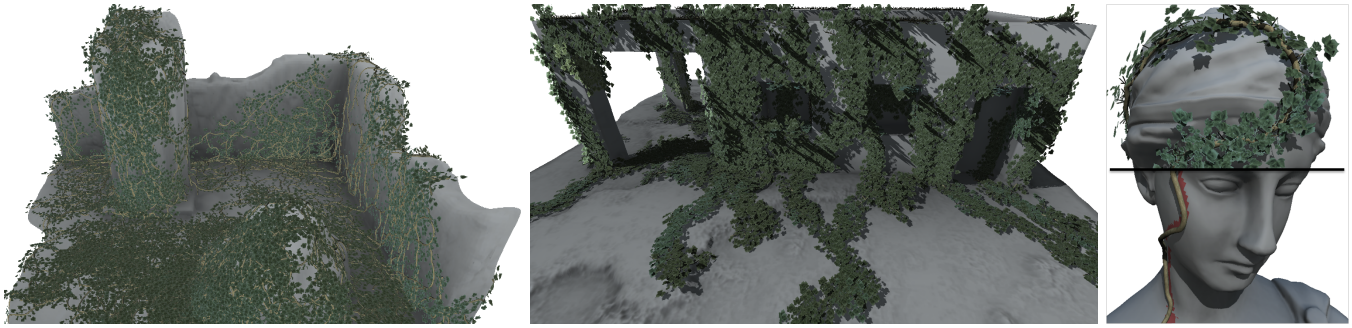
Dynamic On-Mesh Procedural Generation

Cyprien Buron[1]* Jean-Eudes Marvie[1]† Gaël Guennebaud[2]‡ Xavier Granier[2]§

[1]Technicolor [2]Univ. Bordeaux / INRIA / IOGS



(a) 4.48M polygons / 136ms (b) 8.84M polygons / 336ms (c) Painting constraint

Figure 1: (a,b) Two scenes generated and rendered with our system using on-mesh procedural extrusions. Given a base mesh and a procedural grammar of ivy growth, our GPU-based marching rule generated the ivy geometry on-the-fly in parallel with interactive performance. (c) The grammar expansion can be easily guided through a user-friendly painting interface.

Abstract

We present a method to synthesize procedural models with global structures, such as growth plants, on existing surfaces at interactive time. More generally, our approach extends shape grammars to enable context-sensitive procedural generation on the GPU. Central to our framework is the unified representation of external contexts as texture maps. These generic contexts can be spatially varying parameters controlling the grammar expansion through very fast texture fetches (*e.g.*, a density map). External contexts also include the shape of the underlying surface itself that we represent as a texture atlas of geometry images. Extrusion along the surface is then performed by a marching rule working in texture space using indirection pointers. We also introduce a lightweight deformation mechanism of the generated geometry maintaining a C^1 continuity between the terminal primitives while taking account for the shape and trajectory variations. Our method is entirely implemented on the GPU and it allows to dynamically generate highly detailed models on surfaces at interactive time. Finally, by combining marching rules and generic contexts, users can easily guide the growing process by directly painting on the surface with a live feedback of the generated model. This provides friendly editing in production environments.

Index Terms: 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)

*e-mail:cyprien.buron@technicolor.com
†e-mail:jean-eudes.marvie@technicolor.com
‡e-mail:gael.guennebaud@inria.fr
§e-mail:xavier.granier@inria.fr

1 Introduction

Synthesizing highly detailed 3D shapes on top of existing surfaces is a crucial step to enrich the visual complexity and realism of a scene. Manually creating such shapes is a very tedious task. Taking inspiration from 2D texture synthesis techniques, *geometric texture synthesis* methods [3, 4, 26] strive to automatically cover a given surface by stitching pieces extracted from a single 3D exemplar pattern. This involves an expensive process preventing interactive feedback, and making the control of the end result rather difficult. Even though the end result can be described in a compact manner, the need to explicitly generate the geometry offline might also lead to memory and resource management issues for very detailed models. Moreover, such approaches are limited to the generation of repetitive patterns, and complex global structures such as branching models as in Figure 1 cannot be automatically created.

Starting from a very lightweight representation, procedural modeling systems enable the generation of highly detailed objects through *amplification rules* [10, 21]. Grammar based procedural modeling is well suited to the creation of structured models. In the context of large urban scenes, recent works showed that the generation can be accomplished in real-time by exploiting the high parallelism and computation power of graphics hardware [11, 20]. However, none of these methods consider the underlying surface during the generation process. To this end, Li *et al.* [9] proposed to use tangent vector fields guiding the procedural shape generation on a given surface. Unfortunately, in their approach the generation is still an offline process thus impoverishing editing abilities and canceling the lightweight nature of a procedural representation.

In this paper, we present a *context-sensitive* shape grammars modeling system enabling the real-time generation of geometric patterns with global structures on top of an underlying surface. Unlike previous work [9, 26], our approach does not require a tangent field to be designed prior to the procedural evaluation. We rely on a parametrized mesh allowing for self-controlled generation. A painting tool permits to guide the expansion (*e.g.*, along paths) and spatially adjust rule parameters in an interactive manner. We address the respective challenges through the following technical contributions:

- We introduce a *marching* rule to consistently spread procedural models over arbitrary surfaces using an atlas of geometry images as an intermediate representation.
- We present an image based mechanism to query rule parameters and conditions from external contexts.
- We propose a smooth G^1 geometry synthesis mechanism of terminal shapes based on cubic Bézier interpolation.

In addition, our algorithm is GPU-ready, and can be easily implemented within existing parallel generation systems. Our GPU-based prototype implementation can generate in real-time highly detailed models composed of several millions of polygons such as the growth of the ivy plant in Figure 1. Such models can thus be edited interactively.

2 Related works

Example-based texture synthesis algorithms have been studied to add geometric details on surfaces by replicating an input swatch (*e.g.*, a small piece of geometry as polygonal mesh, volumetric texture or level sets) over a given surface [3, 4, 26]. When locally similar geometries need to be assembled together, a stitching step is performed by matching adjacent geometries and searching the min-cut to minimize the seams. As input shapes are periodic or nearly-periodic, the synthesized models exhibit patterns with stationary or local structures only.

On the other hand, patterns with global structures can be modeled using procedural approaches. Starting from a lightweight description of an object, amplification rules allow generation of a highly detailed model. Among procedural methods, L-systems [10, 15, 17] and shape grammars [14, 21, 25] have been widely used to generate vegetation or architectural models [24]. Context-free grammars have been extended to context-sensitivity in order to interact with exterior environments and adapt their behavior accordingly [13, 16, 18]. In addition, various approaches have been developped to constrain the grammar to fit a given structure using strokes, paints, guides or voxels [2, 7, 22]. All these procedural methods can generate geometry on the plane or in the 3D space but not on arbitrary surfaces. Another common drawback is the required high generation time, preventing interactive feedback for complex scenes.

Recent approaches take advantage of the GPU to perform parallel evaluation of grammar rules. Parallelism can be accomplished per pixel [6, 12], per seed [11], or per rule [20]. Coupled with level-of-details techniques, such methods achieve interactive generation and rendering for massive scenes and enable on-the-fly editing which is critical from a user perspective. Nonetheless, none of these approaches can handle grammar expansion on generic surfaces.

Some modeling software provide procedural generation tools such as Autodesk Maya PaintEffects [1]. The user is asked to paint on a surface where a hard-coded grammar has to be generated. Growth on surfaces is simulated by planting multiple seeds along a stroke path instead of really growing the initial seed. Then, the expansion of each seed is performed locally on the tangent plane without interaction with the real surface or nearby seeds. In the *Ivy Generator* tool [23], growth on surfaces is accomplished by testing all polygons of the underlying surface which is too expensive to aim for interactive performance.

To our knowledge, Li *et al.*'s work [9] is the only method doing a true growing of procedural models on an underlying surface. In their work, a predefined set of values and vector fields are used as contexts to guide the expansion of a shape grammar. For instance, those fields can drive the translation rule, select rules, or more generally serve as rule parameters. Even though this approach yields nice geometric patterns with a global structure, it also requires from seconds to minutes to generate a model.

3 Overview

Our approach takes as input a shape grammar to be grown on an arbitrary parametrized surface. We chose to base our approach on a parametrization rather directly marching on the given 3D mesh for several reasons. Firstly it provides us the ability to easily adjust the marching step size according to the desired quality regardless of the mesh triangulation (*i.e.*, we can easily jump over many triangles in one texture fetch). Decoupling the actual surface geometry and the geometry used as support for the marching is an essential ingredient enabling to adjust the smoothness of the support with respect to the grammar needs. As extensively detailed in previous works, a geometry image representation is also amenable for a very compact and efficient multi-resolution representation of the underlying surface (basically using mipmap levels). For instance, when growing a shape of a given width, all details smaller than a factor of this size are irrelevant and are ignored using a lower resolution level. Secondly, directly marching over a triangular mesh is significantly more expensive as it involves many incoherent memory fetches through the topological data structure, as well as intersection computations that have to be implemented carefully to avoid numerical issues when passing close to vertices or along edges. Thirdly, this makes our approach independent of the actual surface representation (triangular mesh, quad mesh, NURBS, subdivision surfaces, etc.). Finally, we emphasize that in production pipelines, meshes are almost always parametrized to support 2D texturing. Therefore, requiring a parametrized mesh as input is a rather small constraint in practice compared to the numerous benefits.

Our *context-sensitive* shape grammar react to *external contexts* which, in our case, lie on the underlying surface. We distinguish two kinds of external contexts: the shape of the underlying surface itself, and any other spatially varying information controlling the grammar expansion such as, for instance, density, scale, colors, etc. Those information can be either generated automatically (*e.g.*, based on surface curvature), or interactively by the user through direct on-mesh painting. In order to avoid huge distortions in the parametrization, we allow the surface to be decomposed into individually parametrized patches. Then, each patch is discretized into an image-based representation mapping any valid point of the image plane to a 3D point on the surface. This representation is detailed in Section 4. It will serve as the basis to both track the surface and to store external context information in a unified manner. The marching algorithm is described in Section 5. In particular, we show how to consistently and robustly walk across the different patches of the underlying surface during the grammar expansion. Then, we show in Section 6 how to instantiate and deform terminal shapes such that the generated geometry remains smooth while molding the shape of the underlying surface. Finally, we give some details of our implementation within GPU shape grammars in Section 7, and, as an example, we detail in Section 8 how our approach can be used on the specific case of ivy growth.

4 Texture-based external contexts

Our context-sensitive shape grammar uses external contexts lying on the given underlying surface. In order to ease their storage and usage within a GPU implementation, we propose storing them as texture maps. In this representation, shape grammar parameters correspond to classical texture maps, while the 3D surface corresponds to *geometry images* [5]. A geometry image is computed by rasterizing the 3D coordinates of the input surface into the texture plane such that each pixel stores the 3D coordinates of its respective surface point. Connecting the neighboring texels yields a quad mesh closely approximating the input surface. Following the work of Sander *et al.* [19], we limit distortions due to the parametrization using multi-chart geometry images. The input surface is decomposed into multiple flatter charts allowing for low parametric distortions. Charts are organized as a texture atlas (as in Figure 2b)

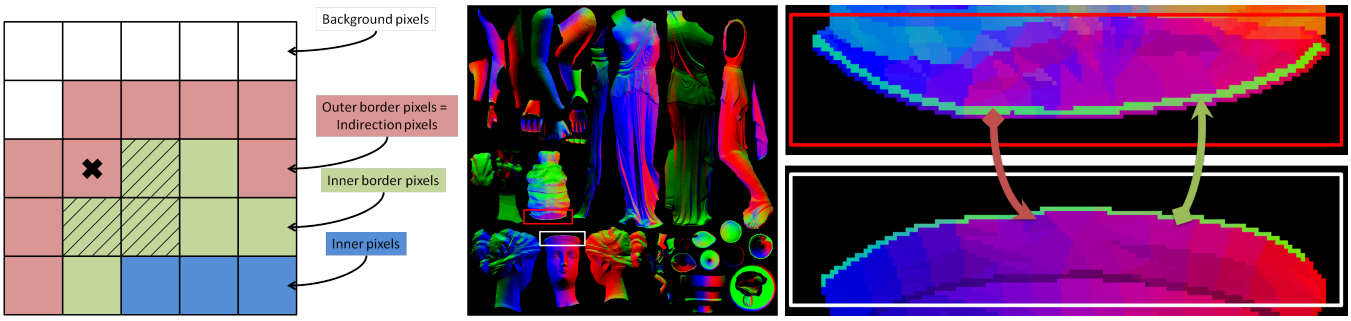


Figure 2: a) For the indirection pixel represented with a black cross, we compute the mean of neighbor inner pixels (dashed pixels) to find the closest inner border pixel in another chart. b) Indirection geometry image of the Hebe model, with a zoom on the top head part (c) showing pointers to jump from one side of the head to another. The inner border pixels of each side of the head are replicated as indirection pixels on the other side.

by optimizing space occupancy while ensuring that adjacent chart borders correctly match in the 3D space. We refer the reader to [19] for the details on the construction of multi-chart geometry images.

Inspired by the work of Lefebvre and Hoppe [8], we extend this representation with indirection pointers allowing to navigate in texture space from one chart to the adjacent one in constant time. To this end, charts must be organized in the atlas such that we have at least a one pixel band available around each chart to store indirection values.

Our technique to compute the indirection values is depicted in Figure 2a. We start from a multi-chart geometry image fulfilling the above criteria plus the associated chart IDs. For each *indirection pixel*, that is, for each pixel lying on the *outer-boundary* of a given chart, we compute its closest 3D surface position **p** by averaging the adjacent texels lying on the *inner-boundary* of the chart. Then we search within the inner-boundary texels of the other charts the pixel whose respective 3D surface point is the closest to **p**. The texture coordinates of the fetched pixel is finally stored in the indirection pixel. This search can be greatly accelerated by indexing the inner-boundary texels into a kd-tree. Figure 2b-c shows an indirection geometry image computed on the Hebe model decomposed in dozens of charts.

As our representation of external contexts is image-based, we make available the current texture coordinates *texCoord* to the scope of the grammar so that the grammar can exploit standard texture fetches to query spatially varying attributes at run-time. This provides the ability to constraint any rule at any time in a generic manner.

5 Marching on a Surface

Recall that given an initial seed generated on a surface, our general objective is to create geometric patterns on top of this surface. To perform this operation we introduce the *marching rule* which is responsible for performing extrusions along a surface context represented as an indirection multi-charts geometry image. We formally define this rule following the naming defined in Computer Generated Architecture (CGA) shape grammars [14]:

Pred→Marching(float length, float angle, bool condition)
{TopSucc, ExtrudedSucc}

where the marching is controlled by the *length* of the marching step in image space, the rotation *angle* of the current marching direction, and the *condition* which has to be satisfied at destination to apply the rule. *Pred* denotes the symbol triggering the rule, and *TopSucc*, *ExtrudedSucc* denote the translation and extrusion symbols respectively.

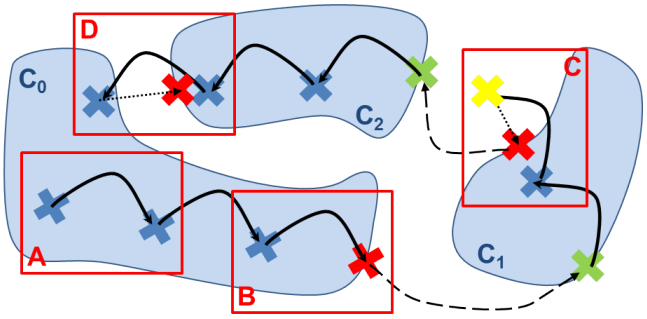


Figure 3: Illustration of the four cases which can occur during the marching process. Charts are indicated in blue (C_0, C_1, C_2) and fetched pixels are represented by crosses. Blue and yellow crosses correspond to inner and background pixels respectively, while a red cross represent an indirection pointer redirected to an inner pixel (green cross).

During the grammar expansion, the central problem is to find the destination pixel P_{dest} from the current pixel P_{cur} in the marching direction $\vec{d}$. As a first guess, we pick the destination pixel $\hat{P}_{dest}$ as $\hat{P}_{dest} = P_{cur} + \vec{d}$. Depending on P_{cur} and $\vec{d}$, the marching algorithm may encounter the following four different cases:

1. If $\hat{P}_{dest}$ is a valid inner pixel belonging to the same chart as P_{cur} (as in Fig. 3A), then no redirection is required and $P_{dest} = \hat{P}_{dest}$.

2. If $\hat{P}_{dest}$ is an indirection pixel (as in Fig. 3B), then we can directly jump to the referenced pixel and propagate the current marching direction: $P_{dest} = ref(\hat{P}_{dest})$.

3. If $\hat{P}_{dest}$ corresponds to an empty pixel (*i.e.*, a background pixel as in Fig. 3C), then a binary search is performed along the discrete segment $]P_{curr}, \hat{P}_{dest}[$ until we find the indirection pixel. After indirection, the marching is continued in the appropriate direction with the remaining step length.

4. If $\hat{P}_{dest}$ is a valid inner pixel belonging to a different chart (as in Fig. 3D), then we proceed as in case 3 considering every pixel having a different chart ID as background.

An example of marching on the Hebe geometry image is depicted in Figure 4, corresponding to the ivy scene generated in Figure 1c.

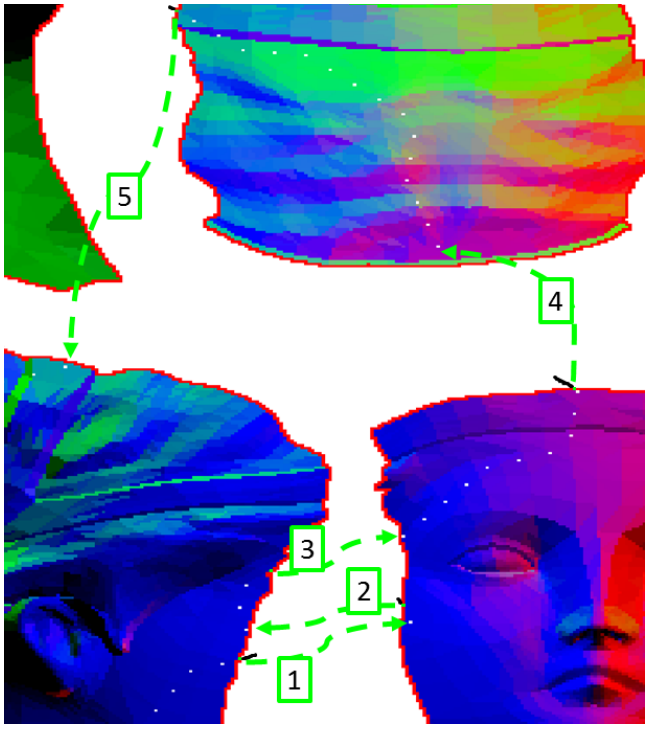


Figure 4: Example of the marching steps performed on the Hebe model to generate the ivy scene in Fig 1c. White dots indicate sampled pixels during marching, black ones correspond to a marching back, and green arrows are jumps across charts in marching order.

In the rest of this section we detail how to robustly find the best indirection pixels (Section 5.1), and how to consistently propagate the marching direction when jumping from one chart to another (Section 5.2).

5.1 Best indirection pixel

A given indirection pixel is usually fetched from various marching directions. However, depending on the marching direction, the referenced chart might not be the best one to continue the propagation, and a better indirection pixel may be found in the neighborhood of the initial indirection pixel. This is especially the case nearby chart corners. For instance, let us consider three charts sharing their edges at one corner. For each chart, the indirection pixel at the corner is unique and somewhat arbitrarily refer to one of the two adjacent charts. So the referenced charts may be appropriate for one particular marching direction but not for another one if it is directed to the non chosen chart.

As depicted in Figure 5, we propose addressing this issue by looking in the neighborhood to find a possibly better indirection pixel. Let P_0 be the initial indirection pixel obtained for the current pixel P_{cur} and 2D marching direction $\vec{d}$ as described before. We first establish a reference 3D marching direction $\mathbf{r}$ within the current chart as the normalized vector between the 3D position of the pixel P_{cur} and P_{near} where P_{near} is the nearest inner pixel to P_0. Formally, we set $\mathbf{r} = pos(P_{near}) - pos(P_{cur})$. Then, we consider the one-ring neighborhood of indirection pixels P_i around P_{near}. For each P_i, we fetch the 3D vertex position at the referenced pixel to compute a candidate 3D direction $\mathbf{c}_i = \overrightarrow{pos(ref(P_i)) - pos(P_{cur})}$. Finally we pick the indirection pixel minimizing the distortion, that is the one whose candidate direction $\mathbf{c}_i$ is the closest to the reference $\mathbf{r}$. If $P_{near} = P_{cur}$, then no reference direction can be computed and we keep P_0 as the best candidate.

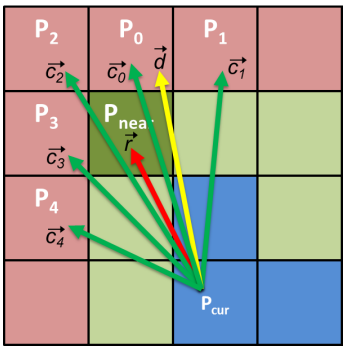


Figure 5: Selection of the best indirection pixel according to a 2D marching direction (yellow arrow), a reference 3D direction (red arrow) and multiple candidate 3D directions (green arrows).

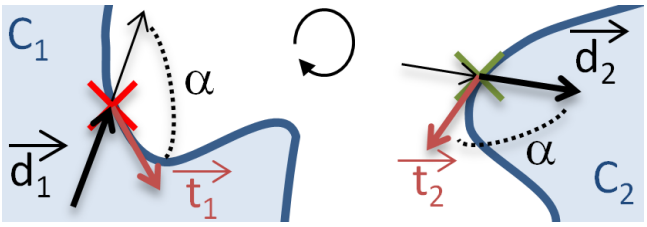


Figure 6: During a jump between two charts, the marching direction has to be rotated to balance for different chart orientations and parametric distortions. We use 2D tangent vectors estimated at source ($\vec{t}_1$) and destination ($\vec{t}_2$) charts to find the rotation which has to be applied to the marching direction.

5.2 Rotation of marching directions

As can be seen in the Hebe geometry image (*e.g.*, Figure 4, 5$^{\text{th}}$ redirection), because of the necessary rotations and deformations introduced during the unfolding process, adjacent chart boundaries are not necessarily aligned with each others. Consequently, when jumping from one chart to another within the geometry image, we have to adjust the marching direction such that it remains consistent in the 3D space. To this end, we define the marching direction $\vec{d}_1$ in the source chart as the angle α made with the 2D tangent $\vec{t}_1$ of the boundary of the chart. As illustrated in Figure 6, this relative angle can be safely propagated to the destination chart, and the new marching direction $\vec{d}_2$ can then be recovered from the local tangent boundary $\vec{t}_2$. The estimation of the tangent vectors is depicted in Figure 7. At the source chart, we start by scanning the one ring neighborhood of the current indirection pixel P_0 until we find the indirection pixel that follows an empty one. In our example, this is the pixel P_4^A. Then, in order to get smooth and reliable estimation, we then follow the K indirection pixels P_i and define $\vec{t}_1$ as the average of the vectors going from P_0 to the visited pixels P_i. At the destination chart, we apply the same algorithm to find the first inner border pixel (P_2^B in our example), and follow the K inner border pixels. From these two tangent vectors, we compute the rotation angle to be applied to the marching directions when jumping through P_0. In practice, these computations can be done at pre-processing time, and we only have to store the resulting rotation angle in the geometry images. Notice that a counterpart of this approach is that it does not allow for mirroring operations when optimizing chart placement.

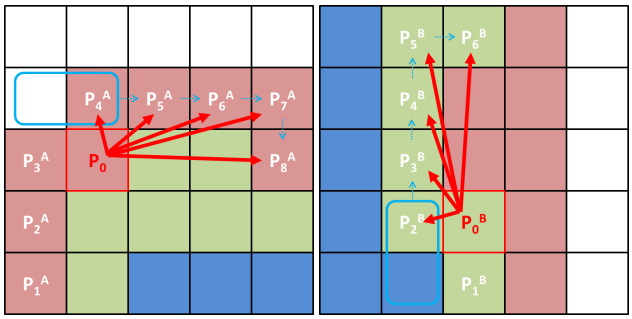


Figure 7: Illustration of the estimation of tangent vectors for the source (left) and destination (right) charts. Oriented tangents are estimated by scanning neighborhood boundaries (blue arrows) and averaging multiple estimated tangent directions (red arrows).

6 Smooth Terminal Generation

The result of the marching process is a set of polylines connected at branches. In our framework, geometric primitives are instantiated on a per segment basis. For disconnected terminal elements, for instance to spread pebbles, this can be realized independently in a naive way. In the more general case, for instance to generate continuous vine branches, we must deform the primitives to ensure a C^1 continuity between them. To this end, we first transform the polylines to G^1 parametric curves using cubic Bézier curves. Then, these curves will serve as a basis to smoothly deform the terminal geometries.

As smooth deformations of terminal shapes are not always desired, we add a parameter to the shape rule such that the grammar can control the smoothness along the trajectory (discontinuous, C^0, C^1). This section only describes the smoothest configuration as reducing the smoothness constraints is straightforward.

Cubic Bézier interpolation

Given a polyline defined by the sequence of 3D points $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \ldots$, our primary goal is thus to replace each segment $\mathbf{p}_i$, $\mathbf{p}_{i+1}$ by a cubic Bézier curve $B_i : \mathbb{R} \to \mathbb{R}^3$ defined by the control points $\mathbf{p}_i, \mathbf{b}_i^1, \mathbf{b}_i^2, \mathbf{p}_{i+1}$:

$$B_i(t) = (1-t)^3\mathbf{p}_i + 3(1-t)^2 t\mathbf{b}_i^1 + 3(1-t)t^2\mathbf{b}_i^2 + t^3\mathbf{p}_{i+1} \; .$$

The curves must ensure a G^1 continuity with the adjacent ones. Moreover, in order to make the curve follow the underlying surface, we further constraint it to be tangent to the underlying surface at the extremities. These two criteria are satisfied as soon as the three points $\mathbf{b}_{i-1}^2, \mathbf{p}_i, \mathbf{b}_i^1$ are aligned and lie in the tangent plane of the surface.

Our construction of the unknown control points $\mathbf{b}_i^k$ is illustrated in Figure 8. First, we estimate the tangent vector $\mathbf{t}_i$ of the curve at the point $\mathbf{p}_i$ as the projection onto the surface tangent plane of the average of the two adjacent segment directions:

$$\mathbf{t}_i = (\mathrm{I} - \mathbf{n}_i\mathbf{n}_i^T)\left(\frac{\mathbf{p}_i - \mathbf{p}_{i\text{-}1}}{\|\mathbf{p}_i - \mathbf{p}_{i\text{-}1}\|} + \frac{\mathbf{p}_{i+1} - \mathbf{p}_i}{\|\mathbf{p}_{i+1} - \mathbf{p}_i\|}\right) \; .$$

Here, $\mathbf{n}_i$ is the normal of the underlying surface at the point $\mathbf{p}_i$. Let $\hat{\mathbf{b}}_i^1$ be the point at the third of the segment $\mathbf{p}_i$, $\mathbf{p}_{i+1}$, that is: $\hat{\mathbf{b}}_i^1 = 2\mathbf{p}_i/3 + \mathbf{p}_{i+1}/3$. Then, the Bézier control point $\mathbf{b}_i^1$ is computed as the projection of $\hat{\mathbf{b}}_i^1$ onto the tangent line defined by $\mathbf{p}_i$, $\mathbf{t}_i$. After simplification, we get:

$$\mathbf{b}_i^1 = \mathbf{p}_i + \frac{1}{3}\frac{\mathbf{t}_i\mathbf{t}_i^T}{\|\mathbf{t}_i\|^2}(\mathbf{p}_{i+1} - \mathbf{p}_i) \; .$$

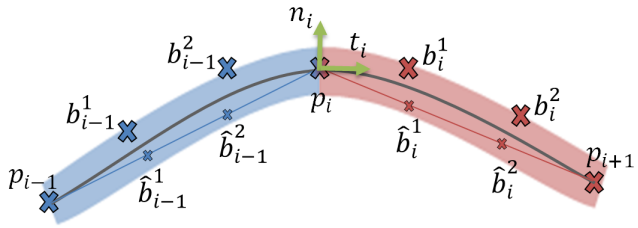


Figure 8: Two additional control points are computed for each generated segment, to perform geometric terminal smoothing based on Bezier cubic curves.

The other control points are computed analogously. This approach naturally generalizes to end-points and branches by defining the tangent vector $\mathbf{t}_i$ from the average over *all* adjacent segments.

Finally, the instantiated primitives are deformed by interpolating a local frame along the obtained cubic Bézier curves.

Parallel evaluation

During the grammar expansion, the Bézier control points have to be computed prior to the instantiation of geometric primitives over extrusion or marching elements. However, the construction of the i-th Bézier curve requires the tangent vector at the $\mathbf{p}_{i+1}$ point which itself requires the knowledge of the end-point $\mathbf{p}_{i+2}$ of the next element. This creates a forward dependency complexifying the parallel instantiation of the primitives. We address this issue using a depth-first traversal to evaluate all the points of a branch before computing the Bézier control points. This explains why the top successor *TopSucc* is evaluated before the extruded one *ExtrudedSucc* in our definition of the marching rule (Section 5). Then, the actual instantiation is realized in parallel per geometric primitives while providing a smooth G^1 continuity between adjacent ones.

7 Implementation

We implemented our on-mesh geometry synthesis algorithm within the GPU shape grammars pipeline [11]. This pipeline benefits from the highly parallel nature of recent graphics hardware to perform per-seed grammar development concurrently (see Figure 9). The CPU-based *rule compiler* step transforms the user-defined grammar (*i.e.*, the set of rules) to GPU compatible rule and parameter maps. Then, on the GPU side, a parallel *rule expander* step evaluates the rule map for each input seed to generate an intermediate lightweight structure of the desired object, that is, the set of abstract terminal elements. If both the grammar and parameters are static, then the result of the previous stage can be cached. Finally, as a second GPU stage, the *terminal evaluator* instantiates on-the-fly the actual geometry of terminal primitives.

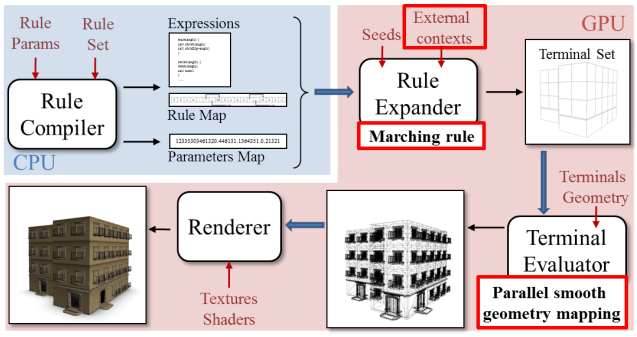


Figure 9: Integration of our marching and context-sensitive algorithm within the GPU shape grammars pipeline. New steps are indicated in red.

To support our context-sensitive shape grammar approach, three new steps are added to the previous pipeline. They correspond to the red boxes in Figure 9. Texture-based external contexts are added as resources to the *rule expander* so that context-sensitive grammars can be evaluated within the GPU. We implemented the marching rule within the parallel expanding kernel and we also compute the deformation parameters that will have to be applied to the terminal shapes (*i.e.*, normal and tangent information). With respect to the GPU shape grammars pipeline [11], this stage is still performed with one thread per seed. Finally, during the terminal evaluation, the generated information is used to correctly instantiate and deform the terminal shapes in parallel.

8 Application & Results

Ivy growth example

As a concrete example of our context-sensitive shape grammar and marching algorithm, we provide the following grammar simulating an ivy growth constrained by a binary mask painted on the input mesh:

IvyMainBranch(n) →Marching(0, context.mask(texCoord))
{IvyBranching(n), IvyGeometries}
IvyBranching(n) →Branch(2)
{IvySecBranch(0, rand(0,maxBranches)),
IvyMainBranch(n+1)}
IvySecBranch(n, k)→Marching(rand(-1,1)$\frac{\pi}{2}$, $n < k$)
{IvySecBranch(n+1, limitSecBranches),
IvyGeometries}

For the sake of brevity, we omitted the length displacement of the marching rule. This grammar is composed of two marching stages responsible to the creation of the main branch and secondary branches respectively. The first one goes straight, and it is activated only if the mask texture allows it (*i.e.*, $context.mask(texCoord)$). Recall, that here $texCoord$ refers to the 2D texture position of the marching rule destination. If the condition is not fulfilled for the current marching direction, an arbitrary range of directions sampled in the 2D semicircle around the main direction is used as marching direction candidates. The restriction to a 2D semicircle prevents infinite loops. If no compliant direction is found, the marching is stopped. For each secondary branch, a random number limits the branch length. Those branches are also oriented randomly around the main branch. Lastly, the *IvyGeometries* rule generates both ivy bark geometry and leaves on top of it. In the examples provided in this paper, the bark geometry is made of a detailed cylinder composed of 216 polygons to enable smooth deformations, while leaves are textured quads.

Figure 10 illustrates different possible use cases of the marching algorithm. For all cases, the user is asked to pick on the mesh locations where to start a seed. The marching direction is initialized from the difference of texture coordinates between the release and picking events. Then the grammar is fully evaluated in parallel over seeds at interactive time. By default, we let ivy seeds grow anywhere over the surfaces with recursive branching structures (Fig 10a). Spatial control over the generation is accomplished through texture contexts. Using a brush tool, the user can paint multi-valued contexts and have an interactive feedback on the generated geometry. For instance, in Figure 10b, the user painted a mask describing forbidden growing areas. This is accomplished by setting the *condition* parameter of the marching rule to check the context at destination and stop the expansion or find another path if the area is restricted. Full control over the growth is also possible by starting from an empty mask, and enabling the marching rules only in painted areas as in Figure 10c. The user paints on the mesh the growing path starting from the seeds, and the ivy models grow on-the-fly during the painting. Internally, the sampling of marching directions tries to find a proper direction to follow the growing

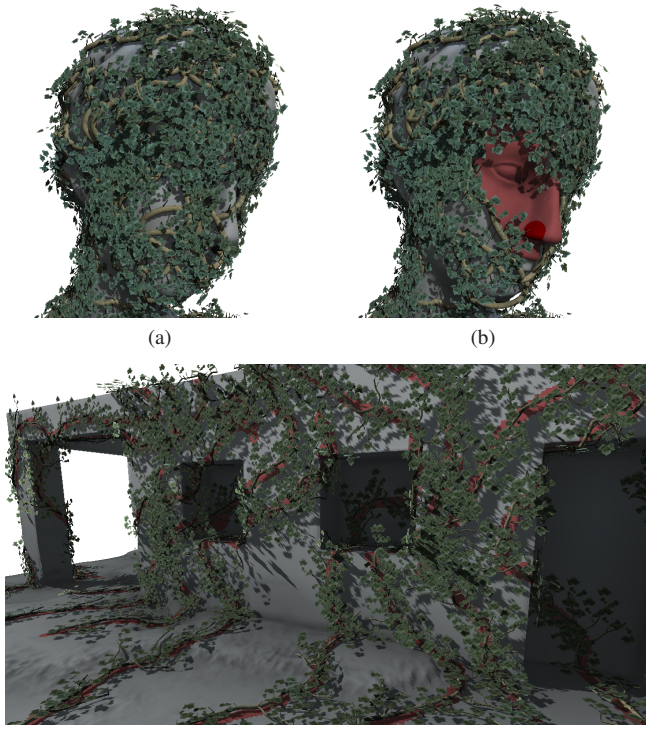


Figure 10: (a) Recursive branching without control. (b) A binary mask is used as an external context to specify forbidden areas. (c) The growth direction is guided through user-friendly painting. In (b) and (c), user painted areas are shown in red.

zone. In addition, smooth geometry deformations are observed for the generated models.

Performance evaluation

We tested our method on multiple scenes of high polygonal complexity. The table 1 details the pre-processing time of 1024×1024 indirection geometry images, including pointers computation, for three different input meshes. The indirection map for the Hebe model is slower to generate than for the two other models because this model is decomposed in many more charts.

	Hebe (fig. 1c)	Ruins (fig. 1a)	Shack (fig. 1b)
# polygons	64K	164K	140K
Generation time (s)	1	0.54	0.73

Table 1: Generation time of various indirection geometry images.

Figure 11 reports the generation time as a function of the number of seeds for two scenarios using a Nvidia GeForce 480 GTX. In the first case (blue curve), only the main branch of an ivy plant is created and the growth direction is guided by painting, while the second case also includes the generation of the secondary branches and leaves. In both cases the reported timings concur with the sublinear complexities observed in [11]. This means that multiple seeds can be generated at a reduced cost. Indeed we notice some plateau according to the input number of seeds. For instance, the same generation time is required from 10 to 42 seeds, and the next plateau begins at around 48 seeds. The slopes correspond to the cost of grid filling on the GPU before stabilization. Our method thus preserves

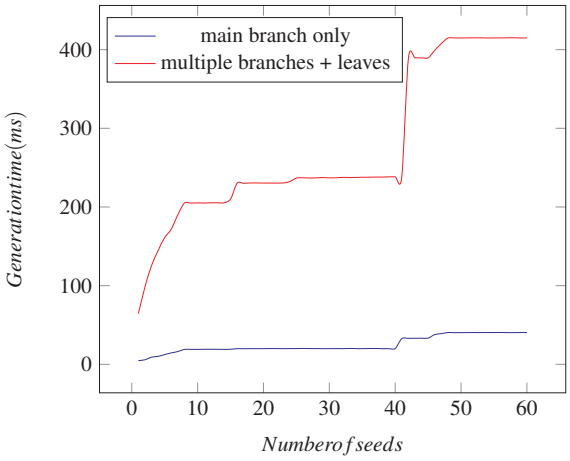


Figure 11: Generation time as a function of the number of seeds. The plateau reveal the high parallelism of our system.

the per-seed parallelization of the GPU shape grammar pipeline. Table 2 reports the generation time corresponding to the different figures of this paper. The scenes in Figure 1b and 10c are built using painted constraints while the one in Figure 1a is a recursive branching structure. The tight constraints and the associated sampling of marching directions in the first two models slow-down the grammar evaluation. According to the previous plateau observation, even more seeds could be generated at the same generation cost.

Model	Fig 1b	Fig 10c	Fig 1a
# of seeds	18	10	25
generation time (ms)	336	257	136
output # of polygons	8.84M	2.5M	4.48M

Table 2: Generation time and complexity for the scenes presented in this paper.

9 Discussion

Our context-sensitive shape grammar seamlessly extends CGA shape grammars [14] by a marching rule enabling on-surface extrusion. Our approach thus provides an ease of writing which is essential for users, as demonstrated by our ivy growth grammar example.

Using textures as external context information allows spatial control through simple user interactions. Even though we only showed binary mask examples, we could also imagine using a scalar valued texture controlling the density as a space occupancy probability, we could guide the marching direction through vector fields [9], or even encode the surface curvature as extra constraints. Possibilities are only limited by the creativity of users and grammar writers.

Our image-based marching approach relies on a given parametrization of the surface. However, it is rarely possible to compute an isometric parametrization, thus some distortions have to be expected. These distortions are of the exact same order as when mapping a standard 2D texture mesh. Moreover, standard atlas generation tools might generate non-convex patches in the geometry image. In both cases, since our algorithm supports texture atlas, it is as simple as splitting the patches to both reduce distortions and guarantee that the patch contours are convex.

The generated geometry is deformed with respect to the surface normals taken at the points sampled by the marching rule. If the marching step is too large, then the generated geometry may intersect the underlying surface, especially at highly curved area. This problem can be limited by adapting the sampling with respect to a curvature map. As the marching step size can be modified at any stage of the grammar derivation, our method implicitly adapts to curvature changes of both the surface (using the curvature context of the mesh) and the trajectory (built-in the grammar). One could either adjust the step size, or adopt an adaptive subdivision scheme according to the curvature context.

Our method fits well within the GPU shape grammars pipeline [11], thus allowing for interactive performance. Additionnaly, our marching-based approach is generic enough to be implemented within other recent parallel generation pipelines [20]. However, as seeds are evaluated in parallel, handling self-collisions becomes very difficult. For instance, tracking the visited areas into a texture map during the marching would lead to inconsistent behaviors between successive frames because of thread concurrency. The first thread writing at a given position at time t, may not be the same at time $t+1$.

10 Conclusion

In this paper we showed how to efficiently synthesize highly detailed geometries on top of existing surfaces. Using a grammar-based procedural modeling approach, we introduced a GPU compatible marching algorithm performing extrusions along surfaces represented as indirection geometry images. Additionally, we showed how generic information on those surfaces can also be used as textures to easily constrain and control the grammar expansion. Finally, we proposed a C^1 geometry synthesis step to instantiate the terminal geometries smoothly.

Our method is easily integrable within GPU-based procedural generation pipelines. It allows generation of complex global structures on existing surfaces at interactive time, and thus enhances the visual complexity of a scene. Furthermore, generic image constraints may be associated to grammar rules to provide user-friendly editing in production environment. For instance, one could generate on-the-fly a scene with multiple growing ivy seeds guided with painting, while having an interactive feedback during painting session.

Acknowledgements

The authors wish to thank Gaël Sourimant for the demo scenes and the editing of the video accompanying this paper.

References

[1] Autodesk®: Maya®. http://www.autodesk.fr/products/autodesk-maya, 2013.

[2] B. Beneš, O. Šťava, R. Měch, and G. Miller. Guided procedural modeling. In *Computer graphics forum*, volume 30, pages 325–334. Wiley Online Library, 2011.

[3] P. Bhat, S. Ingram, and G. Turk. Geometric texture synthesis by example. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 41–44. ACM, 2004.

[4] A. Brodersen, K. Museth, S. Porumbescu, and B. Budge. Geometric texturing using level sets. *Visualization and Computer Graphics, IEEE Transactions on*, 14(2):277–288, 2008.

[5] X. Gu, S. J. Gortler, and H. Hoppe. Geometry images. In *ACM Transactions on Graphics (TOG)*, volume 21, pages 355–361. ACM, 2002.

[6] S. Haegler, P. Wonka, S. M. Arisona, L. Van Gool, and P. Müller. Grammar-based encoding of facades. In *Computer Graphics Forum*, volume 29, pages 1479–1487. Wiley Online Library, 2010.

[7] T. Ijiri, S. Owada, and T. Igarashi. The sketch l-system: Global control of tree modeling using free-form strokes. In *Smart Graphics*, volume 4073 of *Lecture Notes in Computer Science*, pages 138–146. Springer Berlin Heidelberg, 2006.

[8] S. Lefebvre and H. Hoppe. Appearance-space texture synthesis. In *ACM Transactions on Graphics (TOG)*, volume 25, pages 541–548. ACM, 2006.

[9] Y. Li, F. Bao, E. Zhang, Y. Kobayashi, and P. Wonka. Geometry synthesis on surfaces using field-guided shape grammars. *Visualization and Computer Graphics, IEEE Transactions on*, 17(2):231–243, 2011.

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

[11] J.-E. Marvie, C. Buron, P. Gautron, P. Hirtzlin, and G. Sourimant. Gpu shape grammars. In *Computer Graphics Forum*, volume 31, pages 2087–2095. Wiley Online Library, 2012.

[12] J.-E. Marvie, P. Gautron, P. Hirtzlin, and G. Sourimant. Render-time procedural per-pixel geometry generation. In *Proceedings of Graphics Interface 2011*, pages 167–174. Canadian Human-Computer Communications Society, 2011.

[13] R. Měch and P. Prusinkiewicz. Visual models of plants interacting with their environment. In *Proceedings of ACM SIGGRAPH 1996*, pages 397–410. ACM, 1996.

[14] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool. *Procedural modeling of buildings*, volume 25. ACM, 2006.

[15] Y. I. Parish and P. Müller. Procedural modeling of cities. In *Proceedings of ACM SIGGRAPH 2001*, pages 301–308. ACM, 2001.

[16] P. Prusinkiewicz, M. James, and R. Měch. Synthetic topiary. In *Proceedings of ACM SIGGRAPH 1994*, pages 351–358. ACM, 1994.

[17] P. Prusinkiewicz, A. Lindenmayer, J. S. Hanan, F. D. Fracchia, D. R. Fowler, M. J. de Boer, and L. Mercer. *The algorithmic beauty of plants*. The virtual laboratory. Springer-Verlag, 1990.

[18] P. Prusinkiewicz, L. Mündermann, R. Karwowski, and B. Lane. The use of positional information in the modeling of plants. In *Proceedings of ACM SIGGRAPH 2001*, pages 289–300. ACM, 2001.

[19] P. V. Sander, Z. J. Wood, S. J. Gortler, J. Snyder, and H. Hoppe. Multi-chart geometry images. In *Proceedings of the 2003 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, SGP '03, pages 146–155. Eurographics Association, 2003.

[20] M. Steinberger, M. Kenzel, B. Kainz, J. Müller, W. Peter, and D. Schmalstieg. Parallel generation of architecture on the gpu. In *Computer Graphics Forum*, volume 33, pages 73–82. Wiley Online Library, 2014.

[21] G. Stiny and J. Gips. Shape grammars and the generative specification of painting and sculpture. In *IFIP Congress*, pages 1460–1465, 1971.

[22] J. O. Talton, Y. Lou, S. Lesser, J. Duke, R. Měch, and V. Koltun. Metropolis procedural modeling. *ACM Transactions on Graphics (TOG)*, 30(2):11, 2011.

[23] Thomas luft: Ivy generator. `http://graphics.uni-konstanz.de/~luft/ivy_generator`, 2007.

[24] C. A. Vanegas, D. G. Aliaga, P. Wonka, P. Müller, P. Waddell, and B. Watson. Modelling the appearance and behaviour of urban spaces. *Computer Graphics Forum*, 29(1):25–42, 2010.

[25] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky. Instant architecture. *ACM Transactions on Graphics (TOG), Proceedings of ACM SIGGRAPH 2003*, 22(3):669–677, 2003.

[26] K. Zhou, X. Huang, X. Wang, Y. Tong, M. Desbrun, B. Guo, and H.-Y. Shum. Mesh quilting for geometric texture synthesis. In *ACM Transactions on Graphics (TOG)*, volume 25, pages 690–697. ACM, 2006.