Multiresolution foliage for forest rendering

By Qingqiong Deng, Xiaopeng Zhang*, Gang Yang and Marc Jaeger

Plants are important objects in virtual environments. High complexity of shape structure is found in plant communities. Level of detail (LOD) of plant geometric models becomes important for interactive forest rendering. We emphasize three major problems in current research: the time consumption in LOD model construction and extraction, the balance between visual effect and data compression, and the time consumption in the communication between Central Processing Unit (CPU) and Graphics Processing Unit (GPU). We present a new foliage simplification framework for LOD model and forest rendering. By an uneven subdivision of the tree crown volume, the cost for LOD model construction is drastically reduced. With a GPU-oriented design of LOD storage structure for foliage, the costly hierarchical traversal of a binary tree is replaced by a sequential lookup of an array. The structure also decreases the communication between the CPU and the GPU in rendering. In addition, Leaf density is introduced to adapt compression to the local distribution of leaves, so that more visually relevant details are kept. According to foliage nature (broad leaves or needles), higher compression are finally reached using mixed polygon/line models. This framework is implemented on virtual scenes of simulated trees with high detail. Copyright © 2009 John Wiley & Sons, Ltd.

Received: 24 June 2008; Revised: 3 March 2009; Accepted: 4 March 2009

KEY WORDS: tree foliage; level of detail; compression; multi-resolution; rendering; GPU-oriented

Introduction

Realistic and real-time rendering of plant scenery is a key task in computer graphics and virtual reality, showing high interest in many applications such as urban visualization, flight simulation, and digital agriculture. However, plants, especially trees, contain rich information for 3D geometric details and topological structures, making them unsuitable for real-time rendering in a brute force way. Different approaches thus have been developed to overcome this difficulty on the basis of data compression and level of detail (LOD) techniques, but such approaches cannot satisfy rendering speed and rendering quality simultaneously.

The purpose of this paper is to propose a new and more efficient foliage simplification algorithm for fast forest rendering. The algorithm we aim to design should have at least three properties. First, it must be general, fast, and practicable for all possible species of leaves. Second, it should construct continuous level of detail models for foliage, so that the switch between different detail levels is smooth and not visually perceptible. Finally, the proposed approach must be able to render outdoor plant communities with rich 3D geometric details, at a high frame rate while visual qualities are well maintained.

Foliage classification classically distinguishes broad-leaf from conifer, and the two kinds of leaves show different visual properties. To fulfill the requirement of generality, we have developed an algorithm for each kind and integrated them in a unified framework. The algorithm proposed for the broad leaves is called the Broad Foliage Simplification Algorithm (BFSA), and the one for thin leaves is called the Thin Foliage
Simplification Algorithm (TFSA). Graphical primitives used to construct the progressive LOD models of the two kinds are simply related to the classification: a quadrilateral for a broad leaf and a line for a thin leaf. The two algorithm share a similar data structure, which is GPU-oriented, for efficient extraction of a LOD model. In addition, leaf density is introduced to construct a non-uniform model for further acceleration. As a result, plants can be rendered much faster than with other foliage simplification methods.\textsuperscript{4–11}

The rest of the paper is organized as follows. After reviewing Related Work in the next section, the construction of continuous foliage LOD models is described in the Simplification of Leaves with Generic Shapes section. In this section, we show how broad and thin leaves with general shapes are simplified in the preprocessing stage with the BFSA and the TFSA, respectively. Then, a GPU-oriented and error-controlled extraction of LOD models for rendering is described in the Error-controlled LOD Models for Rendering section. The extraction is performed on the fly at the rendering stage. We also introduce the leaf density concept in this section, and show how it helps to keep more visual details while the overall compression becomes higher. In the Forest Rendering section, a construction and rendering system of plant communities is presented, by combining our foliage simplification method with a branch simplification method, a plant distribution editor and a plant modeling method. Finally, the experimental results for these techniques are presented in the Experiments section, with conclusion and future work in the last section.

Please note that we only mention the simplification of leaves in the paper, but our approach can be easily extend to other sparse organs such as flowers and fruits.

**Related Work**

An important goal of computer graphics is to reproduce natural scenes as realistically as possible. Until now, plant modeling has been extensively explored, and there have been many advanced approaches that can generate realistic plants efficiently such as L-systems,\textsuperscript{12} AMAP,\textsuperscript{13} and Xfrog.\textsuperscript{14} However, the generated plant models usually have extremely high complexity that surpasses even the capabilities of current high end computers.

Various methods have been proposed to accelerate plant rendering. These methods can be classified into three categories: image-based methods, point-based methods, and polygon-based methods.

**Image-based Methods**

There are many methods in this field. The most commonly used are billboards and impostors, but they can only be used to represent distant plants due to their bad parallax effect. Jakulin\textsuperscript{15} used sets of parallel billboards to represent trees to achieve better effects than a single image. Lluch \textit{et al.}\textsuperscript{16} introduced hierarchical structure into the common billboard representation, and the structure is traversed to get a suitable level of detail in rendering. Meyer \textit{et al.}\textsuperscript{17} proposed a system utilizing a hierarchy of bidirectional textures (HBT) to support self-shadowing and cast shadowing effects for trees. Recently, some authors\textsuperscript{1–18} have developed a more flexible form of billboard, the billboard clouds, to represent a plant by a set of arbitrarily oriented billboards.

Max \textit{et al.} used precomputed z-buffer views to reconstruct images from arbitrary viewpoints.\textsuperscript{19,20} Shade \textit{et al.}\textsuperscript{21} proposed Layered Depth Image (LDI), which was similar to the multiple z-buffer layers, for representing plants. Later, Chang \textit{et al.}\textsuperscript{22} transformed the LDI into a hierarchical form and combined it with the LOD technique.

Besides 2D texture, the volumetric texture was applied to rendering landscape covered by continuous vegetation in References [2,23]. However, this method requires lots of memory, even for a small forest.

**Point-based Methods**

Point models were first used in plant rendering by Reeves \textit{et al.},\textsuperscript{24} in which leaves were represented by a set of small disks. The point/line usually combines polygon to construct hybrid models for trees. At close distances, polygonal geometry is used. With increasing viewing distance, branch meshes will transform into lines and leaves into points. For even farther distance, fewer points and lines will be used by randomly sampling,\textsuperscript{3,25} or by merging small points into one,\textsuperscript{26} which results in a hierarchical point representation. However, traversing the hierarchical structure consumes much CPU resources. Dachsbander \textit{et al.}\textsuperscript{27} propose a method to sequentialize the hierarchical structure to an array for an adaptive rendering of point cloud models. Such an approach has been improved in our paper to minimize the transfer of data from the CPU to the GPU.
Polygon-based Methods

Many geometric simplification methods were explored to avoid unnecessary geometry details, but the methods do not support plants well, since their foliage consists of many isolated surfaces.

Some special approaches have been proposed for the simplification of foliage. The Foliage Simplification Algorithm (FSA) simplifies foliage by repeatedly selecting a pair of leaves to implement leaf collapse: two leaves are replaced by a new larger leaf with a similar position to the original pair. To determine which pair of leaves should be collapsed, FSA defines a cost function. The cost function is redefined in Progressive Leaves Union (PLU) for better preservation of silhouette and the overall foliage area of a crown. Both FSA and PLU can only deal with quadrilateral-shaped leaves. Hierarchical Union of Organs (HUO) improves PLU to include triangular leaves (see Figure 1), and it introduces hierarchy into simplification, by making use of the fact that the leaves in a crown are constructed by instancing a phyllotaxy cluster sample (see Figure 2) with different transformations, so that the simplification within every instance of the sample is the same. Accordingly, there are three levels in the HUO: each single leaf of the phyllotaxy cluster, the phyllotaxy cluster, and the whole foliage. Later, the idea of hierarchy is extended in Reference to simplify leaves according to not only phyllotaxy, but also the topology of branches. The methods mentioned above can simplify broad leaves well, but not coniferous leaves (see Figure 13). Deng et al. proposed a method for this tree species by using two representations, cylinder and line, to represent close and far coniferous leaves respectively, and lines can be merged for further simplification.

All these foliage simplification methods share a similar principle: to compress geometry by repeatedly fusing two leaves into one. So we called them leaf-union based methods. They usually use a binary tree structure to store simplification process data, where the new leaf generated by a leaf union operation is set as the father of the two fused leaves. But such a structure cannot easily be supported by a GPU. So some researchers transformed the binary tree to special data structures to suit the GPU pipeline for acceleration.

Rebollo et al. used two arrays to store simplification data, and loaded the geometric data of all LODs into the memory of the GPU at the beginning of rendering to reduce the data communication between the CPU and the GPU. The method can extract LODs faster than that using binary tree structure, but the transmission of vertex indices from the CPU to the GPU is still a heavy load. Hidalgo et al. saved the geometric data and the simplification data in one array, and loaded the array into the GPU memory. Hence no communication between the CPU and the GPU is needed. But the array is constructed neither by an error criterion, nor by the order of leaf collapse operation. As a result, the extracted LOD may be wrong.

Simplification of Leaves with Generic Shapes

Following the classical classification, we propose two methods: Broad Foliage Simplification Algorithm (BFSA) and Thin Foliage Simplification Algorithm (TFSA) for broad and thin leaves respectively, and integrate them in a unified framework (shown in Figure 3).

We use a simple criterion to determine if the leaves of a tree are broad or thin. It depends on the ratio of the length to the width of the bounding box of a leaf sampled from the tree. If the ratio is larger than a user defined threshold R the leaf is considered as a thin leaf, otherwise, it is broad. In this paper R is set to 10.

The hierarchical simplification idea proposed in the HUO algorithm is inherited in our framework. So both
Q. Deng et al.

Simplify each leaf of the phyllotaxy cluster to a quadrilateral

Leaves

No

Yes

Is leaf broad?

Is leaf complex?

Yes

No

Yes

Simplify the phyllotaxy cluster by leaf collapse

Build octree and simplify representative quadrilaterals by leaf collapse

Calculate leaf densities and modify simplification errors

Store simplification process data in an array

Simplify the phyllotaxy cluster by line merging

Build octree and simplify representative lines by line merging

Files

Preprocessing

Level 1

Level 2

Level 3

Rendering

Figure 3. Our framework for foliage simplification and rendering.

broad and thin leaves are progressively simplified in three levels: single leaf level, phyllotaxy cluster level, and whole crown level.

**Broad Foliage Simplification Algorithm**

The broad leaves in nature may have various shapes rather than just a quadrilateral or triangle. To simplify these leaves, we use a two-step method: simplifying each complex leaf mesh model to a quadrilateral first, and then simplifying the quadrilaterals with the traditional leaf union operation.

The mesh simplification method proposed in Reference [30] is used in the first step of our algorithm. We classify the vertices of a leaf mesh model into three categories: inner, boundary and corner points. The corner points are not decimated during the simplification. To preserve boundary well, we choose two vertices as corner points. One is the vertex that is farthest from the branch bearing the leaf (see Figure 2). Generally, this vertex is the tip of the leaf. The other is the leafstalk.
MULTIRESOLUTION FOLIAGE FOR FOREST RENDERING

Figure 4. A comparison between the HUO and the BFSA on a willow leaf. (a) Simplified results of the HUO. (b) Simplified results of the BFSA.

vertex, which usually has the longest distance to the tip vertex.

The order of vertex decimation is determined according to the decimation errors of vertices. In the simplification algorithm, the errors of corner vertices are evaluated with an extreme high value, so they are always on the rear part of the decimation vertex queue.

The progressive simplification of a complex leaf with the HUO is shown in Figure 4(a). The corresponding results of our algorithm is in Figure 4(b), where the red vertices are the corner points that remain unchanged during simplification. The comparison between the two figures demonstrates that our method can get more smooth simplification results.

After simplifying each complex leaf of the phyllotaxy cluster to a quadrilateral, the progressive simplification of the phyllotaxy cluster and the whole crown is performed subsequently, as in the HUO algorithm. It is an iterative process. In each step of simplification, we choose the pair of leaves minimizing the cost function defined in Reference [7] to implement leaf union. The cost function is a linear equation of six items, as shown in formula (1).

\[ S(X, Y) = \sum_{i=1}^{6} k_i S_i(X, Y) \]  

where \( k_i (k_i > 0) \) are weights, which are specified as 0.0375, 0.3125, 0.0375, 0.0375, 0.2875 and 0.2875, respectively in our experiments. \( S_1(X, Y), S_2(X, Y), S_3(X, Y), S_4(X, Y), S_5(X, Y) \) and \( S_6(X, Y) \) are the normal similarity, positional similarity, area similarity, diameter similarity, union age similarity, and diameter penalty of the two leaves \( X \) and \( Y \), respectively. The diameter of a leaf is the maximum distance among all possible pair of vertices of the leaf, and the union age of a leaf means the number of original leaves decimated to generate the leaf. For more details, see Reference [7]. At each step of simplification, the simplification error of every leaf is recorded. The error of an original leaf is set to 0, and that of a new generated leaf is defined as the Hausdorff distance from this leaf to all the decimated original leaves used to generate it.

Thin Foliage Simplification Algorithm

The simplification method for coniferous leaves proposed in Reference [10] is improved in this paper to deal with thin leaves. Instead of cylinders and lines used in Reference [10], polygons and lines are used in our algorithm to represent close and far leaves, respectively.

There are three steps. Since thin leaves may have complex shapes, we first simplify leaf models into quadrilaterals, which is the same as the BFSA. Then, we convert each quadrilateral into a line. Finally, we repeatedly perform line merging that unites two lines into one. The error of each simplification is stored, just as BFSA does.

The conversion of a quadrilateral into a line is simple. From the four vertices of the quadrilateral, the two vertices with maximal distance are chosen as the two ends of the line. The normal vectors of the two ends are calculated as the average normal of the four vertices. The conversion error is defined as the maximal distance of the four vertices from the new line.

Figure 5(a)–(c) show an example of the conversion from line model to polygon model as camera becomes closer and closer. To distinguish polygons from lines, in the figures, polygons are colored with green, and lines with blue.

Simplification Acceleration

Both the leaf union in the BFSA and the line merging in the TFSA are implemented in an iterative way. In each step of the simplification, cost values for leaf pairs are computed, and the pair with minimum value is then selected to be decimated. Such selection leads to a low preprocessing efficiency of existing methods.
It is not difficult to deduce that neither leaf union nor line merging happens for two leaves that are quite far from each other. Based on such a proposition, we assume that leaf union or line merging is only performed on the pair of leaves whose distance is not larger than a user defined distance threshold $\Omega$. The distance of two leaves is defined as the distance between their center points.

In order to avoid quadratic number of comparisons when looking for leaf pairs that satisfy the distance condition from the whole crown leaves, we cluster leaves into a modified octree structure.\(^3\)\(^1\) With this structure, when looking for distance-satisfied pairs in one box, its neighboring boxes don’t need to be considered.

The modified octree structure is built according to the center points of leaves. We first compute the bounding box for the points, and then recursively subdivide it into eight parts. Instead of splitting a box in three coordinates as the classical octree does, we split the octree on the longest side each time. In order not to miss the pairs with two leaves belonging to different boxes, the boxes are enlarged by $\frac{1}{2} \Omega$ on the corresponding side after each split, as showed in Figure 6. The subdivision will stop when either the number of points in a box is not larger than a user defined number threshold $L$, or one side of a box is not larger than $\Omega$. 

---

Figure 5. Conversion between line model and polygon model.

Figure 6. Creation of two boxes with one extending side in the octree structure.
Using this structure, we can obtain the leaf pairs that satisfy the distance condition quickly. Such pairs are called valid pairs. The valid pairs only need to be calculated one time, and they are candidates for the following leaf union (or line merging) operations. For each simplification, the selection of the best pair from all valid pairs is implemented in two passes. Firstly, we select the pair minimizing the cost function within the valid pairs of each leaf-node of the octree. Secondly, the comparisons are performed among these selected pairs to get the pair with the minimal cost value for all leaf-nodes. After each leaf union (or line merging), the pairs involving one or two of the merged leaves must be deleted, and new pairs involving the new generated leaf must be entered. Then, the selection will be implemented again to find the least cost pair for the next leaf union operation.

Thanks to the distance condition, we can simplify big trees with a huge number of leaves in just a few seconds. For example, less than 8 seconds are used to simplify a 40 years old black poplar with 681 200 foliage polygons (shown in Figure 12). However, if we use other foliage simplification methods, for example, the HUO, it will take more than 10 hours.

We only accelerate the pair selection in the whole crown level, not in the phyllotaxy cluster level since the number of leaves in this level is usually rather small.

**Simplification Process Recording**

After the simplification, the geometric data and the simplification process data are all stored on hard disk.

**Geometric Data**

The vertices of all leaf LOD models of a tree are stored in an array called the vertex array, denoted by \( V \). Each record in \( V \) stores two attributes: position coordinates and normal vector. For broad leaves, another array, polygon array, denoted by \( P \), is built to store the vertex indices that form each leaf polygon of the original foliage model, as well as the vertex indices of the leaves generated during the simplification process. For thin leaves, there are two different representations: polygon and line, so we use two arrays. One is the polygon array that stores the vertex indices that form each polygon. The other is the line array, denoted by \( L \). It stores the vertex indices of each line converted from a quadrilateral, and those generated in the line merging process. For simplicity, we will use \( P \) to represent the union of \( P \) and \( L \) for the thin leaf models.

In our work, we do not use leaf textures for leaf rendering, and all leaves of a tree share the same material, which is also recorded.

**Simplification Process Data**

A special tree structure is used to store simplification relationship data. The structure is composed of two parts in the BFSA. One part is a binary tree that stores the simplification information in the leaf union processes. In this part, the new leaf generated in a leaf union process is stored as a father node of the two collapsed leaves. The other part is an array that stores the simplification information from a complex leaf model to a quadrilateral. Similar to the BFSA, the TFSA also use a binary tree and an array to store simplification relationship data. Each node of the binary tree and each record of the array corresponds to a leaf, storing the index of the polygon (or indices of the polygons) or line that forms the leaf, as well as the simplification error of the leaf.

As mentioned before, the binary tree is costly for extracting LOD and is not GPU-supported. So we adapt our data structure to exploit the power of current graphic hardware.

The approach in Reference [27] sequentializes the hierarchical point representation to an array for adaptive rendering of point cloud models. Each node of the binary tree is associated with two values: \( r_{\text{min}} \) and \( r_{\text{max}} \), where \( r_{\text{min}} \) represents the minimum viewing distance at which the node can be selected for rendering, and \( r_{\text{max}} \) is the \( r_{\text{min}} \) of the node’s direct parent or is infinity when the node is the root node. A node is chosen for rendering if it satisfies \( r \in [r_{\text{min}}, r_{\text{max}}] \), instead of the recursive test of the binary tree \( r \geq r_{\text{min}} \), where \( r \) represents current distance from the viewer to the rendered object. The nodes of the binary tree are arranged in an array according to the order of \( r_{\text{max}} \). Once given an \( r \), a binary search with condition \( r_{\text{max}} < r \) is performed first on the array to eliminate the nodes impossible to be drawn. So only a prefix of the array is needed to perform the non-recursive test, as shown in Figure 7. Finally, the nodes that pass the test will constitute the LOD model corresponding to the current view. In general, these nodes are not sequentially distributed in the array.

We improve the method of Reference [27] by constructing an array that satisfies both \( r_{\text{max}} \) - sorting and \( r_{\text{min}} \) - sorting, so that the nodes of a LOD are sequentially stored in the array, which is more appropriate to process with GPU.
The selection of nodes to form the correct LOD for rendering.

Figure 7. Conversion of a binary tree to an array. Top: A binary tree consisting of nodes a–i with valid range \([r_{\text{min}}, r_{\text{max}}]\). Bottom: (Left) The array converted from the binary tree. (Right) The selection of nodes to form the correct LOD for the current view.

If we separate the valid range \([r_{\text{min}}, r_{\text{max}}]\) of a node into several parts, like \([r_{\text{min}}, r_1]\), \([r_1, r_2]\) \ldots \([r_i, r_{\text{max}}]\), and test each part individually, the test result will not change. Such property can be used to construct the required array by splitting the nodes when they do not satisfy the two criteria simultaneously, at the expense of adding extra nodes in the array.

In our method, the recursive test corresponding to the binary tree is to check whether the simplification error of a node, denoted by \(e\), is equal to or smaller than a spatial error threshold \(r\). So the symbol \(r\) can be corresponded to \(r\), \(r_{\text{min}}\) and \(r_{\text{max}}\) corresponded to \(e_{\text{min}}\) and \(e_{\text{max}}\). In our method, we convert the binary tree to an array called \(F_{\text{array}}\). In \(F_{\text{array}}\), three values, \(e_{\text{min}}\), \(e_{\text{max}}\) and SN are associated to each record. SN stores the index for the node in the polygon array \(\mathbb{P}\). From SN we can get the geometry data of the node. To construct the \(F_{\text{array}}\), we use the \(e_{\text{max}}\)-sorting as a primary criterion, and divide the valid ranges of nodes to propagate some new ones when they don’t satisfy the \(e_{\text{min}}\)-sorting.

The pseudo-code to build the \(F_{\text{array}}\) is shown in Algorithm 1. It uses three auxiliary functions. Function \(\text{eminSatisfied}(\text{child1}, \text{child2}, \text{lastNode})\) tests whether the records in the \(F_{\text{array}}\) still satisfy \(e_{\text{min}}\)-sorting after adding \(\text{child1}\) and \(\text{child2}\). This is easily done by comparing the \(e_{\text{min}}\) of \(\text{lastNode}\) with the simplification errors of \(\text{child1}\) and \(\text{child2}\). If the former is larger or equal to the latter, the function returns true; otherwise, it returns false. Function \(\text{AddNodesInArray}(\text{child1}, \text{child2}, \ldots)\) adds “\(\text{child1}, \text{child2}, \ldots\)” in the \(F_{\text{array}}\) in a descending order of \(e_{\text{min}}\). Function \(\text{Propagate}(\text{node}, \text{currentNode}, e_{\text{min}}, \text{newPropagatedNodes})\) is used to divide the valid range of node, \([e_{\text{min}}, e_{\text{max}}]\) into two parts: \([e_{\text{min}}, \text{currentNode}, e_{\text{min}}]\) and \([\text{currentNode}, e_{\text{min}}, e_{\text{max}}]\), hence generating two new nodes whose SN values are all equal to that of the node. The new propagated nodes are stored in \(\text{newPropagatedNodes}\).

Figure 8 shows the process that builds the \(F_{\text{array}}\) from the binary tree of Figure 7. In the first step, the \(\text{currentNode}\) and \(\text{lastNode}\) are the same, i.e. the node \(a\). It has two children: \(b\) and \(c\). Since both the simplification errors of \(b\) and \(c\) are smaller than the \(e_{\text{min}}\) of the \(\text{lastNode}\), the function \(\text{eminSatisfied}\) returns true. So the nodes \(b\) and \(c\) are directly added at the end of the array in a descending order of simplification error. Now the \(\text{currentNode}\) becomes the node \(b\), and the \(\text{lastNode}\) is node \(c\). The node \(b\) has two children, node \(d\) and \(e\). The simplification error of the node \(d\) is larger than \(e_{\text{min}}\) of the \(\text{lastNode}\), making the function \(\text{eminSatisfied}\) return false. So node \(c\) will be partitioned. As shown in the third step of Figure 8, the node \(c\) with valid range \([3, 10]\) is divided into two new nodes with valid ranges \([3, 7]\) and \([7, 10]\), respectively. The two new nodes share the same SN: \(c\), and they replace the original node \(c\). After division, the two new nodes and the two children of \(\text{currentNode}\) are added in the array according to the value of \(e_{\text{min}}\). Noting that in the next step, the \(\text{currentNode}\) is not its next, but the node \(d\). In the following steps of Figure 8, the adding of the children of each \(\text{currentNode}\) will preserve the \(e_{\text{min}}\)-sorting criteria, so the function \(\text{Propagate}\) does not need to be implemented.

After converting the binary tree of the tree structure storing the simplification data to the \(F_{\text{array}}\), the \(F_{\text{array}}\) will be combined with the array of the tree structure to generate a unified array, called simplification relationship array and denoted by \(S\), which can be used for efficient extraction of LOD models and fast forest rendering.
Algorithm 1 Conversion of a binary tree to an array

Input: Binary tree data $B$  
Output: $F_{array}$

Init: Add the root node of the $B$ as the single record in the $F_{array}$; set $currentNode$ as the first record of the $F_{array}$

while $currentNode \neq NULL$ do

if $currentNode$ is a leafnode of $B$ then

$currentNode \leftarrow currentNode$'s next

else

child1 $\leftarrow currentNode$.Child1; child2 $\leftarrow currentNode$.Child2

lastNode $\leftarrow$ the last record of the $F_{array}$

recordNum $\leftarrow$ the number of records of the $F_{array}$

if $eminSatisfied$(child1, child2, lastNode) = true then

$AddNodesInArray$(child1, child2)

$currentNode \leftarrow currentNode$'s next

else

for $node = currentNode$'s next to lastNode do

$Propagate$(node, currentNode.$e_{min}$, newPropagatedNodes)

delete $node$ from the $F_{array}$

end for

$AddNodesInArray$(child1, child2, newPropagatedNodes)

$currentNode \leftarrow$ the record in the position of recordNum + 1 of the $F_{array}$

end if

end if

end while
Error-controlled LOD Models for Rendering

We discuss now how to render a crown with an appropriate LOD model according to viewing conditions.

A permitted pixel error threshold $\delta$, specified by the user, is used as a trade-off between rendering speed and visual quality. A larger $\delta$ will result in a higher frame rate but lower image quality. When rendering a plant, $\delta$ is first converted to a spatial error threshold $\epsilon$ by using the camera parameters and the distance between the plant and the camera. The distance is defined as the the distance from the camera to the center of the plant minus the radius of the enclosing sphere of the plant. So the rendering quality of the leaves that are closest to the camera can be guaranteed, although other leaves with farther distance from the camera maybe keep unnecessarily details. If the distance is less than zero, i.e. the camera is in the crown of the plant, the original model is extracted immediately.

After the spatial error threshold $\epsilon$ is specified for a plant, a suitable LOD model will be obtained from the simplification relationship array $S$ according to $\epsilon$. The vertex array $V$ is a static structure that remains unchanged during rendering, and it is initially loaded into the GPU memory to reduce the communication between the CPU and the GPU. At each frame, once we get the current LOD, we can calculate corresponding vertex indices using value $SN$ and the polygon array $P$. The vertex indices will be arranged in one array in the BFSA, and two arrays for polygon and line representations respectively in the TFSA. The polygon concerned in the BFSA and TFSA includes quadrilateral and triangle, and we uniformly associate four vertex indices for both of them, where the fourth and the third index is the same in triangle case. Finally, the indices array (or arrays) can be sent to a GPU for rendering.

Efficient Extraction of LOD Models

The LOD model for the current view can be achieved through two binary searches performed on the array $S$. The first search is to find the first entry in $S$ satisfying the condition $e_{\max} \leq \epsilon$. This entry is noted as $N$. The second one is to find the last entry satisfying condition $e_{\min} > \epsilon$, noted as $M$. The nodes between $M$ and $N$ satisfy the condition $\epsilon \in [e_{\min}, e_{\max})$, so that they compose an appropriate level of detail model for current $\epsilon$.

This extraction algorithm is very efficient, and it can be further accelerated by taking advantage of temporal coherence when the viewing position moves gradually. We associate each tree with three values: $M_l, N_l, \epsilon_l$, which store the values of $M, N$ and $\epsilon$ of the last frame. In the drawing process, we first compare $\epsilon$ with $\epsilon_l$. There are three conditions. If $\epsilon = \epsilon_l$, then $M = M_l$ and $N = N_l$. If $\epsilon > \epsilon_l$, we implement the first binary search with the condition $e_{\min} > \epsilon$ in the range from the first node of the array to the node $M_l$ to get $M$, and implement the second search with the condition $e_{\max} \leq \epsilon$ from the node $M$ to the node $N_l$ to get $N$. If $\epsilon < \epsilon_l$, we perform the first search with the condition $e_{\max} \leq \epsilon$ from the node $N_l$ to the last node of the array to get $N$, and perform the second one with the condition $e_{\min} > \epsilon$ from the node $M_l$ to the node $N$ to get $M$.

The transmission of vertex indices from the CPU to the GPU can be avoided also. Since each LOD forms a segment of $S$, we can build another array (for thin leaves we will build two arrays for line and polygon, respectively) in preprocessing, which stores vertex indices for each record of $S$ in the same order. The new array (arrays) is called index array, denoted...
by \( I \). Before rendering, both \( V \) and \( I \) are loaded into the GPU, and \( S \) into the CPU. For each new view, we only need to update \( M \) and \( N \) in \( S \) through two binary searches mentioned above, and convert them to their corresponding positions in \( I \), which are noted as \( M \) and \( N \). Then, the rendering can be easily implemented. In this condition, changing detail levels almost requires no communication between the CPU and the GPU. As a result, we can achieve interactive frame rates even when rendering tens of thousands of trees.

### Leaf Density for Non-uniform LOD

In the error-controlled rendering algorithm described above, a uniform LOD according to the distance from the tree to the viewer is used. However, human eyes are sensitive to the detail of the region with sparse leaves, but less sensitive to detail when leaves are dense. This characteristic of human eyes suggests that simplification degrees can be different in the different parts of the tree crown. The parts with dense leaves can be simplified more, resulting in higher data compression without sacrificing the final visual quality.

To describe the leaf distribution in a crown, a new concept, leaf density \( \Delta \), is introduced. Leaf density \( \Delta = \Delta(\lambda, \rho) \) is defined for the leaf \( \lambda \) as the number of leaves whose barycenter points fall in a sphere of radius \( \rho \) centered on the barycenter of the leaf \( \lambda \). The radius \( \rho \) is a constant in our work, so \( \Delta = \Delta(\lambda, \rho) \) can be abbreviated as \( \Delta = \Delta(\lambda) \).

In practice, the radius \( \rho \) is set to be equal to the distance threshold \( \Omega \), so that the record of valid pairs in the Simplification Acceleration subsection can be reused for the calculation of leaf densities. The modified octree structure is used in the third level of the tree, so the valid pairs are pairs of phyllotaxy clusters rather than those of leaves. Fortunately, the densities of phyllotaxy clusters also reflect the leaf distribution in a crown. Since leaves are normally distributed evenly in a phyllotaxy, it is reasonable to assume that the leaves in one phyllotaxy cluster have the same density value, which is equal to the density of the phyllotaxy cluster. The density of a new leaf generated by a leaf union (line merging) is calculated as the area-weighted average density of the two collapsed leaves.

After getting leaf densities for all leaves in the crown, they will be normalized to be in an interval \([1, \eta]\) by using formula (2). The parameter \( \eta \) is specified by the user.

\[
\Delta'(\lambda) = \frac{\Delta(\lambda) - \Delta_{\min}}{\Delta_{\max} - \Delta_{\min}} (\eta - 1) + 1
\]  

where \( \Delta_{\min} = \min(\Delta(\lambda) ; \lambda \in \Lambda) \), \( \Delta_{\max} = \max(\Delta(\lambda) ; \lambda \in \Lambda) \), and \( \Lambda \) is the set of all leaves.

It is noted that the essential test in our method is \( \varepsilon \geq e \). So modifying either \( \varepsilon \) or \( e \), can change test result. However, modifying \( \varepsilon \) needs to be done in the rendering process, thus increasing the rendering time cost; and leaf densities are view-independent. Therefore, we prefer to construct non-uniform LOD models for foliage by adapting the simplification errors of leaves in preprocessing, before converting the binary tree to an array to construct the array \( S \).

The simplification error \( e(\lambda) \) of a leaf \( \lambda \) can be adapted to \( e'(\lambda) \) as:

\[
e'(\lambda) = \frac{e(\lambda)}{\Delta'(\lambda)}
\]  

As a result, the larger the \( \Delta'(\lambda) \) is, the smaller the simplification error of the leaf \( \lambda \) will be, which means more significant simplification can be allowed for leaf \( \lambda \).

Figure 9 compares the effects of simplification with and without the consideration of leaf density on a maple in autumn. Figure 9(a) is the original model whose leaves distribute non-evenly. Figure 9(b) and (d) are simplified results without taking into account leaf density. Figure 9(c) and (e) are two corresponding results with density effect. The spatial error thresholds of Figure 9(b) and (c) are the same, but (c) has larger compression ratio while maintains similar visual quality as (b). The foliage polygon number of Figure 9(d) and (e) are similar, but (e) preserves details better.

### Forest Rendering

In addition to the foliage, the trunk and branches are important parts of a plant. The way to process trunk and branches for forest rendering are briefly described here. Plant trunk and branches are processed simultaneously, since they have similar shapes. Plant branch skeletons, with corresponding radius at each node, are used to construct branch surface models. Theoretically, plant branches are continuous surfaces connected according to plant topology, and each branch is represented by a generalized circular cylinder. Each generalized circular cylinder is converted to a model of connected prisms according to the error \( \varepsilon \) for this conversion in rendering. Normal vectors of the connected prisms are calculated as that of the end circle. The vertex coordinates and normal vectors of the connected prisms are recorded for...
rendering, where the normal vector of the surface at each vertex is saved for smooth rendering (Gouraud shading). The spatial error threshold $\epsilon$ is controlled by camera settings and the plant’s distance to the camera, the same as in the Error-controlled LOD Models for Rendering section. This error is view-dependent, so branch models are constructed according to the resolution needed. Skeleton data is a polyline, i.e. a list of connected nodes. This polyline can be simplified before the construction of branch mesh models.

Since the radius of each node in a branch is different, different sections of the branch prisms have different numbers of sides. The numbers of sides comes from the error between a circle and an equilateral polygon. This way the precision will be higher and complexity will be lower than the ordinary cylinder-based representation of branch models used for plant modeling. Figure 10 shows the process from skeleton to a multi-resolution branch model of a 6 years old, 2.2 meters high, holly tree that is 36 meters away from the viewer. Figure 10(a) is original skeletons of the tree, where the red polyline is the trunk, and green polylines are the first order branches, and black ones are the second order branches. Figure 10(b)–(e) are LOD models of the trunk and branches of a tree. The variation of side numbers and polyline nodes can be seen through a comparison of Figure 10(e) and (c).

In our approach, the original skeleton models are obtained from plant modeling software. This approach is an extension of the work of Bloomingthal$^{32}$ from the static model to multi-resolution model.

Combining our foliage simplification method with this branch simplification method, as well as a plant distribution model, and a plant modeling tool, we can construct and render single plant as
well as complex outdoor scenes, such as forests. Figure 11 shows the structure of our system for forest rendering. It consists of two phases, preprocessing and rendering.

There are four steps in the preprocessing phase. The first step is to create a plant scene using a plant distribution model. In our system, AMAP Landmaker™ is used to generate a forest by instancing a limited number of plant samples. The forest information, including plant position, orientation, and the instancing information, is stored in a scene file. The second step is to generate a geometric model for each plant sample by plant modeling. The geometric models are then simplified in the third step. Each tree is divided into two parts: the solid component, including trunk and branches; and the sparse component, such as leaves, flowers and fruits. The two parts are simplified separately due to their different topological properties. The sparse component is simplified using the method in the Simplification of Leaves with Generic Shapes and Error-controlled LOD Models for Rendering sections, and the branches are simplified using the method described at the beginning of this section. In the fourth step, the LOD models of both parts are stored on disk.

At the beginning of the rendering phase, the plant distribution information and the data of each plant sample are loaded. To render each frame, an octree-based culling algorithm is used to clip the trees that are outside of the current viewing frustum. Then for each tree in the viewing frustum, the appropriate LOD models of foliage and branches will be extracted according to the current viewing parameters and the tree position. Finally, these models will be rendered.

All instances of a tree sample share the same data for both branches and sparse organs, and each instance only needs to know its own LOD when rendering. So, although there may be a huge number of trees in a scene, the memory cost is not very large.
Experiments

All our algorithms are written in the C language based on OpenGL. We use Vertex Buffer Objects (VBOs) to store the vertex array $V$ and index array $I$ of the foliage, as well as the multiresolution representations of the branches of each tree. All experiments are implemented on a PC with a dual Pentium Xeon 1.7GHz CPU, 1GB memory, and an NVIDIA GeForce 8600GT GPU. Tree models are produced by AMAP.\textsuperscript{29}

In our experiments, the distance threshold $\Omega$ is set to 0.2 meters. Too large a $\Omega$ will cause a large number of valid pairs, and too small a $\Omega$ will result in a large number of isolated leaves, both of which will decrease the efficiency of foliage simplification. The number threshold $L$ is set to 100, and the upper bound of normalized density value $\eta$ to 2.

Figure 12 shows the simplification ranks of a 40 years old black poplar, from its full model to a very coarse approximation. The numbers of the foliage polygons in the four figures are 681200, 35385, 1573, and 69, respectively.

Figure 13 shows a comparison of the simplified results of a coniferous tree using a traditional foliage simplification algorithm and our method, respectively. It turns out that our result is more faithful to the original model.
Figure 14. LOD models of a 25 years old Siberian crab apple tree at four positions with different distances.

Figure 14 shows four LOD models of a crab apple with leaves and flowers. The detailed information of the four models are shown in Table 1.

Figure 15 shows a comparison of the HUO algorithm and our method on four models: a holly, a white poplar, a black poplar and a Scots pine. The images of the first row of Figure 15 are original models; the images of the second row are the simplified models of the HUO algorithm, and the images of the last row are the corresponding results of our method. The polygon numbers used for representing sparse organs is shown below each model. The visual qualities of simplified results with the two methods are similar, while the compression ratios of our method are higher. We also compare the preprocessing time of the two methods for the four trees. The result is shown in Table 2.

To show the efficiency of our LOD extraction algorithm and the improvement on rendering rate after adopting a GPU-supported data structure, we do experiments with a poplar and a Scots pine shown in Figures 16 and 17, respectively. The poplar has broad leaves and the Scots pine has thin leaves. In the experiments, a detail of foliage is converted to be a value between 0 and 1 for a unified measurement, where 0 is the most detailed approximation and 1 the coarsest. We use four different rendering strategies. STRATEGY I is a traditional way. It extracts the LOD by recursively traversing a binary tree structure, and it is not GPU-oriented. STRATEGY II and STRATEGY III are half GPU-supported, i.e. the vertex array $V$ is stored in the GPU. The difference between the two strategies is that STRATEGY II uses the same LOD extraction algorithm as STRATEGY I, while STRATEGY III uses the algorithm proposed in the Efficient Extraction of LOD Models subsection. STRATEGY IV uses the same LOD extraction algorithm as STRATEGY III, but it is fully GPU-oriented. Both the vertex array $V$ and the index array $I$ are stored in the GPU. We measure the time that the four strategies spend on extracting and rendering each foliage level of detail for each tree. The results are shown in Figures 16(c) and 17(c). It can be seen that STRATEGY IV is much more efficient than the others. The geometric data sizes of the two original models are 2.58M and 3.61M, and the storage costs of the four strategies are shown in Table 3.

Figure 18 is a close view of a virtual forest that consists of both broad and thin trees built from 7 species. The total number of foliage polygons of the original tree models in the forest is $8.7 \times 10^9$, and the number of the original branch polygons is $7.3 \times 10^9$. Figure 19 is a bird’s eye view of the same forest.

Figure 20(a)–(c) are three snapshots of another virtual forest in spring. It contains 5 different broad-leaf trees. The total number of the original foliage polygons in the forest is $9.9 \times 10^9$, and the number of the original branch polygons is $4.7 \times 10^9$.

The image resolution of the five forest figures is 1280 by 1024. Other details of the figures are shown in Table 4. In these figures, Shadows are generated from single planar maps mapped and alpha blended on the soil, supposed to be a planar receiver. Tree shadow

<table>
<thead>
<tr>
<th>Tree ID</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance (m)</td>
<td>10.30</td>
<td>17.63</td>
<td>27.52</td>
<td>51.80</td>
</tr>
<tr>
<td>Polygon number of sparse organs</td>
<td>129,489</td>
<td>53,554</td>
<td>16,687</td>
<td>2,565</td>
</tr>
<tr>
<td>Compression ratio</td>
<td>100%</td>
<td>41.36%</td>
<td>12.89%</td>
<td>1.98%</td>
</tr>
</tbody>
</table>

Table 1. Simplification details of the four trees in Figure 14
Table 2. Comparison of pre-processing time in seconds

<table>
<thead>
<tr>
<th>Tree name</th>
<th>Holly</th>
<th>White poplar</th>
<th>Black poplar</th>
<th>Scots pine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polygon number of sparse organs</td>
<td>35,800</td>
<td>33,275</td>
<td>282,800</td>
<td>191,940</td>
</tr>
<tr>
<td>Preprocessing time of the HUO</td>
<td>1515.9</td>
<td>7.5</td>
<td>1498.4</td>
<td>668.5</td>
</tr>
<tr>
<td>Our preprocessing time</td>
<td>1.7</td>
<td>0.2</td>
<td>2.5</td>
<td>1.3</td>
</tr>
<tr>
<td>Speedup ratio</td>
<td>891.7</td>
<td>37.5</td>
<td>599.4</td>
<td>514.2</td>
</tr>
</tbody>
</table>

map image are generated once at initialization step. At rendering stage the mapping size (length) and direction are defined from light position. On still images, in order to increase image realism and allow self shadowing, we do apply a post process based on anti-aliased shadow filtering with depth maps. Rendering time on these figures does not include the simple fake shadow calculation.

Conclusions and Future Work

In the context of rendering natural vegetation scenes, we present a foliage simplification and real-time rendering framework that exploits the capabilities of current graphics hardware.
Figure 16. Comparison of four strategies on the time spent on foliage rendering of a poplar: (a) Tree model, (b) Model complexity for different LOD levels, and (c) Curves of four strategies of time spent on foliage rendering with different detail degree.

The framework involves two kernel algorithms, the Broad Foliage Simplification Algorithm (BFSA) and the Thin Foliage Simplification Algorithm (TFSA). Both involve appropriate foliage clustering for fast union, compression degree related to local foliage density, and a GPU-oriented data structure for fast rendering. Compared with other foliage simplification and rendering methods, our framework shows four primary advantages:

**Generality:** Unlike other methods simplifying quadrilateral or triangle leaves only, our method can be used to deal with most species of leaves, which may be represented by complex mesh models.

**Simplification efficiency:** By discarding a large number of leaf pairs that are too distant to be united in the foliage simplification process, the speed of our simplification algorithm is drastically improved, resulting in a highly efficient preprocessing.
Figure 17. Comparison of four strategies on the time spent on foliage rendering of Scots pine: (a) Tree model, (b) Model complexity for different LOD levels, and (c) Curves of four strategies of time spent on foliage rendering with different detail degree.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Poplar</th>
<th>Scots pine</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU memory</td>
<td>GPU memory</td>
</tr>
<tr>
<td>STRATEGY I</td>
<td>5.43M</td>
<td>0.00M</td>
</tr>
<tr>
<td>STRATEGY II</td>
<td>1.02M</td>
<td>4.41M</td>
</tr>
<tr>
<td>STRATEGY III</td>
<td>3.63M</td>
<td>4.41M</td>
</tr>
<tr>
<td>STRATEGY IV</td>
<td>2.89M</td>
<td>8.27M</td>
</tr>
</tbody>
</table>

Table 3. Comparison of the CPU and GPU memory cost of the four strategies.
Compression efficiency: Our algorithm allows different resolutions in a crown according to its leaf distribution. So the overall compression ratio can be higher without sacrificing the visual effect.

Rendering efficiency: By converting the traditional hierarchical data structure to an array, the LOD model can be quickly extracted. Moreover, the array is fully GPU-oriented, so changing LOD models requires...
Figure 20. Screenshots of a forest in spring.

<table>
<thead>
<tr>
<th></th>
<th>Figure 18</th>
<th>Figure 19</th>
<th>Figure 20(a)</th>
<th>Figure 20(b)</th>
<th>Figure 20(c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of trees</td>
<td>22,560</td>
<td>22,560</td>
<td>16,888</td>
<td>16,888</td>
<td>16,888</td>
</tr>
<tr>
<td>Number of trees in view</td>
<td>1,452</td>
<td>1,058</td>
<td>279</td>
<td>605</td>
<td>311</td>
</tr>
<tr>
<td>Foliage compression ratio</td>
<td>3.10%</td>
<td>0.87%</td>
<td>0.39%</td>
<td>0.63%</td>
<td>0.50%</td>
</tr>
<tr>
<td>Number of frames per second</td>
<td>12.82</td>
<td>21.28</td>
<td>62.50</td>
<td>32.26</td>
<td>66.67</td>
</tr>
</tbody>
</table>

Table 4. The information of the forest figures
almost no communication between the CPU and the GPU, and tens of thousands of trees can be rendered in interactive frame rates.

Further gains are possible thanks to following potential improvements.

- First, the user specified parameters of the modified octree structure, i.e. the distance threshold $\Omega$ and the number limit $L$, could be specified automatically in the future.
- Second, the selection of the best leaf pair could take the advantage of the underlying topological structure of the plant, grouping leaves in “botanically-faithful” unions and decreasing the number of leaf candidates to merge.
- Third, there is still some room to exploit the GPU. For example, the occlusion queries function of the GPU can be used to speed up the rendering further.
- Fourth, the density-related compression idea can be applied to other scenes consisting of a large range of objects, not just forests.
- Finally, our method now is optimized for viewing trees from a moderate distance, but not trees from close up. This can be solved in the future by subdividing each tree crown into different parts and extract a LOD for each part according its distance to the camera, respectively.

**ACKNOWLEDGEMENTS**

All plant models in this paper are from the output of AMAP-Genesis™. This work is supported by National Natural Science Foundation of China Projects No. 60672148, 60872120; National High-Tech Research and Development 863 Plan of China under Grant No. 2006AA01Z201, 2008AA01Z201, and 2008AA10Z218; and the French National Research Agency within project NATSIM ANR-05-MMSA-45.

**References**


Authors’ biographies:

Dr. Gang Yang now is a lecturer at School of Information Science and Technology, Beijing Forestry University. He received his PhD in computer science from the Institute of Software, Chinese Academy of Sciences in 2006. His bachelor and master degrees are all obtained from Computer Science Department of Shan Dong University. His research interests include realistic rendering, texture synthesis, virtual plants, and nonphotorealistic rendering, and has published more than 20 papers on national and international conferences and journals.

Dr. Qingqiong Deng received her bachelor degree from Huazhong University of Science and Technology, Wuhan, China, in 2003, and received her PhD from the Sino-French Laboratory LIAMA and National Laboratory of Pattern Recognition at CAS Institute of Automation in 2008. Now she is a postdoctor in the College of Information Science and Technology of Beijing Normal University. Her major field of research is real-time rendering of outdoor scenes.

Dr Xiaopeng Zhang received his MS degree in Mathematics from Northwest University in Xi’an in 1987, and his PhD in computer graphics from the Institute of Software, Chinese Academy of Sciences (CAS), in Beijing in 1999. He is an associate professor in the Sino-French Laboratory LIAMA and National Laboratory of Pattern Recognition at CAS Institute of Automation. His main interest is computer graphics and pattern recognition. Dr. Zhang was invited as a foreign specialist for forest rendering in INRIA, the French National Institute for Research in Computer Science and Control. He received the National Scientific and Technological Progress Prize (Second Class) in 2004. He has more than 80 publications and he advised 7 PhD students.
Dr Marc Jaeger is a computer graphics scientist (IEEE member) interested in natural phenomena. Graduate from University Louis Pasteur of Strasbourg (PhD, 1988), he specialized on plant growth simulation and visualisation at CIRAD Amap laboratory. From 1990 to 1998, he led a small team working on volume imaging. He joined then the GreenLab project (LIAMA, China), dedicated to plant functional structural modeling, simulation and visualisation. Back to France within DigiPlante team (associate project of INRIA, CIRAD, and Ecole Central of Paris), his current research topics concern mainly landscape functional simulation and visualization.