Component-Based Model Synthesis for Low Polygonal Models

Nicolas Maréchal*
Université de Lyon, CNRS
Université Lyon 1, LIRIS,
UMR5205, F-69622, France

Éric Galin†
Université de Lyon, CNRS
Université Lyon 2, LIRIS,
UMR5205, F-69676, France

Éric Guérin‡
Université de Lyon, CNRS
Université Lyon 1, LIRIS,
UMR5205, F-69622, France

Samir Akkouche§
Université de Lyon, CNRS
Université Lyon 1, LIRIS,
UMR5205, F-69622, France

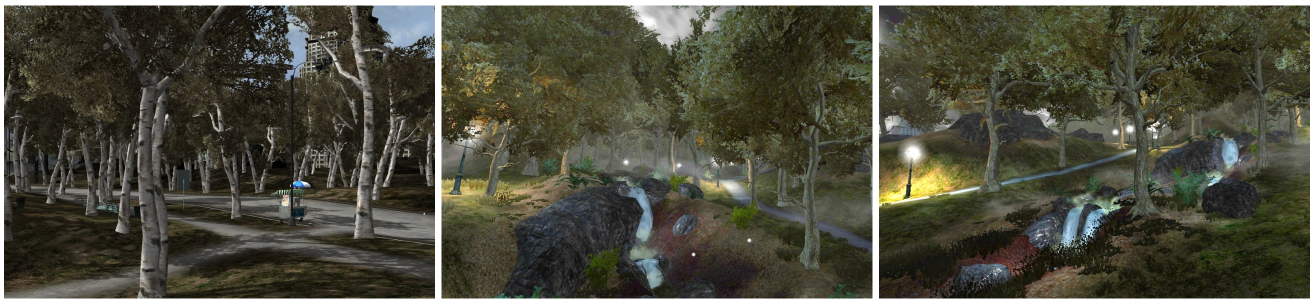

Figure 1: An area of Central Park with trees generated with our method from an initial model.

Abstract

This paper presents a method for semi-automatically generating a variety of different objects from an initial low polygonal model. Our approach aims at generating large sets of models with small variants with a view to avoiding instance replications which produce unrealistic repetitive patterns. The generation process consists in decomposing the initial object into a set of components. Their geometry and texture are edited and the modified components are then combined together to create a large set of varying models. Our method has been implemented in the *Twilight 2* development framework of Eden Games and Widescreen Games and successfully experimented on different types of models.

Keywords: Example-based modeling, components, procedural, variety

Index Terms: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—

1 Introduction

With the rapid growth of computing power, the demand for more realistic sceneries has increased considerably. Modeling complex landscapes is an important problem in computer graphics. The challenge stems not only from the complexity of the geometry and texture of the shapes (including the many surface details such as cracks, erosion or patina produced by aging and weathering) but also from the diversity of objects in a scene. Instantiation techniques that are used to handle memory demanding scenes often lead to unrealistic replication effects which reveals the synthetic nature of the scene. Thus, there is a need for techniques that automatically generate variations of objects.

Several techniques for adding details [3] and generating defects on objects [33] have been proposed and play an important part in the overall realism of a natural scene. Procedural generation techniques have been successfully developed for generating variations of terrains [19], plants [24, 22] and buildings or cities [18, 4]. Those methods suffer from several limitations. In general, procedural generation methods rely on numerous and complex rules which are difficult to control. Most existing techniques are specific to a restricted category of objects. It is difficult to conform to a given artistic style desired by an artist, which is essential in the entertainment industry.

In contrast, example-based approaches [21, 29] have been proposed for generating a variety of models from sample images or objects. Most existing techniques produce high resolution models with many triangles, in particular when blending or connecting the differents parts of the original models into a seamless topologically consistent mesh [12].

In this paper, we focus on low polygonal objects that are used in video games development like the ones generated by SpeedTree®[1] or using retopology techniques as in 3D-Coat[2]. The amount of VRAM memory available on architectures like Xbox 360 and Playstation® 3 is relatively small (512MB and 256MB respectivelly), which prevents artists from relying on highly detailed procedural generation methods.

Therefore, we propose an original technique suited for video games designers that semi-automatically synthesizes a vast variety of objects from an initial textured model. Our method proceeds in three steps. The initial model is first decomposed into a set of components. The geometry and texture of the components are then edited to create a larger set of modified components, which are eventually combined to synthesize a variety of final models. The geometry generation process consists in cutting the initial object into components which are edited and modified to create an atlas of geometry. The texture generation process consists in adding texture details to the original texture of the object. Those details are compacted into an atlas of texture components and may be combined to create a large variety of textures.

The main contributions of our paper are as follows. We present a generic technique for semi-automatically synthesizing hundreds or thousands of different objects from an initial low polygonal model of arbitrary type, geometry, topology and texture (Figure 4, 14) with

*e-mail: nicolas.marechal@liris.cnrs.fr
†e-mail: eric.galin@liris.cnrs.fr
‡e-mail: eric.guerin@liris.cnrs.fr
§e-mail: samir.akkouche@liris.cnrs.fr

[1]http://www.speedtree.com/
[2]http://www.3d-coat.com/

a very small memory overhead. Our method can be applied to any kind of model and does not depend on the technique used to produce the original object as in [6]. Our component decomposition and editing steps guarantees that modified components will continuously and seamlessly match during the assembly process. Our atlas texture generation and compacting process also guarantees that the texture components will seamlessly match, and avoids the modification of the parameterization of the objects.

Note that we did not investigate a method for automating the decomposition of any kind of model into components. The main reason for this is that the way objects are decomposed into components is not only sensitive to the geometry and texture of the object, but also depends on its very nature. Moreover, artists demand to have a very tight control over the overall process.

Because of its efficiency and the small memory overhead needed to synthesize thousands of objects, our method can be used in applications where memory constraints are very strong. This is the case in particular in the video game industry that needs techniques for generating and rendering variations of objects with coarse tessellation in real time.

The remainder of this paper is organized as follows. In Section 2, we present an overview of related work. In Section 3, we present geometric and texture characteristics of video games objects. In Section 4 and 5, we detail our algorithms for generating geometry and texture components. Eventually, we present our results in Section 6 before concluding in Section 7.

2 Related work

Several approaches exist for generating variations of objects. Our work relates to several research fields including procedural modeling of geometry and example-based geometric modeling.

2.1 Procedural generation techniques

Grammar-based generation techniques such as L-Systems [24, 20, 28, 22] and geometric rule-based methods [30] have been successfully used to generate plants and simulate their growth. More recently, grammar-based techniques [18, 32, 16, 23] as well as sketching [5] methods have been used for creating buildings and cities.

While those methods provide some realism and editability, they require some expertise for effective use. In particular they require considerable effort to replicate unexpected local structural modifications or to create a specific desired shape. Although those methods can synthesize a vast variety of realistic objects, every different model is represented individually which is memory demanding and impractical for creating and rendering large sceneries with hundreds or thousands of different models in real time. Approximate instancing [9] and stochastic simplification of aggregate details [7] can be used to simplify large scenes while preserving the overall appearance. Alternatively, large sceneries with hundreds of thousands of plant instances can be generated by using aperiodic tiling so as to avoid repetition and aliasing artifacts [10].

2.2 Example-based synthesis

Those methods [29, 21] aim at creating a variety of shapes from a set of initial argument models. Interpolation techniques and morphing have been successfully used to create variations [27, 14, 1]. The challenge is to finely tune the correspondences between the initial and final objects so as to generate geometrically consistent interpolated models, which is impractical when applied to objects with a completely different or very complex geometry such as trees. Moreover, models generated by such interpolation methods are not adapted to a low polygon context and cannot be instantiated.

Funkhouser *et al.* [12] proposed a method for creating new 3D objects from parts of previously existing ones stored in a database. Parts are obtained by cutting objects with given user strokes. A contour is found and can be refined by new user strokes if necessary. Object parts are identified with the help of the mesh connectivity. A new object is obtained by assembling parts with fillets between parts contour and eventually smoothing them to have a seamless connection.

When dealing with low polygonal models used in video games, automatically cutting objects into parts is a very complex task because of the lack of connectivity information between triangles (Figure 2). Moreover, to keep instantiating parts possible and to have a low memory overhead, we cannot join parts by adding a smoothing fillet.

Merrell [17] recently proposed a procedural method for generating different geometric models from an initial model based on example-based texture synthesis techniques [31, 11]. The method consists in cutting the initial model into pieces by a three dimensional grid, and then assembling them into different arrangements while preserving continuity in the matching process. Because of the decomposition along a three dimensional grid, this method is restricted to a limited set of objects.

3 Characteristics of video games objects

Real-time rendering combined with low memory constraints of video games consoles play an important part in the way objects are created by artists. This section presents the geometry and texture constraints encountered with low polygonal objects.

3.1 Low polygonal models

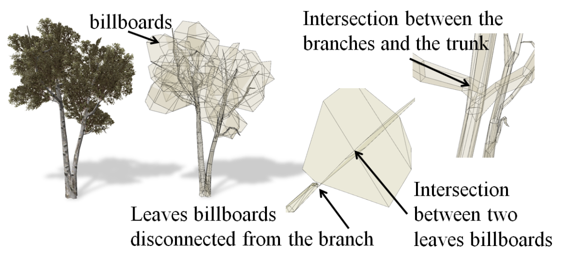
billboards
Intersection between the branches and the trunk
Intersection between two leaves billboards
Leaves billboards disconnected from the branch

Figure 2: A textured and wireframe view of a birch (745 vertices and 628 faces) and a closeup showing connectivity problems.

In video games, large scenes must be rendered in real-time with hundreds of objects and low memory capacity. Therefore, objects designed by artists must be created with a reduced number of vertices and faces. For example, the birch in Figure 2 is composed of only 745 vertices and 628 faces including several billboards representing the leaves. To reach this low polygons number, artists often model objects as a polygon soup or as a set of disconnected and self intersecting small meshes (Figure 2). Artists do spend an enormous amount of time carefully modeling and texturing the object to get beautiful results without visual artefacts.

Moreover, modeling objects in this way makes it very difficult to automatically decompose them into components by using traditional segmentation methods [2] because of connectivity and self intersection problems. In Section 4, we present a framework adapted to this kind of objects.

3.2 Texture repetition

Because of the low memory constraint, artists rely on small texture tiles to texture an entire object. Thus, the texture is repeated several times on the object (tiling). Figure 3 shows a branch textured with a small bark texture tile and the corresponding texture coordinates. This is performed by using (u, v) texture coordinates beyond the unity interval $[0, 1]^2$, which enable the artist to seamlessly texture a surface with a small tileable texture image (although at the cost of some visible repetitive patterns).

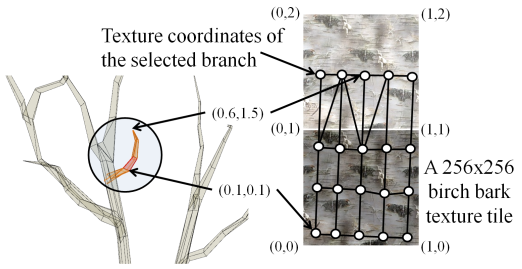
(0,2)
(1,2)
Texture coordinates of the selected branch
(0.6,1.5)
(0,1)
(1,1)
A 256x256 birch bark texture tile
(0.1,0.1)
(0,0)
(1,0)

Figure 3: Repetition of a texture tile.

Since the amount of information that can be carried by vertices is limited, it is crucial to keep and use the same texture coordinates as the ones provided by the artist when editing the texture. In Section 5, we propose a technique that enables us to avoid repetitive patterns while preserving the existing texture coordinates onto the object.

4 Generation of geometry variety

The component-based generation of geometry variety consists of three steps: first we decompose the input model into components, then we edit and modify them to create an atlas of components and finally we combine the modified components to create new variants of the initial model (Figure 4). Let n denote the number of components composing the initial object. Our system helps the user to decompose a given initial textured mesh O into components denoted F_i such that:

$$O = \bigcup_{i=1}^{n} F_i$$

The components F_i are modified to create variety components sets stored into an atlas. We denote F_i^k the k^{th} modification of F_i and $\mathcal{F}_i$ the whole set of models. The atlas $\mathcal{F}$ stores the components variations $\mathcal{F}_i$ and allows us to generate a set of appearance varying objects $\mathcal{V}$ by assembling components F_i^k. An object $\tilde{O} \in \mathcal{V}$ is defined by:

$$\tilde{O} = \bigcup_{i=1}^{n} F_i^k$$

Let $\#\mathcal{F}_i$ denote the number of components contained in the set $\mathcal{F}_i$. Our approach allow us to generate a large number of varying objects with a reduced number of components.

$$\#\mathcal{V} = \prod_{i=1}^{n} \#\mathcal{F}_i \qquad (1)$$

4.1 Decomposition into components

The decomposition step splits an initial textured mesh object O into components F_i. A component is a piece of the initial object obtained by cutting. Our technique allows the use of any cutting method. In our implementation, the user can choose between two methods (Figure 5):

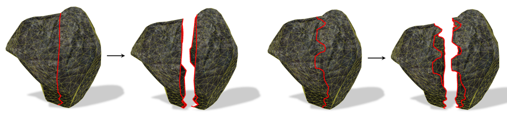

Figure 5: Cutting path which adds vertices (left) and using existing vertices (right).

- In the general case, the user draws a path on the surface of the object. This method adds new vertices along existing edges whose texture coordinates are obtained by linear interpolation of the texture coordinates of the cut faces vertices.

- Whenever the number of vertices is limited by a real-time rendering constraint, the best solution is to define a path by selecting existing vertices, avoiding the creation of new ones. We apply this method in the context of objects from video games.

Let R_{ij} denote the common contour that defines the connecting region between two components F_i and F_j obtained by a cutting operation. Figure 6 illustrates the cutting of a birch into two components F_i and F_j and their common connecting region R_{ij}. During the decomposition of O, we construct a graph $\mathcal{G}$ associated

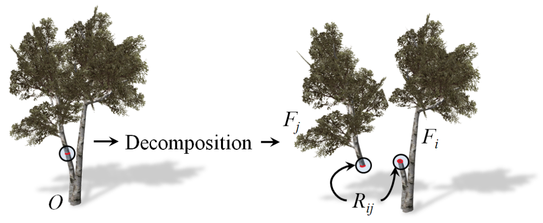

Figure 6: Birch cut into two components.

to O and denoted $G = (\mathcal{N}, \mathcal{A})$ where $\mathcal{N} = \{N_i\}$ denotes the set of nodes of the graph $\mathcal{G}$ and $\mathcal{A}$ refers to the arcs between two nodes (Figure 7). After the decomposition step, the nodes N_i are initialized with a single component F_i. An arc A_{ij} is created between two nodes N_i and N_j when their components F_i and F_j have a common connecting region R_{ij}.

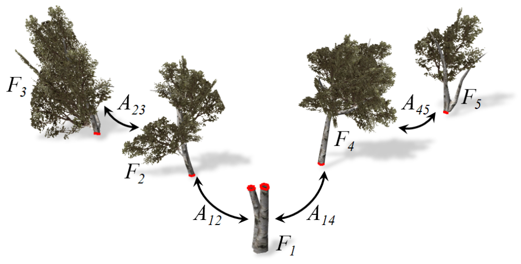

Figure 7: Graph with the birch cut into five components.

4.2 Atlas of components creation

This atlas of modified components is defined by editing the mesh of the different components F_i, introducing variety in components denoted $\mathcal{F}_i = \{F_i^k\}$. In our system, we have implemented several mesh editing techniques including Free Form Deformation [26, 8] and Axial Deformation [13]. We also provide low level editing modes, such as vertex editing which are essentiel for editing low polygon models in the context of video game. At this stage in the modification, all the component variants from $\mathcal{F}_i$ contained in N_i must still be connected to all the component variants from $\mathcal{F}_j$ contained in N_j when those nodes are connected in the graph $\mathcal{G}$ and conversely.

When the connecting region R_{ij} is affected by a modification of the component F_i and/or F_j, the graph $\mathcal{G}$ is updated to guarantee texture and geometry continuity between components of nodes N_i

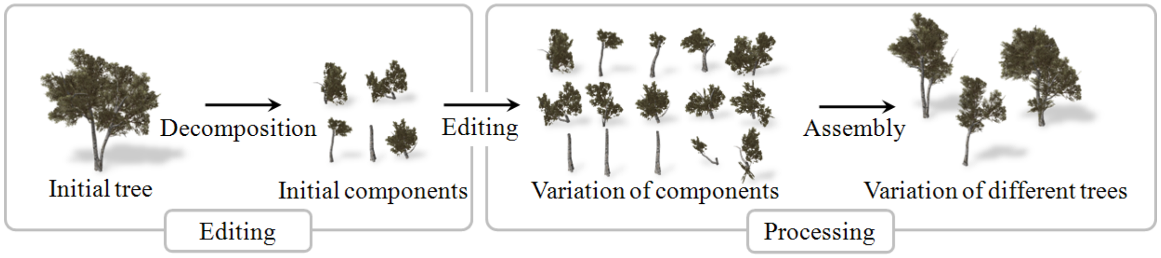

Figure 4: Synthetic overview of the geometry generation process.

and N_j. Depending on the area where the modification is applied, the graph is updated as follows.

4.2.1 Local modification of a component

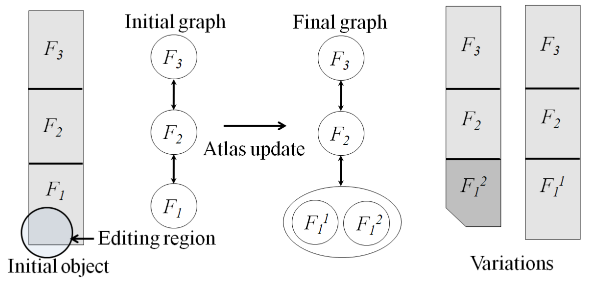

Figure 8: Atlas update when connecting regions are not affected.

When the modification of the component F_i does not affect the vertices of its connecting region, a new component is added to the node N_i of $\mathcal{G}$ (Figure 8).

4.2.2 Deformations preserving a connecting region

A powerful way to create significant shape variations consists in locally deforming two components F_i and F_j around their connecting region R_{ij} while preserving the shape of connecting region R_{ij}. Preserving the connecting region guarantees that all the variations of components in N_i and N_j obtained from F_i and F_j will connect seamlessly. Thus, we avoid atlas splitting and maximize the combinatory. This is particularly useful for generating variations of branching shapes, such as trees (Figure 9) or pipes.

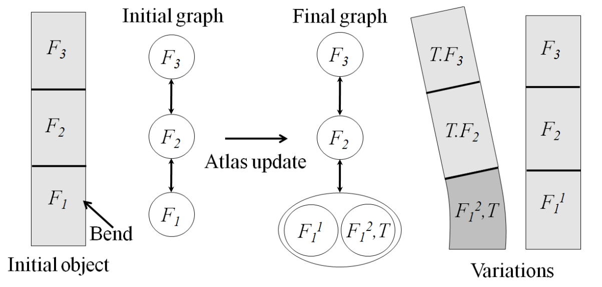

Figure 9: Atlas update for a solid transformation of the connecting region.

When applying an affine transformation like a translation, rotation or scaling to the whole set of vertices composing the connecting region R_{ij} of the component F_i, we store the transformations in a frame associated to R_{ij} which enables us to accurately connect components together during the assembly process. For example, in Figure 9, if component F_1^2 is selected, we must apply the frame transformation T to F_2 and F_3.

4.2.3 Modification of the connecting region

In the general case, when the modification deforms the shape of the connecting region R_{ij}, the graph $\mathcal{G}$ is duplicated to create a new one $\mathcal{G}'$ where other variants of $\mathcal{F}_i$ and $\mathcal{F}_j$ are removed. This extends the possibilities by dividing the atlas (Figure 10).

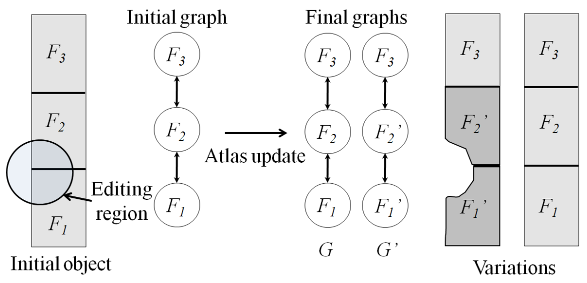

Figure 10: Atlas update for a modification affecting the connecting region.

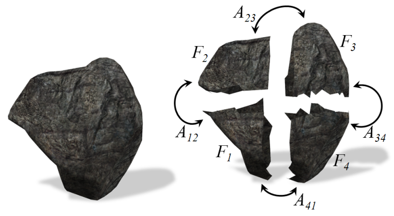

Figure 11: Fully cyclic graph obtained from the decomposition of a rock into four pieces.

In some cases, the graph associated to an object can be fully cyclic without terminal nodes (Figure 11). Applying a solid transformation to a connecting region of a fully cyclic object makes the assembly impossible and introduces cracks. Consequently, we forbid this type of modification in our implementation and thus guarantee the assembling property of all components.

4.2.4 Identical connecting regions

After the decomposition step, we check if identical connecting regions exist between components or at both ends of a component. Then, the user has the ability to edit the initial graph by adding local cycles between nodes that have matching connecting regions. Thus, some components can be repeated several times as with shape grammars [18] (Figures 12 and 13).

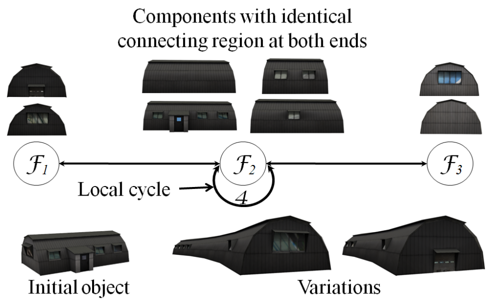

Figure 12: Components having the same connecting region at both ends and the associated graph.

To control how many times a component can be repeated, a maximum repetition number is associated to each local cycle of the graph by the user. This prevents infinite loop during assembly. This repetition possibility increases considerably the number of variations that can be generated.

In our implementation, we restrict the addition of local cycles on tree like graphs. Hence, affine transformations of the connecting region of components in a local cycle can be made without any risks of cracks (Figure 12). Note that terminal nodes ($\mathcal{F}_1$ and $\mathcal{F}_3$ in Figure 12 and $\mathcal{F}_1$ and $\mathcal{F}_4$ in Figure 13) cannot be in a local cycle. In our implementation, we forbid the addition of local cycles when terminal nodes are concerned.

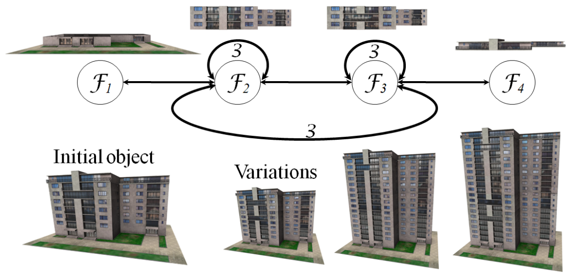

Figure 13: A complex graph with many user edited local cycles, and a set of buildings generated with this graph.

4.3 Assembly

This step generates a final new object $\tilde{O}$ by traversing the graph $\mathcal{G}$ and selecting a component F_i^k at each node N_i. Components are assembled such that connecting regions R_{ij} should match so as to guarantee seamless geometry and texture continuity between components.

During this assembly stage, it is necessary to recalculate the position of some components in the case of an affine transformation being applied to the connecting region. To achieve that and guarantee the continuity between components, we apply the frame transformation of the connecting region to all the components of the sub-tree.

In our implementation, the choice of the components F_i^k of $\tilde{O}$ can be made manually by the user, or randomly. The manual method gives more accuracy, but is rather limited when a large number of objects has to be generated like in the forest synthesis case (Figure 1). Since the modifications applied to components are mostly slight, the assembly gives slightly modified objects. Therefore, automatic selection of them gives very acceptable results.

As shown in section 3.1, the initial objects may have self-intersecting parts (Figure 2). Moreover, new self-intersections may be introduced in the generated objects during the assembly process. To solve this problem, we first detect the initial self-intersections in the initial object and label intersecting faces with the same identifier. Note that a face can have multiple labels since it can intersect with other faces. Whenever a new object is generated, we detect the intersections between the faces of the new model, and the object is rejected if two intersecting faces are labeled with different identifiers.

5 Component-based Generation of Texture Variety

Given an initial object O with its texture denoted as T, we propose a method that generates an atlas of texture components that may be combined and assembled to create different texture maps onto the object. The originality of our approach is that we do not need to modify the texture coordinates on the mesh. This is crucial in the video game industry where only a limited number of information such as texture coordinates may be stored for every vertex.

Our method proceeds in three steps. First, we add details to the initial texture T by directly painting on the object. We create a new small texture component T_k for each modification. The second step arranges texture components T_k in a set of texture components atlases. The variety of textures is obtained by selecting and combining texture components in the atlases (Figure 14).

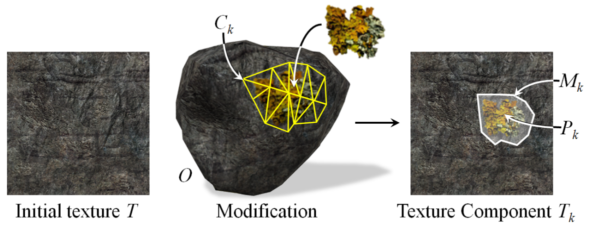

Figure 15: Texture component $T_k = (P_k,\ M_k,\ C_k\ )$ obtained after a local modification.

5.1 Texture modification process

This step consists in applying successive modifications on the texture of the object. In our system, we modify the texture by painting directly on its faces. Each modification is stored in a new texture component (Figure 15).

The painted region defines a texture component denoted as $T_k = (P_k,\ M_k,\ C_k\ )$. P_k refers to a pixel mask that defines the modified region in the texture domain. C_k denotes the cluster of faces that contains references to the modified faces of the object. M_k is a larger pixel mask such that $P_k \subset M_k$ which contains the modified faces C_k projected in the texture domain.

Usually, a local modification on the object results in a local modification in the texture domain. Different faces of the object may overlap in the texture domain (Figure 16) or even a single face of the object may contain cyclic repetitions of the texture (Figure 17).

To solve this parameterization overlapping problem, we create new texture components whenever local modifications on different parts of the object enter into conflict by overlaping in the texture domain.

Another important problem is texture tiling that may occur within a single face of the object (Figure 17) and results in an undesired repetition of the modification into the face itself whenever the texture coordinates range beyond unity interval $[0, 1]^2$. In this

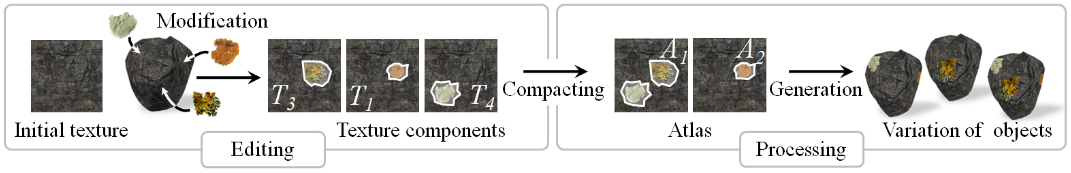

Figure 14: Synthetic overview of the texture generation process.

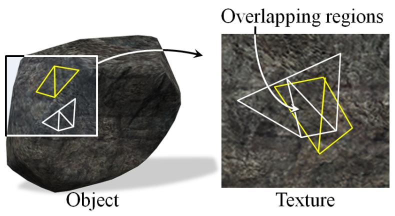

Figure 16: Clusters overlapping.

case, we subdivide the face where the texture repetition occurs into subfaces so that the range of texture coordinates gets into unity interval. Therefore, we remove the tiling and guarantee that any local texture modification will not produce any undesired repetition.

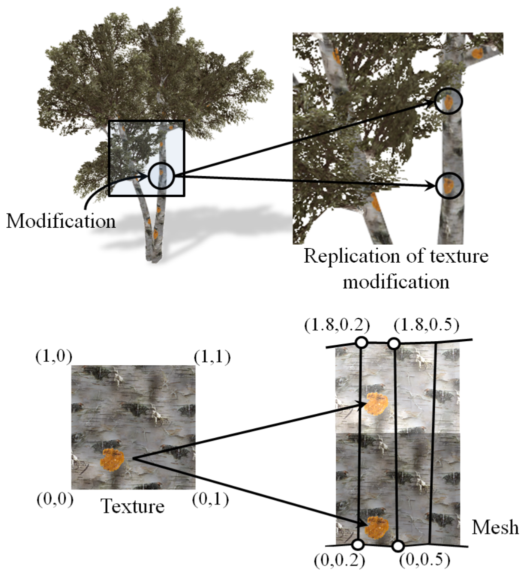

Figure 17: Repetition of the texture modification may occur because of the tiling.

When the user performs n edition steps, n texture components are created which is memory expensive. Thus, it is necessary to reduce the number of texture components by compacting them into texture components atlases.

5.2 Compacting process

This step merges the non-conflicting texture components T_k into texture components atlases denoted as A_i (Figure 18).

Let $T_i = (P_i, M_i, C_i)$ and $T_j = (P_j, M_j, C_j)$ denote two texture components. T_i and T_j are said to be non-conflicting and can be compacted in the same atlas provided $P_i \bigcap M_j = \emptyset$ and $P_j \bigcap M_i = \emptyset$. This compacting method is not efficient in terms of memory compared to methods minimizing the number of unused pixels [15, 25] in texture components atlases. In contrast, our technique preserves the existing texture coordinates of the object and avoids the creation of new ones for each texture component's atlas. This enables us to guarantee the seamless mapping of texture components.

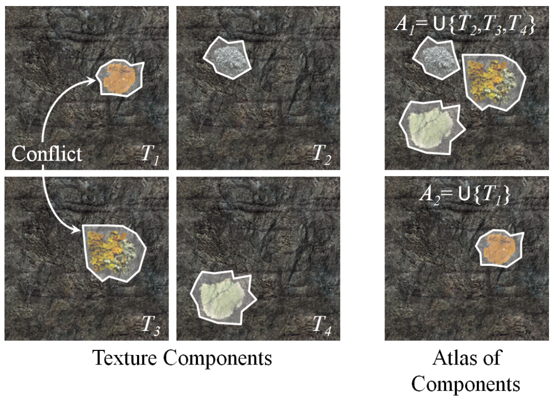

Figure 18: Texture components are compacted in texture components atlases.

In our implementation, we enumerate all compacting possibilities by checking intersections between each pixel mask and each face mask of each texture component. We finally keep the set of texture components atlases which has the smallest cardinal number.

After the compacting step, every face of the cluster C_k of the texture component T_k is labelled with the associated texture components atlas identifier A_i in which it has been compacted. At this stage, pixel and face masks are no longer necessary and can be removed from texture components.

5.3 Variety generation process

This step consists in choosing which modified texture components will be mapped onto the final object (Figure 19).

In our implementation, this step is automatically performed by associating a random probability of appearance for every texture component. The user may also directly control which texture fragments should be used for a specific model. For every selected texture component T_k, we assign the corresponding atlas identifier A_i containing the component T_k to the faces of the cluster C_k. Faces of the cluster are then locked so as to prevent the selection of texture components sharing common faces. This avoids texture component overlapping and partial replacement of a texture component by another.

When displaying the object, we use the identifier associated to each face to load the atlas containing the selected texture components and to bind the right texture modifications on these faces using initial texture coordinates. In this way, we avoid the packing

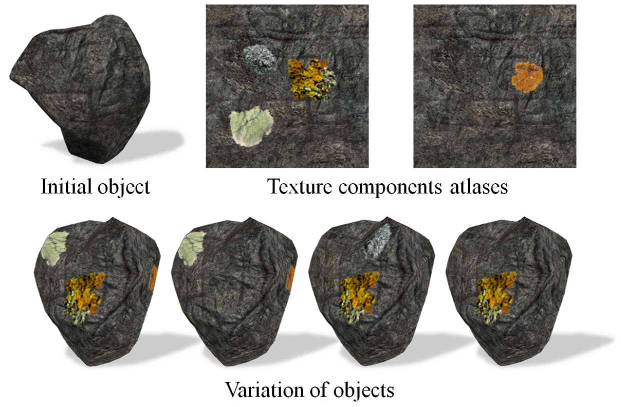

Figure 19: Four rock variations among $2^4 = 16$ possibilities.

of the texture atlases into a single texture that would imply texture coordinates modifications. Moreover, in the case of an initial tilling texture, the packing would break the ability to tile and would generate a large texture made of several copies of the initial texture.

6 Results

We have implemented and integrated our algorithms as an application in the *Twilight 2* development framework of Eden Games and Widescreen Games companies for the GENAC 2 project. Graphical assets used to illustrate our results are drawn from *Alone In The Dark*™ and *Test Drive Unlimited*™ video games from Eden Games company.

Table 1 presents the characteristics of the initial objects: vertices and faces number, memory (in kB) of the model. Table 2 reports the number of components generated after the cutting process, the number of variations per component, the total number of generated objects and the memory overhead.

Model	Vertices	Faces	Memory
Birch	745	628	328
Rock	578	1 152	146
House	568	923	186

Table 1: Models complexity and memory (in kB).

6.1 Interface

Our model has been implemented into the *Twilight 2* platform of EDEN Games. All the following operations are accessible into the interface as a contextual menu. The user can use standard mesh cutting and editing tools to perform the decomposition of the input object into components. Standard cutting tools often increase the overall complexity of the models by generating many new edges and new faces. Therefore, our interface allows the used to select faces either by picking one by one or with a rectangle selection tool. At the end of the fragmentation process, the user invokes a *detach from object* command and the component is created.

Deformations are then applied to the components as follows. The user can use a variety of deformation tools, such as Free Form Deformation [26, 8] and Axial Deformation [13], as well as low level mesh editing commands such as removing some faces of the model (for example it can consist of removing the foliage of a branch), or moving the vertices of the model (for example stretching the roof of a building by moving its top vertices).

Throughout the decomposition and component deformation steps, the interface prompts the atlas of fragments as a set of icons so that the user can keep track of all the fragments that have been created. During the assembly process, the user either selects the fragments to generate a specific variant, or simply invokes a *create random instance* command from the menu to generate a new model randomly.

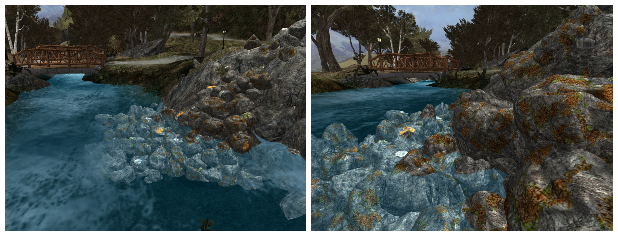

Figure 20: Automatic generation of rocks used in a rock slide.

6.2 Realism

Intensive instantiation of objects creates unnatural replications. Results demonstrate that our method is efficient to remove those replications and to increase realism by generating a great number of different objects for a low memory overhead and few modifications.

Model	Components	Variations	Objects	Memory
Birch	6	3	729	+54%
Rock	4	8	4 096	+321%
House	7	3	2 187	+48%

Table 2: Number of objects generated and memory overhead.

Figure 1 shows an area of central park with trees generated with our method from a single initial model. 769 different trees were generated by combining the different components. The modifications applied to components essentially consisted in removing branches and foliage, which explains why the memory overhead is low (54%). In comparaison, only 51 different trees (17 species with 3 different variation per species) were used for the game *Alone In The Dark*™.

Figure 20 shows a rock slide in Central Park. At the beginning, this rock slide was not present in the scene and our system allowed us to create it from a single rock. In addition to the geometric modifications that generate 256 rocks, we produced 4 texture modifications compacted in 2 texture components atlases which gave us the ability to generate 4 096 different rocks. During the geometry modifications, we did not remove any vertices as in the previous example. Therefore, the memory overhead is higher (321%) than in the birch example.

The images were generated from the assets of the video game *Test Drive Unlimited*™ which takes place on Hawaii island. The variations of houses in a suburban environment have been designed manually by artists who added construction elements like garden shelters or garages to an initial construction template. Figure 21 shows different houses obtained automatically with our system.

6.3 Efficiency and usability

An important feature of our approach is that it provides a means of generating a variety of objects from a single examplar. Our method

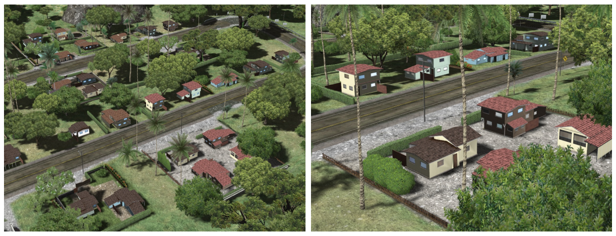
Figure 21: Suburban environment in Hawaii: each house is different and has been generated from an initial model.

is agnostic to the underlying technique used to generate the original model. Moreover, our method can generate a vast number of slightly varying objects with only a few editing operations. For example, three hours were necessary for a non specialist to generate the components and create 729 tree variations, including a very short training period. This is to be compared to the 8 hours spent by an artist in order to create a single tree. Our method permits reducing production times and costs which are crucial in industry.

7 Conclusion

In this paper, we have presented an example-based method for generating many variants of an initial low polygonal model. Our results demonstrate that this technique increases the overall realism by avoiding model replications produced by instantiation.

While our method requires some manual editing, it provides the artist with a good control over the overall process and can be applied to different kinds of objects. Because components can be instantiated, our system is well adapted to real-time rendering constraints and lends itself for the entertainment industry. Furthermore, it reduces production time and cost, which is important in an industrial development environment.

References

[1] M. Alexa, D. Cohen-Or, and D. Levin. As-rigid-as-possible shape interpolation. In *Proceedings of ACM SIGGRAPH*, pages 157–164, 2000.

[2] M. Attene, S. Katz, M. Mortara, G. Patane, M. Spagnuolo, and A. Tal. Mesh segmentation - a comparative study. In *Proceedings of the IEEE International Conference on Shape Modeling and Applications (SMI)*, pages 7–18, 2006.

[3] N. A. Carr and J. C. Hart. Painting detail. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 23(3):845–852, 2004.

[4] G. Chen, G. Esch, P. Wonka, P. Müller, and E. Zhang. Interactive procedural street modeling. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 27(3):103:1–10, 2008.

[5] X. Chen, S. B. Kang, Y.-Q. Xu, J. Dorsey, and H.-Y. Shum. Sketching reality: Realistic interpretation of architectural designs. *ACM Transactions on Graphics*, 27(2):11:1–15, 2008.

[6] X. Chen, B. Neubert, Y.-Q. Xu, O. Deussen, and S. B. Kang. Sketch-based tree modeling using markov random field. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)*, 27(5):109:1–10, 2008.

[7] R. Cook, J. Halstead, M. Planck, and D. Ryu. Stochastic simplification for aggregate detail. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 26(3):79–88, 2007.

[8] S. Coquillart. Extended free-form deformation: a sculpturing tool for 3d geometric modeling. *Proceedings of ACM SIGGRAPH*, 24(4):187–196, 1990.

[9] O. Deussen, P. Hanrahan, B. Lintermann, R. Měch, M. Pharr, and P. Prusinkiewicz. Realistic modeling and rendering of plant ecosystems. In *Proceedings of ACM SIGGRAPH*, pages 275–286, 1998.

[10] A. Dietrich, G. Marmitt, and P. Slusallek. Terrain guided multi-level instancing of highly complex plant populations. In *Proceedings of IEEE Symposium on Interactive Ray Tracing*, pages 169–176, 2006.

[11] A. A. Efros and W. T. Freeman. Image quilting for texture synthesis and transfer. In *Proceedings of ACM SIGGRAPH*, pages 341–346, 2001.

[12] T. Funkhouser, M. Kazhdan, P. Shilane, P. Min, W. Kiefer, A. Tal, S. Rusinkiewicz, and D. Dobkin. Modeling by example. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 23(3):652–663, 2004.

[13] F. Lazarus, S. Coquillart, and P. Jancène. Axial deformations: an intuitive deformation technique. *Computer-Aided Design*, 26(8):607 – 613, 1994.

[14] A. W. F. Lee, D. Dobkin, W. Sweldens, and P. Schröder. Multiresolution mesh morphing. In *Proceedings of ACM SIGGRAPH*, pages 343–350, 1999.

[15] B. Lévy, S. Petitjean, N. Ray, and J. Maillot. Least squares conformal maps for automatic texture atlas generation. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 21(3):362–371, 2002.

[16] M. Lipp, P. Wonka, and M. Wimmer. Interactive visual editing of grammars for procedural architecture. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 27(3):102:1–10, 2008.

[17] P. Merrell. Example-based model synthesis. In *Proceedings of the Symposium on Interactive 3D graphics and games*, pages 105–112, 2007.

[18] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. V. Gool. Procedural modeling of buildings. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 25(3):614–623, 2006.

[19] F. K. Musgrave, C. E. Kolb, and R. S. Mace. The synthesis and rendering of eroded fractal terrains. In *Proceedings of ACM SIGGRAPH*, pages 41–50, 1989.

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

[21] M. Okabe, S. Owada, and T. Igarashi. Interactive design of botanical trees using freehand sketches and example-based editing. *Computer Graphics Forum (Proc. Eurographics)*, 24(3):487–496, 2005.

[22] W. Palubicki, K. Horel, S. Longay, A. Runions, B. Lane, R. Měch, and P. Prusinkiewicz. Self-organizing tree models for image synthesis. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 28(3):58:1–10, 2009.

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

[24] P. Prusinkiewicz, M. S. Hammel, and E. Mjolsness. Animation of plant development. In *Proceedings of ACM SIGGRAPH*, pages 351–360, 1993.

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

[26] T. W. Sederberg and S. R. Parry. Free-form deformation of solid geometric models. *Proceedings of ACM SIGGRAPH*, 20(4):151–160, 1986.

[27] P.-P. J. Sloan, I. Charles F. Rose, and M. F. Cohen. Shape by example. In *Proceedings of the Symposium on Interactive 3D graphics*, pages 135–143, 2001.

[28] L. Streit, P. Federl, and M. C. Sousa. Modelling plant variation through growth. *Computer Graphics Forum (Proc. Eurographics)*, 24(3):497–506, 2005.

[29] P. Tan, G. Zeng, J. Wang, S. B. Kang, and L. Quan. Image-based tree modeling. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 26(3):87:1–10, 2007.

[30] J. Weber and J. Penn. Creation and rendering of realistic trees. In *Proceedings of ACM SIGGRAPH*, pages 119–128, 1995.

[31] L.-Y. Wei and M. Levoy. Fast texture synthesis using tree-structured vector quantization. In *Proceedings of ACM SIGGRAPH*, pages 479–488, 2000.

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

[33] K. Zhou, P. Du, L. Wang, J. Shi, B. Guo, and H.-Y. Shum. Decorating surfaces with bidirectional texture functions. *IEEE Transactions on Visualization and Computer Graphics*, 11(5):519–528, 2005.