Technical Section

View-dependent pruning for real-time rendering of trees

Jesús Gumbau *, Miguel Chover, Inmaculada Remolar, Cristina Rebollo

Institute of New Imaging Technologies, Universitat Jaume I, Castellon, Spain

ARTICLE INFO

Article history:
Received 25 May 2010
Received in revised form 18 October 2010
Accepted 17 November 2010
Available online 27 November 2010

Keywords:
Vegetable species
Real-time rendering
Multiresolution modeling
Hardware oriented

ABSTRACT

The main problem in the real-time rendering of vegetation is the massive amount of primitives to be rendered. These primitives are needed to fully describe the geometry of the plants. However, some of them are not visible depending on the location of the viewer. This work focuses on this fact to interactively reduce the amount of geometry needed to represent the foliage through a view-dependent multiresolution scheme. Following a camera-dependent criterion, the less visible parts of the foliage are detected in real time, and rendered with a decreased level of detail for improving efficiency. This fact considerably reduces the extraction and the visualization time of the geometry that represents the foliage. The novelty of the presented method is that its design is oriented to being efficient on massively parallel architectures, such as the graphics processing unit. Moreover, we introduce a new management system for efficiently handling level of detail objects in order to improve performance for forest scenes.

1. Introduction

Efficient rendering of vegetal species is a key feature for enhancing the realism of outdoor scenes. However, realistic visualization of plants in a natural environment has always posed a challenging problem, due to the massive amount of geometry needed to represent a plant. This is specially true in dense forests, where the massive amount of primitives can easily overwhelm the most advanced rendering system available.

Multiresolution level of detail—LoD—models [30] are well-known methods for altering the polygonal complexity of objects in order to improve performance in highly detailed meshes. The basic idea behind these methods is that highly detailed models do not always need to be represented at full detail. Multiresolution models alleviate this problem by diminishing the amount of triangles in a progressive way, so that the viewer does not perceive great changes while decreasing the level of detail.

Every LoD scheme is based on a simplification method in order to construct the data structure. The appropriate geometry, depending of some criteria, is extracted in real time. Usually, LoD schemes are designed for continuous surfaces. This is the reason they do not deal correctly with sparse meshes, such as the foliage. This fact introduce their own simplification scheme adapted to the nature of the foliage, such as those based on leaf-collapses [27] or others based on pruning [3].

This paper presents a new view-dependent multiresolution model for the foliage of the trees that takes advantage of the graphics hardware. In order to built the LoD scheme, a stochastic pruning method [3,4] is applied in a pre-process, which is proved to deal correctly with sparse meshes, such as the foliage. This fact allows us to interactively remove unneeded data. Moreover, the presented multiresolution scheme provides a view-dependent solution. In real time, the less visible parts of the foliage are detected and rendered with a coarser approximation in order to improve efficiency. The appropriate resolution of the foliage is calculated taking into account both the distance of the tree to the observer as well as the visibility of the leaves. Our algorithm is designed to run efficiently on highly parallel architectures, such as the graphic process unit (GPU). Due to the design of the scheme, the level of detail can be calculated in parallel, so that the extraction time is considerably reduced.

1.1. Contributions

The novelty of our algorithm remains on the fact that it is a completely GPU-based view-dependent multiresolution model for the foliage. The data storage, the LoD extraction and the rendering algorithms have been designed to perform on the GPU. This approach has some direct advantages. First, it removes any traffic between the CPU and the graphics processor, avoiding the PCIe bottleneck.

Secondly, the multiresolution models to date have been designed for single threaded systems or do not specify how their algorithms are
executed on parallel architectures. This work provides the basics for building view-dependent multiresolution models on highly parallel environments such as the GPU.

Finally, we propose a LoD management system that allows for efficiently managing the level of detail of thousands of tree instances without stalling the computing resources and maximizing performance. This management system is also entirely executed on the GPU.

The rest of the paper is organized as follows. We discuss the previous work in Section 2. Next, we present an overview of the method in Section 3. In Section 4, the processes involved in constructing the presented LoD scheme are described and runtime processes are detailed in Section 5. Section 6 deals with the implementation details and the level of detail manager for rendering forest are explained in Section 7. Next, we discuss the differences between our approach and existing techniques in Section 8. Then, we show and discuss the results in Section 9. Finally, conclusions are presented in Section 10.

2. Previous work

Extensive research has been carried out offering real-time visualization of detailed plant species. To solve this problem, many approaches have been proposed that can be classified in two different groups: image-based and geometry-based algorithms. At a glance, image-based methods use less geometry and provide good results at a far-medium distance, while geometry-based methods are able to offer the best results at shorter distances.

2.1. Image-based rendering

This is one of the most common methods of representing trees because of its simplicity. Impostors are the most popular example of image-based rendering. In this method the geometry of the object is replaced with an image of it textured on a polygon within the scene.

This technique has been used in different works presented by several authors until now. Max [22, 23] adds depth information to the pre-calculated images. In this context, Shade et al. [38] and Chang et al. [2] introduce layered depth images (LDI) to render objects from pre-computed pixel-based representations with depth from different viewpoints. This information allows them to recalculate different views from the stored images of the scene.

Other authors work with volumetric textures. Meyer et al. [25] convert complex natural objects into mipmapped volumetric textures before they are raytraced. The works presented in 2001 by Meyer et al. [26] and in 2004 by Reche et al. [31] obtain 2D images from volumetric textures and combine them depending on the position of the camera. However, this results in low quality images for close-ups.

One of the main problems of this methods is that suffers from parallax artifacts that are visible at medium-close distances. Garcia et al. [14, 13] solve the parallax problem by using impostors that group sets of leaves and using indirect texturing to drastically increase the detail of the leaves without incrementing the memory footprint. In recent years, some works have appeared that represent trees using billboard clouds, textures that are always facing the viewer, as those presented by Décoret et al. [7], Fuhrmann et al. [10], Dylan et al. [19] and Mantler et al. [24]. Finally, one of the latest works is presented by Linz et al. [20]. The authors estimate opacity in a volume to generate and visualize view-dependent textures attached to cells of that volume.

Regarding commercial applications, Speedtree [37] is an example of successful implementation of this technique. This software is nowadays one of the most popular in the field of tree visualization in real time.

In order to solve the problem of the loss of realism, some works combine image- and geometry-based rendering, such as Remolar et al. [34].

2.2. Geometry-based rendering

This approach does not lose realism as the viewer moves towards the model, but the number of polygons that form the tree objects makes it necessary to use certain techniques to obtain interactive visualization.

Some works as [27] decrease the number of leaves that form the foliage, increasing the size of the remaining leaves. In this way, the pruning appearance is avoided. Nevertheless, in most of the works published to date the display primitive is changed to points or lines, as in the work presented by Weber and Penn [39] or Stamminger et al. [36]. Deussen et al. [6] and Gilet et al. [12] show the interactive adaptation of the number of points depending on the importance of the object in the final rendered image.

In recent years, several papers based on multiresolution models have appeared. Some of them work with multiresolution models of images, such as the work presented by Meyer et al. [26] and Lluch et al. [18]. Most of the LoD models are based on geometry, as the work presented by Remolar et al. [28]. The authors present a multiresolution representation of a tree exclusively based on isolated polygons. They can represent different resolutions in a same tree following a view-dependent approaching. In 2006, Rebollo et al. [33] improved this representation by adapting the data structures to the graphics hardware. The next year, the same authors present an improved version [32] of this work which uses a GPU-oriented storage for multiresolution data, so that it can be efficiently rendered. They also propose to use a multiresolution model for the trunk based on triangle strips. However, although this work uses an efficient rendering approach, the LoD extraction step is still done on the CPU. Rebollo et al. [29] propose a new approach for fast foliage simplification on the fly and negligible extraction cost. The article is based on generating simplified leaves selecting different vertices based on a vertex-skipping approach. Although this technique performs really fast, it is not able to preserve foliage appearance on high compression ratios.

Finally, [9] presents a new multiresolution model for the foliage that allows for high compression ratios. Their approach is based on recursively collapsing pairs of leaves and finally replacing them by other primitives, as lines or points. These collapses are pre-calculated in a pre-process by subdividing the foliage using a binary tree scheme for rapidly finding pairs to simplify.

3. Method overview

This work proposes a new view-dependent geometry-based multiresolution algorithm for foliage rendering specially designed for highly parallel architectures. The algorithm takes into account the distance of the foliage to the camera, as well as its relative position, in order to provide a view-dependent level of detail solution. The presented scheme is oriented to trees which leaves are represented by two triangles in a quad.

The algorithm is composed of two different stages (Fig. 1). Firstly, there is a pre-process step which prepares the input data (a mesh representing the foliage) for constructing the view-dependent approach. This pre-process performs the following operations. First, foliage is divided into a cloud of cells, represented as oriented bounding boxes (OBBs). Next, the visibility of each cell is computed from a set of external view points surrounding the foliage (Fig. 2). Finally, leaves inside each cell are stochastically
sorted \[3\] in order to perform the simplification operation at run-time.

Secondly, there is a run-time stage which includes the algorithms needed to interactively alter the level of detail. First, each cell evaluates in real time the position of the camera in order to decide the LoD factor associated to it. LoD factors are used as a percentage value for deciding the amount of leaves needed to represent the contents of a cell. Next, based on this factor, a list of triangles that compound the visible leaves is generated for rendering, selecting them from the original set. Finally, the size and the color of the remaining geometry is altered in order to preserve the visual appearance of the original mesh.

From an implementation point of view, performing all these operations on different independent cells means the LoD extraction algorithm is parallelizable, and thus GPU-friendly. The direct benefits of implementing it on the graphics hardware are that the traffic between the CPU and the GPU is minimized and the time needed to perform the needed calculations is drastically decreased, because of the parallel nature of the domain. Therefore, this method is able to take advantage of both processing power of the GPU and a wide bandwidth of on-board graphics memory.

The following sections describe the whole process in detail.

4. Pre-process

In order to build the data structure, the first step is to generate a cloud of 3D cells over the foliage, so that every leaf in the foliage is clustered in a cell. Next, the visibility of each cell is tested from different angles from the exterior of the foliage, as shown in Fig. 2. Thus, each cell is bound to a value that determines the visibility of the leaves it contains from a set of viewpoints located around the foliage. Finally, a stochastic number is assigned to every leaf in the cell and the leaves are re-organized using this number, following the work presented in \[3\].

4.1. Cell cloud generation

The first step is to generate a cloud of cells around the foliage. In this process, it is important to generate the cells taking into account the shape of the foliage and the distribution of the leaves. The objective is to maximize the number of leaves contained in each cell while minimizing the size of the cells needed to be spread over the foliage. For that purpose, we use the method introduced by Gottschalk et al.\[11\] which generates a cloud of oriented bounding boxes that are fitted as tightly as possible to the shape of the object. The number of the cells in the foliage will determine the number of processes performed in parallel in the GPU.

For the cell cloud generation, we implement an OBBTree structure which allows us to generate a tree of tightly packed cells distributed over the foliage. This OBBTree allows us to obtain a cell cloud of a variable number of cells by simply selecting all the cells of a given maximum depth.

For constructing the OBBTree, first the bounding volume of the whole foliage is computed. Then, a process recursively partitions the bounding volume, using the major axis criterion, and builds a tree of a given maximum depth. Fig. 3 shows this recursive process. When the tree is computed, the resulting cell cloud is selected by traversing the tree and selecting the leaf nodes. Fig. 4 shows an example where the cell cloud is computed from an input tree by using an OBBTree.

The number of cells generated in a foliage is the criterion used for parallelizing our LoD algorithm. In the GPU, each kernel execution processes the level of detail of each single cell in parallel. In our scheme, the amount of cells determines the softness of the view-dependent approximation because having more cells means a finer granularity. Therefore, the more threads in parallel, the better visual results obtained. In this scheme, we decided to setup our kernels to use 512 threads per block, the maximum amount of threads per block in CUDA. Then, in this pre-process, it is generated an OBBTree which has 512 cells as leaves of the tree data structure. All the leaves are included in one of these cells distributed around the foliage.

4.2. Visibility determination

After performing the previous step, the visibility of each cell is computed in order to provide a visibility factor that determines the visibility of the leaves it contains from a set of external cameras.
These cameras or viewpoints are uniformly distributed around the foliage in order to capture the shape of the foliage from a finite number of directions, as shown in Fig. 2. In order to decide the amount of view points and their distribution around the foliage, we took into consideration the following works from Lindstrom et al. [21] and Castelló et al. [5]. The good results obtained in their works using uniformly distributed view points led us to the decision to keep this criterion for distributing the cameras around the foliage. Moreover, they suggest in their work that using more than 20 cameras to perform an image-driven simplification does not provide more accurate information. In our tests, we have found that usually the number of necessary pre-cached cameras is a small value. After various experiments, we tested that 16 cameras were enough to capture the general shape of the foliage in order to provide good estimations.

The visibility factor is a function of the type \( \text{vis}(\text{cell}, \text{camID}) \) which associates each cell/camera pair with a floating-point value in the range \([0,1]\). This factor determines how much of the foliage it contains is visible from a given viewpoint. Cell visibility is computed, for every cell/camera pair, by taking into account the number of pixels of the leaves in this cell that are not occluded by the rest of the foliage. The cost of this pre-process is \( O(n_{\text{cells}} \times n_{\text{cameras}}) \), with \( n_{\text{cells}} \) being the number of cells in the cloud and \( n_{\text{cameras}} \) the number of cameras located around the tree.

Our implementation uses hardware occlusion queries to obtain the number of pixels that pass the z-buffer test. This part of the process is done on the GPU. Algorithm 1 clarifies the process involved in calculating the visibility values.

\[
\text{vis}(\text{cell}, \text{camID}) = \frac{\text{visPixels}}{\text{totalPixels}} \in [0,1]
\]  

(1)

Basically, for each viewpoint, \( \text{camID} \), the contents of each cell are rendered four times. The first and second passes are used to obtain the number of pixels visible from the current viewpoint without occlusion from other cells: the first pass is used to set up the depth buffer and the second one is used to obtain the number of visible pixels (\( \text{totalPixels} \)). The other two passes are used to calculate the number of non-occluded pixels visible from the current viewpoint: the third pass renders all cells to fill the depth buffer and the last pass renders the current cell again to count the number of pixels that pass the depth test and thus the number of pixels which are visible from the current viewpoint (\( \text{visPixels} \)). Therefore, the visibility factor for each cell–camera pair is calculated using Eq. (1).

### Algorithm 1. Cell visibility determination on the GPU

```plaintext
begin
  cells ← ListofCells
  leaves ← ListofLeaves
  cameras ← ListofCameras
  for all ce ∈ cells do
    for all ca ∈ cameras do
      clearBuffers(color, depth)
      renderLeavesCell(ce, ca) ← firstpass
      resetQuery()
      renderLeavesCell(ce, ca) ← secondpass
      totalPixels ← queryRendered()
      renderLeavesAll(ca) ← thirdpass
      resetQuery()
      renderLeavesCell(ce, ca) ← fourthpass
      visPixels ← queryRendered()
      visCellView[ce, ca] ← visPixels
    end for
  end for
end
```

4.3. Cell-based stochastic sorting of leaves

Once the leaves are clustered in cells, they are sorted following the criterion introduced by Cook and Halstead [3]. A stochastic criterion is used to assign a random number to each leaf, which determines the order of simplification of each leaf in the cell. This process makes it possible to optimize the rendering of models made up of a large amount of disconnected geometry, such as plants and trees.

In order to make this simplification process easier, leaves are stored in the GPU taking into account this random number. Each leaf is represented by two triangles, whose indices are finally stored in the graphics hardware. This fact allows for optimal performance performing memory accesses.

5. Run-time

This section describes the process performed at runtime on the GPU in order to generate and visualize the appropriate level of detail. Three stages are clearly differentiated: LoD determination, triangles list generation and appearance preservation.

5.1. LoD determination

In order to prevent unneeded geometry from being rendered, a \( \text{LoD}_{\text{cell-factor}} \) is calculated to determine in real time the appropriate level of detail of the geometry contained in every cell. In order to obtain a view-dependent approach, we have implemented in this scheme a function of the type shown in Eq. (2). It depends both on the visibility of the cell in the current situation of the camera and on the distance of the object to it. However, this function can be easily adapted to different requirements of the scene.

\[
\text{LoD}_{\text{cell-factor}}[\text{viewLoD}(\text{cell}), \text{dist}(\text{near,far})] \in [0,1]
\]  

(2)

Firstly, the visualization of one cell, \( \text{viewLoD}(\text{cell}) \), is interactively determined by using the pre-calculated LoD factors of the cell in the pre-process. For every cell, the visibilities of the three closest viewpoint directions are linearly combined and weighted, to approximate an estimation of the current visibility. Fig. 5 clarifies this process.

\[
\text{viewLoD}(\text{cell}) = \sum_{k=1}^{3} \text{camWeights}_k \cdot \text{vis}(\text{cell}, \text{camID}_k)
\]  

(3)
where $\text{camWeights}_k$ and $\text{camIDS}_k$ represent the importance and the unique identifier of the selected camera $k$, respectively (see Algorithm 2 for details).

Secondly, the function $\text{dist}(\text{near, far})$ maps the relative positions of the tree ($\text{treePos}$) and the observer ($\text{camPos}$) into a value in the range $[0, 1]$. It takes into account two user-defined values which represent the distances for minimum and maximum LoD (near and far planes). This function is defined in the following equation:

$$
\text{dist}(\text{near, far}) = \begin{cases} 
\text{near} & \text{if } \text{dist}(\text{near, far}) < \text{near} \\
0 & \text{if } \text{dist}(\text{near, far}) > \text{far} \\
1 & \text{else}
\end{cases}
$$

(4)

Finally, the appropriate LoD ($\text{LoD}_{\text{cell, factor}}$) is determined by using both the visibility of the cell and the distance to the camera and is calculated as follows:

$$
\text{LoD}_{\text{cell, factor}} = \text{dist} \cdot \text{viewLoD}
$$

(5)

These parameters causes the system to behave as a variable multiresolution model because the level of detail depends on both the position and the orientation of the observer. Notice that setting $\text{viewLoD} (\text{cell})$ to 1 for each pair (cell, camera) causes the system to behave as a uniform multiresolution system which only depends on the distance to the viewer.

At the end of the process, one $\text{LoD}_{\text{cell, factor}}$ is obtained for each cell around the foliage. These values are calculated in parallel on the GPU and stored in a buffer located in video memory. The storage cost of this buffer is $O(n_{\text{cells}})$, with $n_{\text{cells}}$ being the total number of cells distributed around the foliage.

5.2. Triangles list generation

The objective of this process is to obtain the leaves needed to visualize the foliage at a given level of detail. All the cells are processed taking into account the $\text{LoD}_{\text{cell, factor}}$ previously calculated. As a result of this process, a list of triangle indices that represent the leaves in the current approximation is generated.

As leaves have been previously ordered using the stochastic number in the pre-process, outputting a certain level of detail is accomplished by just copying into the render buffer the first $n_{\text{leavesCell}} \cdot \text{LoD}_{\text{cell, factor}}$ leaves contained in each cell, where $n_{\text{leavesCell}}$ is the amount of leaves in that cell that represent the best approximation.

Taking advantage of the graphics hardware, the resulting indices list is generated for all cells simultaneously. For this reason, it is necessary to determine, for each one of them, the offset position in the resulting index buffer to start writing indices to. This fact avoids collisions writing to the buffer, as it is shown in Fig. 6. Let $o_n$ be the offset position for cell $n$ and $u_n$ the amount of unpruned triangles of cell $n$, $o_{n+1}$ is calculated as follows:

$$
o_{n+1} = o_n + u_n
$$

(6)

It is important to notice that calculating a valid offset for a cell requires the sum of the offsets of previous cells. Although this problem seems to be inherently sequential, there exist some works in the literature that deal with this problem and how it can be efficiently implemented in parallel systems using the all-prefix-sums operation described in [1]. More information about the algorithm can be found in [15,16] for a detailed description of the algorithm and its efficient CUDA implementation.

Once cell offsets are calculated, the system is ready to start generating indices. Finally, for each cell a global variable is incremented in video memory to indicate the total amount of indices generated. This information is necessary in order to know the amount of geometry finally generated in the LoD extraction process for rendering purposes.

5.3. Appearance preservation

The runtime modifies the area of the rendered leaves in order to reduce the visual impact of pruning. Thus, the visual appearance remains the closest possible to the original unpruned model. In this LoD scheme, the technique presented in [3] has been applied. The total area of all leaves of the object can be expressed as Eq. (7), where $a$ is the average area of each single leaf and $n$ is the number of leaves in the most detailed representation.

$$
\text{area}_{\text{total}} = na
$$

(7)

Let $u$ be the parameter in the interval $[0, 1]$ that quantifies the amount of geometry that remains after the pruning process is applied to the foliage. This parameter takes into account the distance of the object to the observer ($z$). The function is defined in Eq. (8). Let $h$ be the parameter that controls the aggressiveness of the pruning function.

$$
u = z^{-\alpha} s^2
$$

(8)

Therefore, $nu$ is the number of leaves in the current level of detail and $\text{area}_{\text{total}}$ is the total area of the unpruned foliage in this approximation. As pruning decreases the number of leaves, the area of the foliage also decreases. It must be compensated in order to maintain the visual concordance between levels of detail. Then, rendered elements are scaled by the factor $s$ to compensate the pruning of primitives.

$$
\text{area}_{\text{total}} = (nu)(as), \quad s = 1/u
$$

(9)

In practice, this step is performed in the vertex shader and has an almost negligible rendering cost.
When the tree is rendered at a medium distance it happens that leaves are so small that the texture on them is no longer distinguishable. As it has been previously said, leaves are represented by a quad textured by an image with alpha channel for opacity. Due to the texture sampling performed by the graphics hardware on this kind of textures, when they are so small, lots of pixels receive an incorrect averaged alpha value which causes the foliage to visually lose leaf density as it moves away from the observer. This is solved by performing a color correction adjustment in the pixel shader as follows. We compare the size of the pixel with the size of the leaf for gradually disabling the alpha values of the leaf texture till no alpha channel is used and the whole leaf is rendered with a single color.

6. Run-time implementation details

The implementation of the runtime stage is based on CUDA, the programming API that takes advantage of the unified multiprocessor architecture of the GPU. It enables efficient management of all the resources of the GPU without the limitations of the pipeline, offering an interface for the rendering API such as OpenGL or Direct3D to share data and resources. Our method takes advantage of this architecture to access and write geometry in the on-board memory of the graphics device.

The process performed at run time by our implementation is detailed in Algorithm 2. Every kernel invocation in the algorithm is accompanied by the symbols ⟨and⟩ just behind it. This symbols contain a number that represents the number of threads the kernel executes for completing the task.

The algorithm uses the following data structures:
- LoDCells is a buffer allocated in video memory. Its length is equal to the number of cells. It is used to store the appropriate LoD of each cell according to the current situation of the camera.
- cameras is a buffer allocated in host memory. Its length is equal to the number of view points. It is used to store the position of each view point surrounding the tree.
- cells is a buffer in video memory storing the indices to the leaves contained in each cell.
- offsets is a buffer in video memory to store the offsets for each cell in the resulting index buffer of leaves.
- lodIndices is a buffer in video memory used to store the indices of the triangles representing the foliage in the appropriated LoD.

Algorithm 2. Algorithm for processes performed at run time.

\[ nC \leftarrow \text{NumberofCells} \]

FindCameras([in]point, [in]cameras, [out]camIDs, [out]camWgs)
CalcOffsets <nC> ([out]offsets, [in]LoDCells)
MapOGLBufferToCUDA(loIndices)
KernelDoLoD <nC> ([in]cells, [in]offsets, [out]lodIndices)
UnMapOGLBuffer(loIndices)

Algorithm 2 works as follows. First, function FindCameras seeks for the three nearest view points given the current camera location, vpoint. This function outputs three camera identifiers along, camIDs, with three weights defining the influence of each camera in the current situation, camWgs. As it was said above, 16 cameras were enough to capture the general shape of the foliage in order to provide good estimations.

Next, the function AssignLOD calculates a single LoD_cell_factor for each cell (in the range [0,1]) on the GPU. Results are stored in the buffer LoDCells. The function takes into account the distance of the object to the camera, vpoint and combines it with the viewpoint-dependent cell factors associated to the three nearest pre-cached cameras weighted according to their influence, camWgs. This LoD_cell_factor is calculated in parallel for every cell.

Then, function CalcOffsets calculates the offsets, where each cell must start writing data to the buffer lodIndices in order to avoid collisions. This way, each cell is assigned a space in the final buffer and data can be written in a parallel way on the GPU.

Finally, the OpenGL buffer is mapped in order to make it accessible from CUDA, function MapOGLBufferToCUDA. Indices data are written in the buffer through the invocation of the function KernelDoLoD which copies the resulting indices to the index buffer sequentially.

After this process finishes, the buffer lodIndices is unmapped and the tree is ready to be rendered at the current level of detail.

As was said in a previous section, the criterion used for parallelizing our LoD algorithm is based on the cells. Each kernel execution processes the level of detail of each single cell in parallel. The official CUDA documentation recommends around 192 or 256 minimum threads per block. However, given that the amount of registers used by our kernels is less than 16, we decided to setup our kernels to use 512 threads per block, the maximum amount of threads per block in CUDA. In this way, we are able to maximize occupancy of the CUDA resources as well as enabling the algorithm for scaling well on future graphics hardware.

In order to achieve high memory bandwidth in CUDA, memory is divided into equally sized memory banks, which can be accessed simultaneously. Bank conflicts arise when various kernels attempt to access to the same memory bank at the same time, which causes bad performance. In our scheme, each single thread represents a single cell of the cloud and manages its level of detail. Each cell contains its own set of leaves, which are mutually exclusive. This way, neither calculating the LoD_cell_factor nor generating the final indices list access to the same memory location and thus, avoiding the bank conflict problem.

Finally, the size of the leaves is altered at run-time using a vertex shader. In a pre-process step, each of the four vertices of a leaf is assigned a pre-computed vector \( \vec{e} \) in the following way:

\[
\forall i \in \{0,1,2,3\} \Rightarrow \vec{e}_i = \vec{v}_i - 0.25 \sum_{j=0}^{3} \vec{v}_j
\]  

The resulting \( \vec{e} \) vector is passed to the vertex shader and used for determining the direction on which each vertex must be moved in order to alter the size of the leaf.

7. Forest rendering

In this paper we have introduced a multiresolution model for foliage rendering completely oriented to the GPU. However, efficiently handling a forest is not trivial and some extra work needs to be done in order to avoid unnecessary calculations. Every multiresolution model has associated an extraction time for providing a certain level of detail. Even though our algorithm is executed on the GPU and it provides better extraction times than other CPU oriented algorithms (like [34,27,33]), it would be inefficient to be extracting the appropriate level of detail of hundreds or even thousands of trees every frame.

To solve that problem, we provide a level of detail management system which minimizes the number of LoD extractions in the whole scene. This LoD manager runs entirely on the GPU in order to maximize performance. When the viewpoint changes, some LoDs...
of the tree instances have to be updated. Let $\text{LoD}_{\text{err}}$ be the difference between the current ($\text{LoD}_{\text{stored}}$) and the desired ($\text{LoD}_{\text{desired}}$) level of detail for a tree instance in the present view. This term is defined as follows:

$$\text{LoD}_{\text{err}} = |\text{LoD}_{\text{desired}} - \text{LoD}_{\text{stored}}|$$

where $\text{LoD}_{\text{desired}}$ is defined as the sum of $\text{LoD}_{\text{cell, factor}}$ for all tree cells at a given time:

$$\text{LoD}_{\text{desired}} = \sum_{i} \text{LoD}_{\text{cell, factor}}$$

When a forest with a determined number of tree instances ($t$) is rendered, the total error of this scene can be defined as

$$\text{Forest}_{\text{err}} = \sum_{i} \text{LoD}_{\text{err}}$$

(11)

Our LoD manager requires an user-defined threshold $\delta$, that the error of the forest, $\text{Forest}_{\text{err}}$, cannot exceed. When this error is greater than $\delta$, a set of tree instances are selected for updating their level of detail based on different factors: their individual distance to the observer ($\text{dist}$) and their $\text{LoD}_{\text{err}}$. The LoD manager computes the factors $\text{dist}$ and $\text{LoD}_{\text{err}}$ on all tree instances in parallel on the GPU, and obtained their LoD urgency ($\text{LoD}_{\text{u}}$) as

$$\text{LoD}_{\text{u}} = \frac{\text{LoD}_{\text{err}} \cdot \text{old}}{\text{dist}}$$

(12)

The term $\text{old}$ in Eq. (12) denotes the amount of time the LoD factor of a single tree remains unchanged. In practice, this is implemented as a CUDA array, and it is incremented every time the LoD management is executed and it is reset when a tree instance is selected for changing its LoD.

This formulation ensures that tree instances that are closer to the observer's position will update their level of detail more frequently because, as they are potentially affecting more pixels than further trees, they are considered more important in the scene. The $\text{old}$ term in equation ensures that all trees are going to be recalculated once in a while, preventing further trees of being never updated.

Furthermore, our level of detail management system is able to efficiently determine the amount of tree instances that intersect the viewing frustum and therefore are selected for being rendered. This is important for large forest scenes where thousands or even tens of thousands of trees are used.

Regarding the implementation, the LoD manager runs as follows. First, it runs a CUDA kernel for every tree instance of the forest, calculates $\text{LoD}_{\text{err}}$, updates $\text{old}$ values for every one of them and, finally, calculates $\text{Forest}_{\text{err}}$. In the same step, viewpoint visibility is also calculated by intersecting each tree bounding volume against the frustum. These operations are efficiently performed due to their simplicity and the massive parallel computing power of current GPUs. If the $\text{Forest}_{\text{err}}$ exceeds the threshold established by the user, a second CUDA kernel is used for performing an ordering over each tree marked as visible in the previous step. The sorting criterion used is $\text{LoD}_{\text{urgency}}$.

Our sorting algorithm is based on previous work for efficient parallel sorting found in the literature [17,35]. Then, a new user-defined parameter is involved in the process. This new parameter defines the time for performing LoD extraction tasks before rendering each frame. During this time, the LoD extraction process is executed for the first trees in the array, until the time defined by the user is burnt-out.

8. Discussion

This section discusses the differences and the advantages of our method against other geometry-based LoD approaches for the foliage.

Our approach has been designed from scratch as a completely GPU-based view-dependent multiresolution model for the foliage. This means that the data storage, the LoD extraction and the rendering algorithms have been designed to be stored and executed on the GPU. This approach has some direct advantages. Firstly, compared to [27,32] it removes any traffic between the CPU and the graphics processor, avoiding the PCIe bottleneck of

![Fig. 7. Performance comparison charts for extraction times between our method and [32]. LoD 1 means 100% and LoD 0 means 0%. 0.1% used as minimum LoD. (a) Olea Europaea, (b) Fraxinus Ornus, (c) Quercus Cerris and (d) Cedrus Atlantica.](image-url)
these methods due to their necessity of uploading to GPU memory all vertex indices each time the LoD extraction process is performed.

In addition, it provides a level of detail management system that is also designed to be executed in parallel on the GPU and prevents the GPU to perform unnecessary LoD operations on large forest scenes. Rebollo et al. [32] use a foliage subdivision system for preventing uploading the whole foliage to the GPU each time the LoD changes. However, without a LoD manager both [27,32] provide especially bad performance, because the system is easily stalled with data transfers.

The work presented by Deng et al. [9] uses a technique similar to [8] for efficiently rendering LoD meshes on the graphics hardware. Although this approach is very efficient and flexible and does not require bus traffic for extracting the LoD, it has three major drawbacks. Firstly, as the rendering is based on sequential point trees [8], it uses a large number of rendering calls to visualize each tree. This is specially inefficient when a large number of trees are being rendered, such as a forest. Moreover, they use a combination of triangles and lines for rendering the foliage, increasing the amount of needed drawing calls. Secondly, as the authors say in their paper, their approach is designed to view trees at moderate distances, not being suitable for close-ups. In contrast, Fig. 9 shows the realism achieved with our method even for close-ups. Lastly, their LoD approach is not view-dependent and does not allow for decreasing the level of detail in hidden parts of the foliage, which is important for better preserving the appearance of the tree while reducing compression ratios.

9. Results

This section demonstrates our technique with practical tests for measuring performance and visual quality. We have configured our testbed framework in the following way. All tests have been performed on an Athlon64 3500+ with 3GB of RAM and a GeForce 8800GT graphics card. The trees used in the experiments are geometrically described in Table 1. This section is divided in two parts for separating the tests performed on a tree level and those tests performed on a forest level.

Table 2 shows the time employed in the pre-process stage. The processes involved in this step are not performed in real time. After they are executed, the data structure is prepared to interactively extract the appropriate level of detail.

9.1. Single tree analysis

For our single tree analysis we configured a rotating camera around each tree in order to provide a good estimation for the
visibility dependent factor of our method. Moreover, for considering the distance factor of the foliage to the observer, the tree is moved away from the camera incrementally from the near to the far planes in order to test the whole range of active distances.

In our experiments, we have checked the time (in milliseconds) employed in reducing the level of detail to 66%, 33% and 15% as well as the time needed for pre-processing each foliage using 16 uniformly distributed cameras and a cell cloud of 512 boxes. Each measurement involves the time of extracting the geometry corresponding to the LoD, i.e., the time of mapping the hardware vertex buffers to CUDA, the execution of all the necessary processes in the CUDA kernels and the time of un-mapping the buffers to be rendered.

Table 3 shows the level of detail extraction times for these four different types of trees. For each value on the table, a number of independent measurements (20 for each value in our case) were taken and averaged in order to provide a good estimation of the LoD extraction times. Moreover, an additional comparison with our method running on a CPU is shown. It can be clearly seen that due to the parallel hardware-accelerated nature of our method it outperforms the extraction times obtained by the CPU-based method. Notice that, since our method has been designed to run efficiently on a highly parallel architecture, it cannot perform well on a single-core CPU.

Table 3
LOD extraction times in milliseconds for different tree models.

<table>
<thead>
<tr>
<th>Tree</th>
<th>66%</th>
<th>33%</th>
<th>15%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Olea europaea</td>
<td>0.6</td>
<td>0.56</td>
<td>0.51</td>
</tr>
<tr>
<td>Fraxinus ornus</td>
<td>0.90</td>
<td>0.72</td>
<td>0.66</td>
</tr>
<tr>
<td>Quercus cerris</td>
<td>0.92</td>
<td>0.70</td>
<td>0.65</td>
</tr>
<tr>
<td>Cedrus atlantica</td>
<td>2.3</td>
<td>1.63</td>
<td>1.23</td>
</tr>
</tbody>
</table>

Table 4
Comparison of our storage cost with [9].

<table>
<thead>
<tr>
<th>Tree</th>
<th>CPU memory</th>
<th>GPU memory</th>
<th>CPU memory</th>
<th>GPU memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Olea europaea</td>
<td>0.00</td>
<td>1.29</td>
<td>0.00</td>
<td>9.21</td>
</tr>
<tr>
<td>Cedrus atlantica</td>
<td>2.89</td>
<td>8.27</td>
<td>9.08</td>
<td>11.33</td>
</tr>
<tr>
<td>Our method (M)</td>
<td>0.63</td>
<td>1.27</td>
<td>4.51</td>
<td>9.19</td>
</tr>
<tr>
<td>Deng et al. [9] (M)</td>
<td>8.27</td>
<td>9.08</td>
<td>11.33</td>
<td>0.63</td>
</tr>
<tr>
<td>Rebollo et al. [32] (M)</td>
<td>1.27</td>
<td>4.51</td>
<td>9.19</td>
<td>0.63</td>
</tr>
</tbody>
</table>

Fig. 10. Performance comparison of forest scenes with our approach and with full LoD. (a) Dense forest and (b) Sparse forest.

Fig. 11. Visual results of our technique for a forest with a LoD reduced to 10%.
Taking advantage of the view-dependent nature of our pruning algorithm we are able to greatly reduce the geometrical complexity of the resulting tree while still achieving good visual results, shown in Fig. 13. This figure shows a comparative study of the pruning quality of our method, comparing the quality of the resulting unpruned geometry against the original tree at the same distance to the camera and with the same orientations. Notice how the appearance of the trees is preserved even when the complexity of the foliage is reduced to 10% of the original complexity.

Table 4 shows a comparative study of storage costs with two trees of different polygonal complexity using our method and the methods presented by Deng et al. [9] and Rebollo et al. [32]. It can be seen that as our method is completely implemented on the GPU, our CPU storage costs are null. Moreover, the use of stochastic pruning allows our method to require less memory to be stored overall than its competitors.

9.2. Forest analysis

For the forest results analysis we set up two kind of scenes: dense and sparse forests. For the dense forest we populated the scene with 6450 trees while only 210 trees were used for the sparse forest scene. Both scene types are populated with the following types of trees: *Fraxinus ornus, Quercus cerris, Olea europaea and Cedrus atlantica* (see Table 1 and Fig. 13 for details). Shadow map resolution in our scenes is $4096 \times 4096$ pixels, however, it is not recalculated every time, but only on the first frame, assuming that neither the trees nor the light source (the sun) change. Table 4 shows a detailed information rendering scenes in Figs. 11 and 12. These scenes are selected because they represent examples of two different types of scenarios: sparse (Fig. 12) and dense forests (Fig. 11).

Fig. 11 shows a visual comparison of our technique managing the level of detail of our dense forest scene. At the camera location used to render the scene showed in Fig. 11, 508 tree instances are detected to lie inside of the frustum and thus marked as visible. The LoD manager for the forest uses around 12 ms each frame to perform the tasks described in Section 9. These tasks include the tree visibility determination, urgency calculations and tree ordering for selecting the most urgent trees to be updated. The LoD manager selects an average of eight trees per frame for LoD recalculation, an average of 440 LoD extractions per second at 50 frames per second. It can be seen in Fig. 11 how the view-dependent nature of our technique is able to remove potentially invisible detail from the most hidden parts of the trees, so that the impact on the final image is minimized.

Performance improvements obtained using our approach are shown in Fig. 10 for the two forests scenes: dense and sparse. It can be seen that our technique is especially useful when used for large forests scenes (Table 5).

![Fig. 12. Snowy scene showing our pruning algorithm in a sparse forest environment with 10% LoD.](image)

![Fig. 13. The upper rows show the tree rendered at full geometrical complexity. The lower rows show the same tree at the same distances at the following reduction factors: 75%, 50% and 10%.](image)

<table>
<thead>
<tr>
<th>Table 5</th>
<th>Performance results rendering scenes on Figs. 11 and 12.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fig. 11</td>
<td>Fig. 12</td>
</tr>
<tr>
<td>Trees in scene</td>
<td>6450</td>
</tr>
<tr>
<td>Trees in view</td>
<td>508</td>
</tr>
<tr>
<td>LoD extraction/management (ms)</td>
<td>12</td>
</tr>
<tr>
<td>Average compression ratio (%)</td>
<td>10</td>
</tr>
<tr>
<td>Rendering frequency (LoD) (fps)</td>
<td>56</td>
</tr>
<tr>
<td>Rendering frequency (no LoD) (fps)</td>
<td>13</td>
</tr>
</tbody>
</table>
10. Conclusions

This work presents a view-dependent multiresolution level of detail method for real-time rendering of the foliage designed for highly parallel systems. The method is based on stochastically pruning unneeded leaves for a given LoD, depending on the distance to the viewer and its relative position with the foliage. In this way, high amounts of leaves can be rapidly discarded while preserving the general shape of the tree.

The design of this LoD scheme is based on the GPU. All the processes involved in obtaining the geometry that represents the appropriate resolution of the foliage run in this graphics hardware. Bandwidth traffic between the CPU and the GPU, typically found in multiresolution models, is completely removed since CUDA allows for gathering and scattering operations from any direction of the video memory. It also allows for freeing the GPU from extracting the level of detail by translating this task to the GPU. Thus, it can be performed in parallel spreading the task among the whole amount of multiprocessors of the graphics hardware. As the technique runs completely on the GPU, it becomes more scalable, taking into account that the velocity at which the GPU increases its performance is much higher compared to the CPU.

Finally, we introduce a level of detail management for the forest in order to avoid stalling the system when dealing with dense forests composed of many hundreds of trees. This management system is designed to take advantage of the parallel nature of the GPU for maximizing performance and scalability.

As future work, the next step is to improve the realistic representation of the trees taking illumination into account. We are developing solutions screen space global illumination techniques combined with per-leaf lighting pre-calculations for improving the quality of our visual results. Furthermore, another topic that actually has been studied is the simulation of the effect of the wind on the foliage, taking advantage of the hardware graphics capabilities.

Acknowledgments

This work was supported by Ministerio Español de Ciencia y Tecnología (project TIN2007-68066-C04-02) and by Bancaja (P1 1B2007-56).

Appendix A. Supplementary material

Supplementary data associated with this article can be found in the online version of 10.1016/j.cag.2010.11.014.

References