The tangent, binormal and normal of the input building are set at the beginning of the grammar evaluation, while the local base is updated throughout the rule expansion. This information is also available for the user through accessors such as *localTangent()* or *globalNormal()*. Users may thus easily set the desired projection.

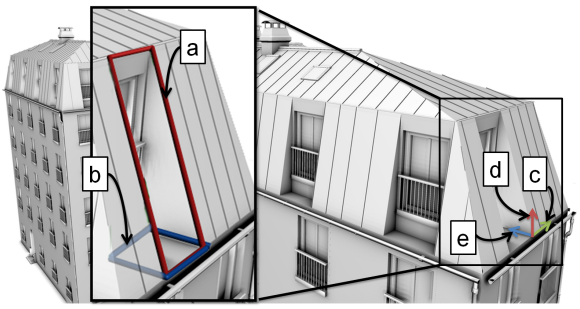


Figure 5: *Illustration of the* Project *rule. The blue quadrilateral* b) *is the projection of the red quadrilateral* a)*. The projection axes are the local tangent* c) *and the cross product* e) *between the local tangent and the global normal* d)*.*

4. Case study

The following rule sequence describes a roof with a balcony window, using *Join* and *Project* rules. This sequence corresponds to the first roof section of the roof part detailed in Figure 5.

```
TopWall     →Join(3, roofPos1, roofPos2)
                {RoofSection, NULL, MansardTop}
RoofSection →Split(X, 2, 1, ∼){RoofCov, Balcony, RoofCov}
Balcony     →Split(Y, 0.5, 2, ∼){RoofCov, BalconyPart, RoofCov}
BalconyPart →Project(localTangent(), cross(localTangent(),
                globalNormal())) {BalconyFlat}
BalconyFlat →Explode() {FloorShape, NULL, RightBalc, BackBalc,
                LeftBalc}
RightBalc   →Join(projectHeight(), currentPos1(), currentPos1())
                { WallBalcShape, NULL, NULL }
LeftBalc    →Join(projectHeight(), currentPos0(), currentPos0())
                { WallBalcShape, NULL, NULL }
BackBalc    →Join(projectHeight(), currentPos0(), currentPos1())
                { NULL, WindowShape, NULL }
RoofCov     →Repeat(XY, sizeTiles){TilesShape}
```

The hole created by the *Project* rule *BalconyPart* is entirely filled by reconstructing the missing parts using the *Join* rules *LeftBalc*, *RightBalc* and *BackBalc*. Note also the *Explode* rule that allows to call distinct successors on the current quadrilateral, and each of its four segments (front, right, back, left).

5. Applications and Results

As for GPU shape grammars, our approach for roof generation works with the Shader Model 5.0 (DirectX 11). We modeled various complex roofs structures featuring highly detailed geometries using *Join* and *Project* rules. Roofs are covered with different geometric shapes such as slates, flat or round tiles, using *Repeat* rules. We also modeled geometric roof elements that are essential to create compelling roofs (Figure 6).

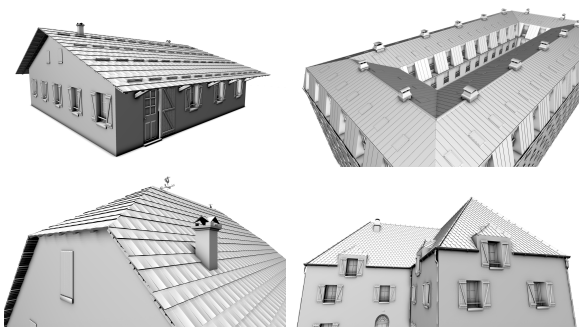


Figure 6: *Simple use of the grammar rules allows us to create dormer windows, flat windows, roof balconies, chimneys, ridge tiles, gutter, beams, and more.*

As for GPU shape grammars, roof parameters can be tuned interactively. We measured timings on object generation (rule expanding stage), and on terminal evaluation, clipping and rendering using a Nvidia GeForce GTX 480 GPU. We observed that buildings (facades and roofs) are expanded in sublinear complexities, confirming GPU shape grammars results. As we expand objects in parallel, we may generate many objects at a reduced cost. For instance expansion of 1 to 20 Normandy-type houses (80 segments) requires a constant time of 11.5 ms. The expansion of 21 to 40 houses (160 segments) then takes 22.3 ms. Terminal evaluation, clipping and rendering scale linearly, taking around 1.02 ms per input segment.

6. Discussion and Future Work

We experimented two strategies for geometric roof covering: the repetition of many terminals yielding a single tile each and the use of fewer terminals leading to batches of tiles (2×2 tiles). Rule expanding is thus faster in the second case. For instance, the halfway dormer roof of Figure 1 is expanded 1.92× faster using batches of tiles. Also, the terminal evaluation part runs 1.1× faster than using unit tiles. Hardware tessellator seems to prefer few denser meshes than many sparse meshes.

In counterpart, misalignment can occur in some specific cases. While the *Repeat* rule creates an even repartition across adjacent children, the *Split* rule may introduce irregularities when the splitting and repetition axes is not identical. The children are either scaled to fit the space, or translated to be aligned with the split. To limit these distortions we can automatically adapt the number of repetitions to the nearest integer part. Another solution would be to globally distribute the children and clip split parts.

Finally, the straight skeleton algorithm may lead to incoherent, unrealistic roof structures. In some cases, one input segment can be associated to multiple destination segments, currently not supported by GPU roof grammars. Nonetheless our approach can be associated to any other user-defined roof structure extractor. Future work will therefore consider new skeleton generation algorithms based on architectural rules and constraints. Such skeletons could be directly used as input of our GPU roof grammars pipeline to model highly detailed roofs.

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. Springer Verlag. 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. 1, 2

[KW11] KELLY T., WONKA P.: Interactive architectural modeling with procedural extrusions. In *Proceedings of SIGGRAPH* (2011), vol. 30, pp. 14:1–14:15. 1, 2

[LD03] LAYCOCK R. G., DAY A. M.: Automatically generating large urban environments based on the footprint data of buildings. In *Proceedings of ACM symposium on Solid modeling and applications* (2003), pp. 346–651. 2

[LWW08] LIPP M., WONKA P., WIMMER M.: Interactive visual editing of grammars for procedural architecture. In *Proceedings of SIGGRAPH* (2008), pp. 1–10. 1

[LWW10] LIPP M., WONKA P., WIMMER M.: Parallel generation of multiple l-systems. *Computers & Graphics 34*, 5 (Oct. 2010), 585–593. 1

[MBG*12] MARVIE J.-E., BURON C., GAUTRON P., HIRTZLIN P., SOURIMANT G.: GPU Shape Grammars. *Computer Graphics Forum 31*, 7 (2012), 2087–2095. 1, 2

[McG11] MCGUIRE M.: Efficient triangle and quadrilateral clipping within shaders. *Journal of Graphics, GPU, and Game Tools 15*, 4 (2011), 216–224. 3

[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

[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. 1

[MWH*06] MULLER P., WONKA P., HAEGLER S., ULMER A., GOOL L.: Procedural modeling of buildings. In *Proceedings of SIGGRAPH* (2006), pp. 614–623. 1, 2

[PM01] PARISH Y., MULLER P.: Procedural modeling of cities. In *Proceedings of SIGGRAPH* (2001), pp. 301–308. 1

[WWSR03] WONKA P., WIMMER M., SILLION F., RIBARSKY W.: Instant architecture. In *Proceedings of SIGGRAPH* (2003), pp. 669–677. 1