Rendering Falling Leaves on Graphics Hardware

Pere-Pau Vázquez and Marcos Balsa*

*MOVING Group
Universitat Politècnica de Catalunya
Jordi Girona, 1-3, E-08034 Barcelona, Spain
+34 93 413 78 89, email: ppau@lsi.upc.edu, markisbcn@gmail.com
www: www.lsi.upc.edu/∼ppau

Abstract

There is a growing interest in simulating natural phenomena in computer graphics applications. Animating natural scenes in real time is one of the most challenging problems due to the inherent complexity of their structure, formed by millions of geometric entities, and the interactions that happen within. An example of natural scenario that is needed for games or simulation programs are forests. Forests are difficult to render because the huge amount of geometric entities and the large amount of detail to be represented. Moreover, the interactions between the objects (grass, leaves) and external forces such as wind are complex to model. In this paper we concentrate in the rendering of falling leaves at low cost. We present a technique that exploits graphics hardware in order to render thousands of leaves with different falling paths in real time and low memory requirements.

Keywords: Natural Phenomena, Tree Leaves, Real Time Rendering, Hardware Accelerated Rendering.

First presented at the International Conference of Theory in Computer Graphics and Applications 2007, extended and revised for JVRB

1 Introduction

Natural phenomena are usually very complex to simulate due to the high complexity of both the geometry and the interactions present in nature. Rendering realistic forests is a hard problem that is challenged both by the huge number of present polygons and the interaction between wind and trees or grass. Certain falling objects, such as leaves, are difficult to simulate due to the high complexity of its movement, influenced both by gravity and the hydrodynamic effects such as drag, lift, vortex shedding, and so on, caused by the surrounding air. Although very interesting approaches do simulate the behaviour of light weight objects such as soap bubbles and feathers have been developed, to the authors' knowledge, there is currently no system that renders multiple falling leaves in real time. In this paper we present a rendering system that is able to cope with thousands of falling leaves at real time each one performing an apparently different falling path. In contrast to other approaches, our method concentrates on efficient rendering from precomputed path information and we show how to effectively reuse path information in order to obtain per leaf different trajectories at real time and with low memory requirements. Our contributions are:

- A tool that simulates the falling of leaves from precomputed information stored in textures.
- A simple method for transforming incoming information in order to produce different paths for each leaf.
- A strategy of path reuse that allows for the construction of potentially indefinite long paths from the initial one.

We have implemented three different versions of our algorithm, with different load balanced from vertex to fragment shader:

1. Complete rendering in the vertex shader, with the program computing the current vertex position using textures.

2. Position computation per leaf in the fragment shader and, in a second pass, the vertex shader determines the position of the vertices accessing the information generated in the previous pass.

3. Position computation per leaf vertex in the fragment shader, and modification of a vertex buffer (*render-to-vertex-buffer*) that is rendered in the next step.

The details of implementation of the different strategies are developed in Section 6 where time comparisons are also shown.

The rest of the paper is organized as follows: First, we review related work. In Section 3, we present an overview of our rendering tool and the path construction process. Section 4 shows how we perform path modification and reuse in order to make differently looking and long trajectories from the same data. Section 5 deals with different acceleration strategies we used. Section 6 compares the efficiency of the different implementations. Finally, Section 7 discusses the results and concludes our paper pointing to some lines for future research.

2 Related Work

There is a continuous demand for increasingly realistic visual simulations of complex scenes. Dynamic natural scenes are essential for some applications such as simulators or games. A common example are forests due to its inherently huge amount of polygons needed to represent them, and the highly complex interactions that intervene in animation. There has been an increasing interest in animating trees or grass, but there has been little work simulating light weight falling objects such as leaves or paper. Most of the papers focus on plant representation and interactive rendering, and only a few papers deal with the problem of plant animation, and almost no paper focuses on falling leaves.

2.1 Interactive rendering

Bradley has proposed an efficient data structure, a random binary tree, to create, render, and animate trees in real time [Bra04]. Color of leaves is progressively modified in order to simulate season change and are removed when they achieve a certain amount of color change. Braitmaier *et al.* pursue the same objective [BDE04] and focus especially in selecting pigments during the seasons that are coherent with biochemical reactions. Deussen *et al.* [DCSD02] use small sets of point and line primitives to represent complex polygonal plant models and forests at interactive frame rates. Franzke and Deussen present a method to efficient rendering plant leaves and other translucent, highly textured elements by adapting rendering methods for translucent materials in combination with a set of predefined textures [FO03].

Jakulin focuses on fast rendering of trees by using a mixed representation: polygon meshes for trunks and big branches, and a set of alpha blended slices that represent the twigs and leaves. They use the limited human perception of parallax effects to simplify the slices needed to render [Jak00]. Decaudin and Neyret render dense forests in real time avoiding the common problems of parallax artifacts [DN04]. They assume the forests are dense enough to be represented by a volumetric texture and develop an aperiodic tiling strategy that avoids interpolation artifacts at tiles borders and generates non-repetitive forests. A recent paper by Rebollo *et al.* ([RGR$^+$07]) also focuses on fast rendering of tree leaves but without animation.

2.2 Plant motion

There have been some contributions regarding plant or grass motion, but without focusing on rendering falling leaves, we present here the ones more related to our work. Wejchert and Haumann [WH91] developed an aerodynamic model for simulating the motion objects in fluid flows. In particular they simulate the falling of leaves from trees. They use a simplification of Navier-Stokes equations assuming the fluids are inviscid, irrotational, and uncompressible. However, they do not deal with the problem of real time rendering of leaves. A recent approach deals with autumn scenaries. It renders leaves and simulates the aging process, and it is similar to ours in the sense that the falling path is not built based on physical simulation, rather, the authors use certain patterns to simulate the behaviour of leaves falling [DGAJ06].

Ota *et al.* [OTF$^+$04] simulate the motion of branches and leaves swaying in a wind field by using noise functions. Again, leaves do not fall but stay attached to the trees and consequently the movements are limited. Reeves and Blau [RB85] proposed an approach based on a particle modeling system that made the particles evolve in a 2D space with the effect of gusts of wind with random local variations of intensity. Our system is similar to the latter in the sense that our precomputed information is used by applying pseudo random variations, constant for each leaf, thus resulting in different paths created from the same initial data.

Wei *et al.* [WZF$^+$03] present an approach that is similar to ours in objectives, because they render soap bubbles or a feather, but from a simulation point of view. This limits both the number of elements that can be simulated and the extension of the geometry where they can be placed. They model the wind field by using a Lattice Boltzmann Model, as it is easy to compute and parallelize. Finally, Shinya and Fournier [SF92] animate trees and grass by complex flow fields by modeling the wind field in a Fourier space and then converting to a time-varying force field.

3 Rendering leaves

In this Section we give an overview on the tasks carried out by our rendering tool.

3.1 Overview

Figure 1 depicts the system. As explained later in this paper, paths are precomputed using Maya and stored as textures. The renderer takes care of managing the list of falling leaves that are sent to the graphics processor and the shaders (sketched as *OpenGL*) modify the input paths in a pseudo-random manner to generate differently looking paths for each leaf. Therefore, static and dynamic geometry is processed with different shaders.
In order to design a practical system for real time rendering of falling leaves in real time, three conditions must be satisfied:

- Paths must be visually pleasing and natural.
- Each leaf must fall in a different way.
- Computational and memory costs per leaf must be low.

The first objective is tailored to ensure realism. Despite the huge complexity of natural scenes, our eyes are used to them, and therefore, if leaves perform strange moves in their falling trajectory, we would note it rapidly. This will be achieved by using a physically based simulator that computes a realistic falling path of a light-weight object under the influence of forces such as gravity, vortex, and wind.

The second objective must be fulfilled in the presence of a set of leaves, in order to obtain plausible animation: if many leaves are falling, it is important to avoid visible patterns in their moves, because it would make the scene look unnatural. In order to reduce memory requirements and computation cost, we will use the same path for all leaves. Despite that, we make it appear different for each leaf by performing pseudo random modifications to the path in real time.

The third one is important because it imposes restrictions on the rendering tool, if we want a system to scale well with the number of leaves, the position computation must have low cost. This can be fulfilled by storing the trajectory information in a texture that is used to modify leaf position. One of our implementations directly reads this information in the vertex shader using the so-called Vertex Texture Fetch extension [GFG04], available in modern NVidia graphics cards (and compatible with the standard OpenGL Shading Language) in order to compute the actual position and orientation of each leaf in the vertex shader. The other, more efficient ones, read this path at fragment shader level. Then, the new position is computed, and an extra pass effectively renders the leaves onto their corresponding positions.

Our algorithm not only reads the input file and transform vertices according to it, but also performs stochastic modifications on the transformations as a function of the initial position of the leaf. This permits the simulation of differently looking paths for each leaf. From now on, we refer to the program that computes the leaf position as the *shader*. Therefore, our explanations will be valid for the three different implementations we did.

Initially, a simple path is calculated and stored in a texture. We could use more than one path, and reduce the arithmetic computations in the shader, although this results in small gain in the experiments we carried out. For each leaf, the shader updates the vertex position and orientation according to the moment t of the animation using the data of the path. Thus, each leaf does fall according to the selected path in a natu-

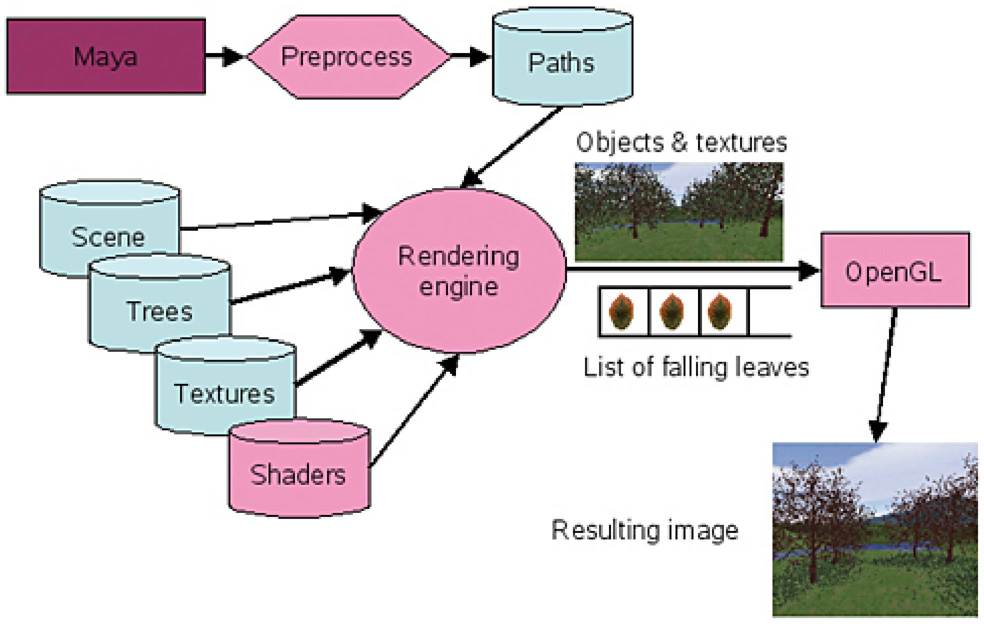

Figure 1: Overview of the rendering system.

rally looking fashion. Next, we explain the path construction process, the contents of the texture path, and overview the algorithm of leaf transformation. Section 4 deals on reusing data to produce different trajectories for each leaf.

3.2 Path construction

There are two possible different methods to construct the initial path we need for our rendering tool: i) Capturing the real information from nature, or ii) Simulating the behavior of a leaf using a physically-based algorithm.

Our initial intention was to acquire the 3D data of a falling leaf and use it for rendering. Unfortunately, the data of leaves in movement is very difficult to acquire due to many reasons, mainly: their movement is fast, thus making it difficult for an affordable camera to correctly (i.e. without blurring artifacts) record the path, and second, it is not possible to add 3D markers for acquisition systems because they are too heavy to attach them to a leaf (a relatively large leaf of approximately 10×15 centimeters weighs 4 to 6 grams). Moreover, in order to capture a sufficiently long path, we would need a set of cameras covering a volume of 4 or 5 cubic meters, which also implies difficulties in the set up.

A different approach could be the simulation of falling leaves using a physically based system. This poses some difficulties too because Navier-Stokes equations are difficult to deal with. Some simplifications such as the systems presented by Wejchert and Haumann [WH91] or Wei *et al.* [WZF$^+$03], have been developed. However, although the falling leaves behavior has not been categorized, for planar discs, some research has been carried out. In Field *et al.* [FKMN97] the dynamics of falling planar discs were analyzed, and the authors concluded that the trajectory is chaotic. Leaves have a much more complex geometry than planar discs, which makes their behavior difficult to analyze. Fortunately, there are other commercial systems that simulate the behavior of objects under the influence of dynamic forces such as Maya. Although this approach is not ideal because it is not easy to model a complex object as a leaf (the trajectory will be influenced by its microgeometry and the distribution of mass across the leaf), and simpler objects will have only similar behavior, it is probably the most practical solution.

On the other hand, Maya provides a set of dynamic forces operators that can be combined in order to define a relatively realistic falling trajectory for a planar object. In our case, we have built a set of falling paths by rendering a planar object under the effect of several forces (gravity, wind, and so on), and recovered the information on positions and orientations of the falling object to use them as a trajectory. We modeled a set of scenes with several dynamic operators affecting part of the path until getting a plausible falling trajetory.

Our paths consist of a set of up to 200 positions (with the corresponding orientation information at each position). As we will see later, these resulting paths are further processed before the use in our rendering tool in order to extract extra information that will help us to render a per-leaf different path with no limit in its length. This extra information is added at the end of the texture. This way all the information needed for the rendering process that will be used by the vertex shader is stored in the same texture (actually we use a texture for positions and another one for orientations).

Note that, independently of the acquisition method, if we are able to obtain the aforementioned information, that is, positions and orientations, we can plug it into our system.

3.3 Path information

Our system basically performs a rendering for each leaf based on path information that is stored in a texture. For each leaf that falls, the same algorithm with the data of the corresponding leaf is executed.
As we have already explained, we start with the precomputed data of a falling leaf and provide it to the shader in the form of a texture. Concretely, the trajectory information is stored in a couple of 1D textures, each containing a set of RGB values that encode position (x, y, z) and rotation (Rx, Ry, Rz) respectively, at each moment t. Given an initial position of a leaf (x_0, y_0, z_0), subsequent positions may be computed as $(x_0 + x, y_0 + y, z_0 + z)$. Rotations are computed the same way. In order to make the encoding easy, the initial position of the path is $(0, 0, 0)$, and subsequent positions encode displacements, therefore, the value of y component will be negative for the rest of the positions. On their hand, orientations encode the real orientation of the falling object. The relevant information our shader receives from the CPU is:

Number of frames of the path: Used to determine if the provided path has been consumed and we must jump to another reused position (see Section 4.2).

Current time: Moment of the animation.

Total time: Total duration of the path.

Object center: Initial position of the leaf.

Path information: Two 1D textures which contain positions and orientations respectively.

The shader also receives other information such as the face normal. Each frame, the shader gets the corresponding path displacements by accessing the corresponding position (t, as the initial moment is 0) of the texture path. This simple encoding, and the fact that the same texture is shared among all the leaves at shader level, allows us to render thousands of leaves in real time, because we minimize texture information change between the CPU and the GPU. Having a different texture per leaf would yield to texture changes at some point. In Section 4.2 we show how to use the same path in order to create different trajectories, and how to reuse the same path when the initial y position of the leaf is higher than the represented position in the path.

3.4 Overview

At rendering time, for each leaf, a shader computes the new position and orientation using the current time t. It performs the following steps:

1. Look for the initial frame f_i (different per leaf)
2. Seek current position in path ($f_i + f_t$, assuming that f_t gives the number of frames passed in time t)
3. Calculate actual position and orientation

In step 2, if we have a falling path larger than the one stored in our texture, when we arrive at the end, we jump to a different position of the path that preserves continuity, as explained in Section 4.2. Once we know the correct position and orientation, step 3 performs the corresponding geometric transforms and ensures the initial and final parts of the path are soft, as explained in the following subsection.

3.5 Starting and ending points

As we have mentioned, the information provided by our texture path consists in a fixed set of positions and orientations for a set of defined time moments. Ideally, a different path should be constructed for each leaf according to its initial position, orientation, and its real geometry. This way, everything could be precomputed, that is, the shader should only replace the initial position of the element by the position stored in a texture. Unfortunately, for large amounts of leaves, this becomes impractical due to the huge demand of texture memory and, moreover, the limitation in number of texture units would turn rendering cost into bandwidth limited because there would be a continuous necessity of texture change between CPU and GPU. Thus, we will use the same path for all the leaves (although we could code some paths in a larger texture and perform a similar treatment).

In our application, a path is a set of fixed positions and orientations. Being this path common to all leaves, and being the initial positions of leaves eventually different, it is compulsory to analyze the initial positions of leaves in order to make coherent the initial orientations of leaves in our model, that depend on the model of the tree, and the fixed initial orientation of our falling path. Note that the position is unimportant because we encode the initial position of the path as $(0, 0, 0)$ displacement. As in most cases the orientations will not match, we have to find a simple and efficient way to make the orientation change softly. Therefore, a first adaptation movement is required.

Our shader receives, among other parameters, a parameter that indicates the moment t of the animation. Initially, when t is zero, we must take the first position of the texture as the information for leaf rendering. In order to make a soft adjustment between the initial position of the leaf and the one of the path, at the initial frames (that is, we have a y displacement smaller than a certain threshold) the shader makes an interpolation between the initial position of the leaf and the first position of the path (by rotating over the center of the leaf). As the leaf moves only slightly in Y direction for each frame, the effect is soft (see Figure 2 left). Rotations are implemented using quaternions.

The same case happens when the leaf arrives to the ground. If the ground was covered by grass, nothing would be noted, but for a planar ground, if leaves remain as in the last step of the path, some of them could not lie on the floor. Note that it is not guaranteed that each leaf will consume all the path because they start from different heights. To solve it, when the leaf is close to the ground, we perform the same strategy than for the initial moments of the falling path, that is, we interpolate the last position of the leaf before arriving to the ground with a resting position that aligns the normal with the normal of the ground. We can see the different positions a leaf takes when falling to the ground in Figure 2 (right).

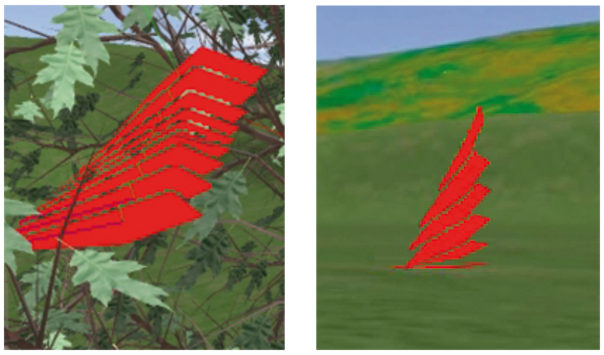
Figure 2: Several superposed images of a support polygon of a leaf (marked in red) in their initial adaptation to the path (left) and the final correction to make it parallel to the ground (right).

4 Path modification

Up to now we have only presented the construction and rendering of a single path. However, if we want our forest to look realistic, each leaf should fall in a different manner. A simple path of 200 positions requires 1.2 Kb for the positions and orientations. If we want to render up to ten thousand leaves, each one with a different path, the storage requirements grow up to 12 Mb. For larger paths (such as for taller trees), the size of textures would grow. Therefore, what we do is to use a single path that will be dynamically modified for each leaf to simulate plausible variations per leaf. This is implemented by adding two improvements to our rendering tool: path variation and path reuse.

4.1 Different falling trajectories

As our objective is to reduce texture memory consumption and rendering cost, we will use the same texture path for each leaf. This makes the memory cost independent on the number of leaves that fall. However, if we do not apply any transformation, although not all leaves start falling at the same time, it will be easy to see patterns when lots of leaves are on their way down. Updating the texture for each falling leaf is not an option because it would penalize efficiency. Consequently, the per leaf changes that we apply must be done at shader level.

In order to use the same base path to create different trajectories, we have applied several modifications at different levels:

- Pseudo-random rotation of the resulting path around Y axis.

- Pseudo-random scale of X and Y displacements.
- Modification of the starting point.

When we want to modify the falling path we have to take into account that many leaves falling at the same time will be seen from the same viewpoint. Therefore, symmetries in paths will be difficult to notice if they do not happen parallel to the viewing plane. We take advantage of this fact and modify the resulting data by rotating the resulting position around the Y axis.

In order to perform a deterministic change to the path, we use a pseudo-random modification that depends on the initial position of the leaf, which is constant and different for each one. Thus, we rotate the resulting position a pseudo-random angle Y, computed as follows: $permAngle = mod(center\_obj.x * center\_obj.y * center\_obj.z * 37., 360.)$. The product has been chosen empirically. This pseudo-random number is precomputed and passed as the fourth coordinate of the rotation texture.

Although the results at this point may be acceptable, if the path has some salient feature, that is, some sudden acceleration or rotation, or any other particularity, this may produce a visually recognizable pattern. Thus, we add a couple of modifications more: displacement scale and initial point variation.

Although it is not possible to produce an arbitrarily large displacement scale, because it would produce unnatural moves, a small, again pseudo-random modification is feasible. The values of X and Z are then modified by a scaling factor, computed as: $scale\_x = mod(center\_obj.z * center\_obj.y, 3.)$ and $scale\_z = mod(center\_obj.x * center\_obj.y, 3.)$ respectively. This results in a non uniform distribution of the leaves at a maximum *fixed* distance of the center of the tree caused by the falling path (whose displacements did not vary in module up to this change). All these modifications result in many differently looking visually pleasing paths. Figure 3 shows several different paths computed by our algorithm and Figure 4 shows the results in a scene.

4.2 Path reuse

Our texture path information does not depend on the actual geometry of the scene, in the sense that we build paths of a fixed length (height), and the resulting trajectories might be larger if the starting points of leaves are high enough. However, we solve this potential problem by reusing the falling path in a way inspired by the Video Textures [SSSE00]. Concretely, once the texture path is consumed, we jump to a different position of the same texture by preserving continuity. Thus, we precompute a set of *continuity points* in the path that preserve displacement continuity: the positions or orientations must not match, but the increments in translations (speed) must be similar in order to preserve continuous animation.

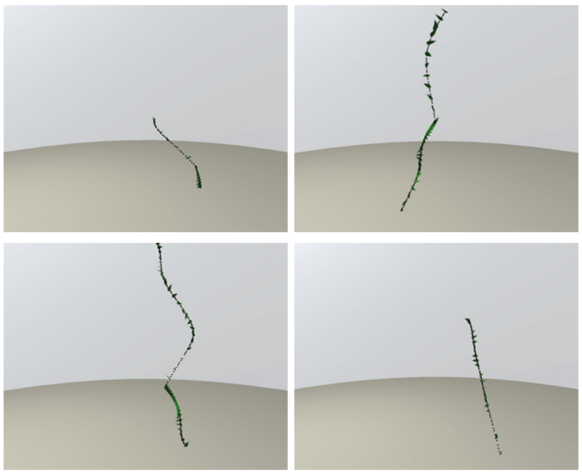
Figure 3: Different variations of the initial path.

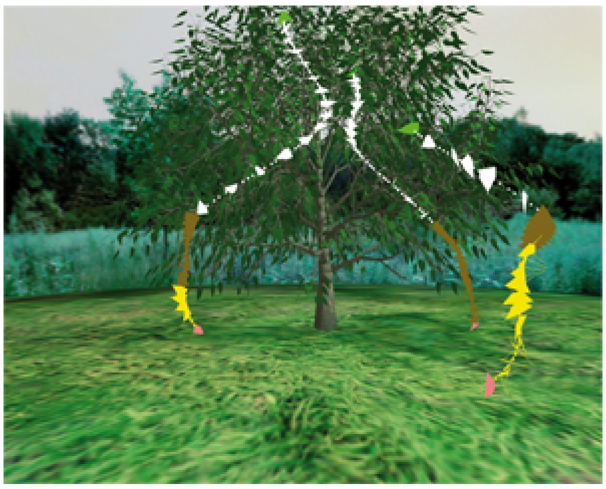
Figure 5: Leaves rendered at different moments in a falling path. Each color indicates a different path usage, pink leaves are on the floor.

The computation of new positions once we have consumed the original path is quite simple. They are computed using the last position of the animation and the displacement increment between the entry point and its previous point in the texture: $path[lastPos] + path[contPointPos] + path[contPointPos - 1]$. This information is also encoded in the same texture, just after the information of the path. The texture will

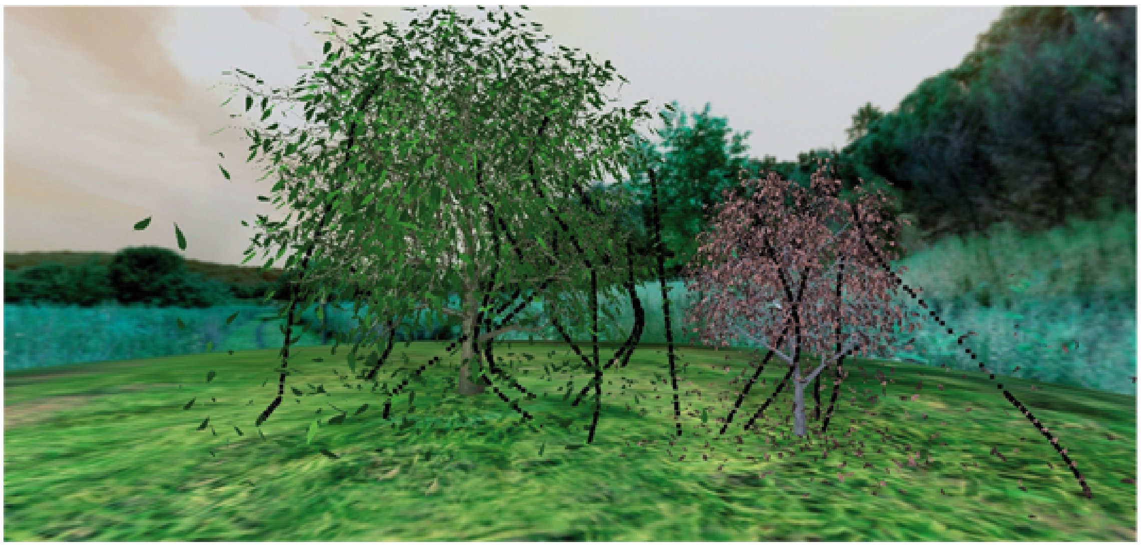

Figure 4: Multiple paths rendered at the same time. Note the different appearance of all of them.

contain then the points of the path and the continuity information (that simply consists in a value that indicates the following point). Then, when a falling leaf reaches the end of the path and has not arrived to ground, the shader computes the next correct position in the path according to the information provided by the texture. If required, changes to the interpretation of the following positions (different displacements) could be added and encoded as the continuity information in the same style than the ones performed to modify paths and therefore we could have a potentially indefinite path with multiple variations. Different stages of these paths are shown in Figure 5 as different colors (pink leaves lie on the floor).

Apart from this information, we add another modification to the vertex shader. In the same spirit than the *continuity points*, we compute a set of *starting points* among which the initial position (*frame0*) is pseudo randomly chosen (OpenGL specification of the shading language provides a noise function but it is not currently implemented in most graphics cards) using the following formula: $mod((centre\_obj.y * centre\_obj.x * 37.), float(MAX\_I - 1))$ where *MAX_I* is the number of initial frames of the animation. These starting points are chosen among the ones with slow speed and orientation roughly parallel to the ground, in order to ensure the soft continuation of the movement of the leaf. Therefore, each path starts in one of the set of precomputed points, making thus the final trajectories of the leaves quite different.

5 Optimizations

Our implementation includes optimizations such as vertex buffer objects lists for static geometry, frustum culling, and occlusion culling. However there is a bottleneck that raises when many leaves are continuously thrown during a walkthrough. If we implement the algorithm as is, while rendering a long walkthrough the framerate decays due to the increasing cost incurred in the shaders. As all thrown leaves are processed by the same shader even if they already are on the floor. The processing cost increases because the accesses to texture are determined by the current frame. When the computed position is *under* the floor, then the shader program looks for the first position that results in the leaf lying on the floor. This requires a binary search with several texture accesses, and texture access at vertex shader level is not optimized as in fragment shaders.

In order to reduce the impact of processing, we have implemented two optimizations:

- Leaf occlusion determination and skipping: During the rendering process, leaves are queried for occlusion determination and, when a leaf is not visible, we skip the rendering for 2 to 4 frames. In the results shown, only half of the leaves are tested each frame, and if they are not visible, we skip three frames.
- Leaf elimination: When we determine a leaf has been falling for a while, we compute its real position in the CPU and, if it is already on the floor,

we update its position, put it into the list of static geometry, and remove it from the list of falling leaves.

The first optimization allows us to reduce the rendering cost, as in most situations, not all falling leaves will be visible (they may be occluded by the trunk or branches, other leaves, or out of the frustum). In order not to overload the CPU, we perform the occlusion queries only on sets of leaves that change each frame. We found a good compromise was to test on one third of the falling leaves each frame and to skip the rendering of the leaves during 3 frames. Occlusion detection returns true if at least three pixels are set as visible. We will show in the following section how these optimizations improve the rendering process.

To solve the second problem (the leaves that are on the ground still incur in a cost penalty), we have implemented a simple solution that consists in selectively detecting and removing the leaves that are on the ground from the list of leaves that are treated by the shader. At each frame, the CPU selects a subset of leaves that started to fall long enough to have arrived to the ground (in our case, two seconds since they started falling is usually enough). For those leaves, the correct position is computed on the CPU, and if they lie on the floor, their position is updated, and the leaf is removed from the list of falling leaves. This step is not too costly, as our experiments show that for one thousand falling leaves, 5 to 13 leaves need to be erased at each frame. Then, the number of leaves tested per frame can be reduced to 10 or 20 and we start with the leaves that were thrown first. This allows us to keep the frame rate roughly constant for long exploration paths and thousands of falling leaves.

6 Implementation

We have implemented three versions of our algorithm:

- A vertex-based method.
- A fragment-based method.
- A render-to-vertex buffer method.

The first one is the simplest, but it is not the optimal version for most graphics cards. The main problem is that vertex texture access is either not available in old ATI graphics cards, or it is relatively slow even in modern graphics cards such as the NVIDIA's 6 and 7 series. The implementation of the vertex method is straightforward. The second and third approaches require two rendering passes, however, they are more efficient than the vertex-based approach. The fragment and render to vertex buffer methods are not so simple and we provide now some details.

6.1 Fragment-based method

In order to implement the falling leaves algorithm on the fragment shader, we need to perform two steps:

1. We render a quad that covers all the scene. Each falling leaf will correspond to one of those fragments. The fragment shader identifies the leaf it must simulate through its fragment coordinates and performs the simulation of the falling path. It then issues two colors that code the actual position and orientation of the leaf and the result is stored in two textures.

2. A second rendering step renders each leaf with an id that identifies its position in the texture. It reads the position and orientation and modifies the current vertex.

This algorithm performs two extra texture access by the vertex shader in the second step. Nonetheless, this results in a significant improvement in the frame rate (see Table 1). This is mainly due to the fact that only one simulation is performed per leaf, no matter how many triangles it has. Therefore, the computation cost is highly alleviated.

6.2 Render-to-Vertex buffer method

The render-to-vertex buffer (*r2v*) also has two steps:

1. We render a quad that covers all the scene. In this case, each vertex corresponds to one fragment. The fragment shader identifies the vertex it must simulate through its fragment coordinates and performs the simulation of the falling path as in the previous approach. It then issues two colors that code the actual position of the vertex and its normal. Those colors will modify the current vertex buffer object (containing the leaves geometry).

2. A second rendering step renders the vertex buffer object. The remaining geometry (the tree trunk and branches) is rendered as usual.

In contrast to the previous approach, this does not require vertex texture access. However, its performance is lower than the fragment-based approach (see Table 1). This is mainly due to two facts. First, we render all the leaves in a single vertex buffer object, and, although this is positive, it does not allow the occlusion query leaf elimination. Second, the fragment shader has to modify each vertex, not each leaf. As a result, the textures are larger and the computational cost higher. Despite that, this approach is more general in the sense that it can be used with older cards, even ATI cards, as they do not support vertex texture fetch but do support render to vertex buffer. This method has another potential problem: as all leaves are encoded in the texture (the ones falling and the ones which are still on the tree or on the floor), for large forests, with leaves represented with a large number of triangles, the texture size might exceed the maximum of the graphics card.

6.3 Results

We have compared the three implementations with different graphics cards and different scenes. In all cases the fragment-based approach is better, although the render to vertex buffer has a very positive quality, it behaves almost constant. Our reference test system is a Pentium Quad processor machine with 8GB of RAM memory and a NVidia GeForce 8800 graphics card.
In Table 1 we can see the average results for a rendering path over 3000 frames around a set of 7 trees. The path is the same, and the leaves (up to 12K) are thrown at the same rate. The scene has 800K polygons. We can see how the falling leaves have a high impact in rendering times, although not as costly as in CPU, where, for more than 1K leaves, the rendering falls to non interactive rates.

Strategy	Average fps
No leaves in movement	209.5
Vertex shader	26.3
Fragment shader	74.7
Render to Vertex	51.5

Table 1: Average framerates through more than 3000 frames of the different strategies. Throughout the path up to 12K leaves are thrown, although not all of them are visible for the observer.

Several aspects are important to note. The vertex shader approach performs less effort than the render-to-vertex buffer method, as in the second case it requires two passes, but is still the slowest one. This is not the expected result, as in GF 8800 cards, the processors are the same for the vertex and the fragment programs, and therefore the vertex-based method should perform better. However, the render-to-vertex buffer method renders a single vertex buffer for all the leaves without any special treatment per leaf, although at the cost of modifying the position texture every time a new leaf falls. The fragment-based method is better in all graphics cards we tested, probably due to the fact that only one fragment shader is executed per leaf. We show in Table 2 the framerates for different graphics cards. As texture memory is not a bottleneck and the scene fits in main memory, the framerates mainly depend on the graphics card, not the CPU processor.

Graphics card	Units	Strategy	Avg. fps
GF 6800	22	Fragment	23.3
		R2V	12.31
GF 8600M GT	32	Fragment	22.12
		R2V	20.1
GF 8800	96	Fragment	74.7
		R2V	51.5

Table 2: Average framerates through more than 3000 frames with different graphics cards. The scene has 800K polygons. Through the path 12K leaves were thrown.

The evolution of the framerate through the path is shown in Figure 6. We show the results with the two strategies used together with a path where no leaf is thrown. In Figure 7 we show the rate of throwing leaves through the path. Moreover, in pink we can also see how many of the falling leaves are effectively rendered by the fragment-based approach, taking into account the per-leaf occlusion query-based optimization. Note that, in most of the path, roughly half of the leaves are visible. The curve falls down when leaves start to arrive to the ground, at that point, the fragment-based approach rapidly recovers its maximum framerate as can be seen in Figure 6.

7 Conclusions

7.1 Discussion

We have implemented the presented algorithm in a Quad Core PC equipped with a GeForce FX 8800 graphics card and 8Gb of RAM memory. Our results

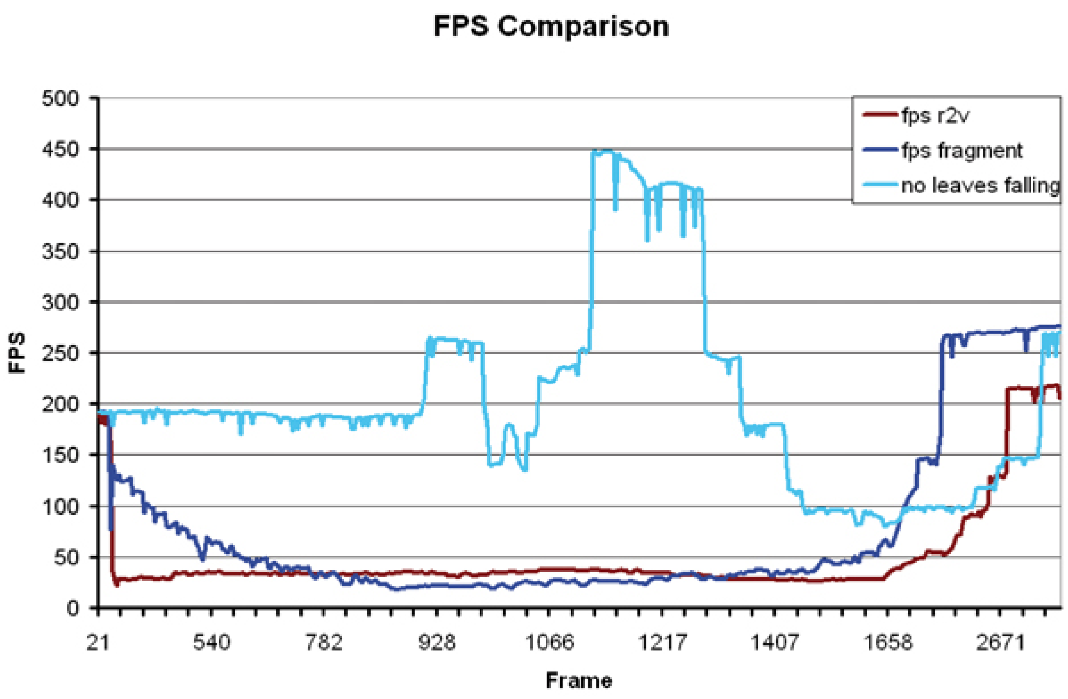

Figure 6: Framerate throughout an exploration path with the simulation carried out in the fragment shader. The scene has 1M polygons and more than 13K leaves are thrown throughout the navigation path.

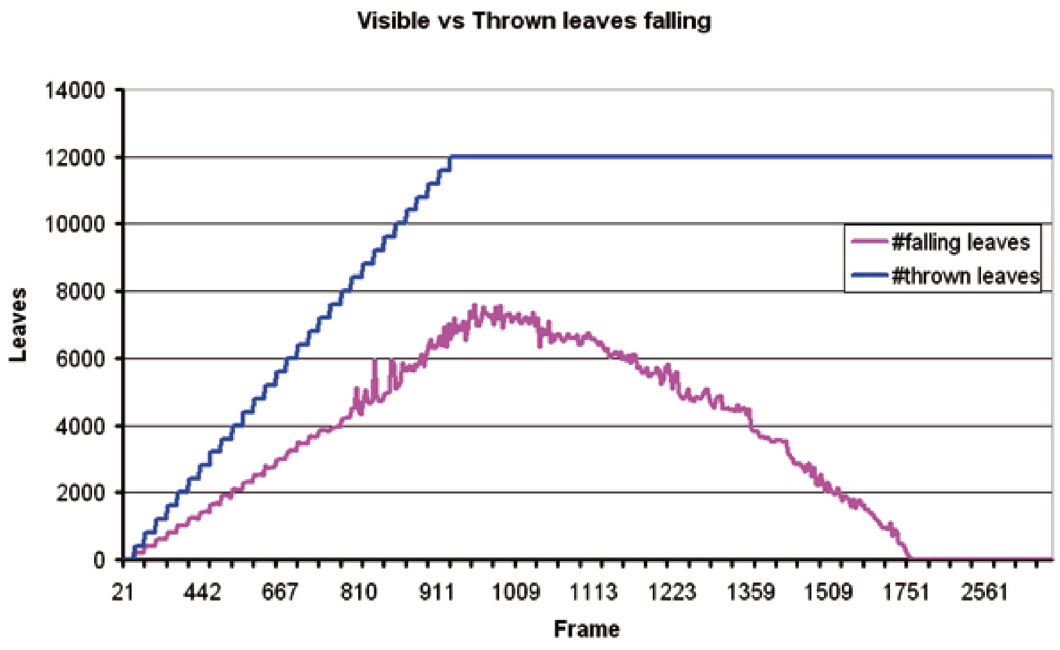

Figure 7: Number of leaves thrown throughout the path. The blue graph shows the total leaves thrown and the pink one shows the number of visible leafs at each moment when using leaf occlusion at the fragment-based approach.

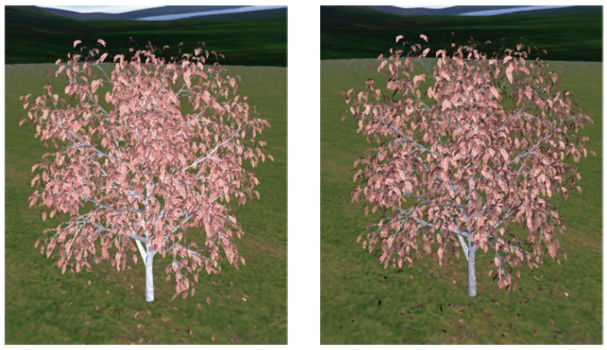

Figure 9: Two images of a tree, one without ambient occlusion (left), and the second with ambient occlusion.

show that we can render several thousands of leaves at real time with any of the strategies implemented. The method that gives the higher framerates is the fragment-based implementation, although the render-to-vertex buffer approach easily keeps a constant rate and is easier to port to older ATI cards. We have also made some tests on different graphics cards, ranging from GF 6800 to 8600 mobile and we achieve real-time in all of them. However, the performance does not necessarily scale linearly with the number of processors, for instance, we expected a more important performance boost in framerate for 8800 card than the one we finally achieved. This gives an idea of the difficulty in balancing the load with the unified architectures.

Figure 8 shows two images of complex scenes (1M and 1.2M of polygons, respectively) where thousands of leaves have been thrown at real time. Our rendering tool includes some optimizations such as view frustum culling, occlusion culling, and vertex buffer objects. Nonetheless, the renderer also implements ambient occlusion for the leaves. The effect is shown in Figure 9, although it is not so noticeable for models with darker leaves.

7.2 Future Work

Natural scenes are usually complex to model and to simulate due to the high number of polygons needed to represent the scenes and the huge complexity of the interactions involved. However, the demands of more realism in scenes do not stop growing. In this paper we have presented a method for rendering leaves falling in real time. Our system is capable of rendering thousands of leaves at rates of up to 70-90 fps. We have also developed a method for reusing a single falling path in such a way that each leaf seems to fall in a different manner. This results in low texture memory storage requirements and bandwidth. We have also presented a method for path reuse in order to make longer falling trajectories by reusing the same information data. In future we want to deal with collision detection and wind simulation.

Acknowledgments

This work has been supported by TIN2004-08065-C02-01 of Spanish Government. The authors would like to thank the comments of the anonymous reviewers which helped improved the paper.

References

[BDE04] M. Braitmaier, J. Diepstraten, and T. Ertl, *Real-Time Rendering of Seasonal Influenced Trees*, Procceedings of Theory and Practice of Computer Graphics (Paul Lever, ed.), 2004, ISBN 0-7695-2137-1, pp. 152–159.

[Bra04] Derek Bradley, *Visualizing botanical trees over four seasons*, IEEE Visualization, 2004, ISBN 0-7803-8788-0, p. 13.

[DCSD02] Oliver Deussen, Carsten Colditz, Marc Stamminger, and George Drettakis, *Interactive visualization of complex plant ecosystems*, VIS '02: Proceedings of the conference on Visualization '02 (Washington, DC, USA), 2002, ISSN 1070-2385, pp. 219–226.

[DGAJ06] B. Desbenoit, E. Galin, S. Akkouche, and J.Grosjean, *Modeling autumn sceneries*, Eurographics'06 Conference, Short Papers Proceedings (Vienna, Austria), 2006, pp. 107–110.

[DN04] Philippe Decaudin and Fabrice Neyret, *Rendering techniques*, ch. Rendering forest scenes in real-time, pp. 93–102, Springer-Verlag, 2004, ISBN 3-905673-12-6.

[FKMN97] S. B. Field, M. Klaus, M. G. Moore, and F. Nori, *Chaotic dynamics of falling discs*, Nature **388** (1997), no. 6639, 252–254, ISSN 0028-0836.

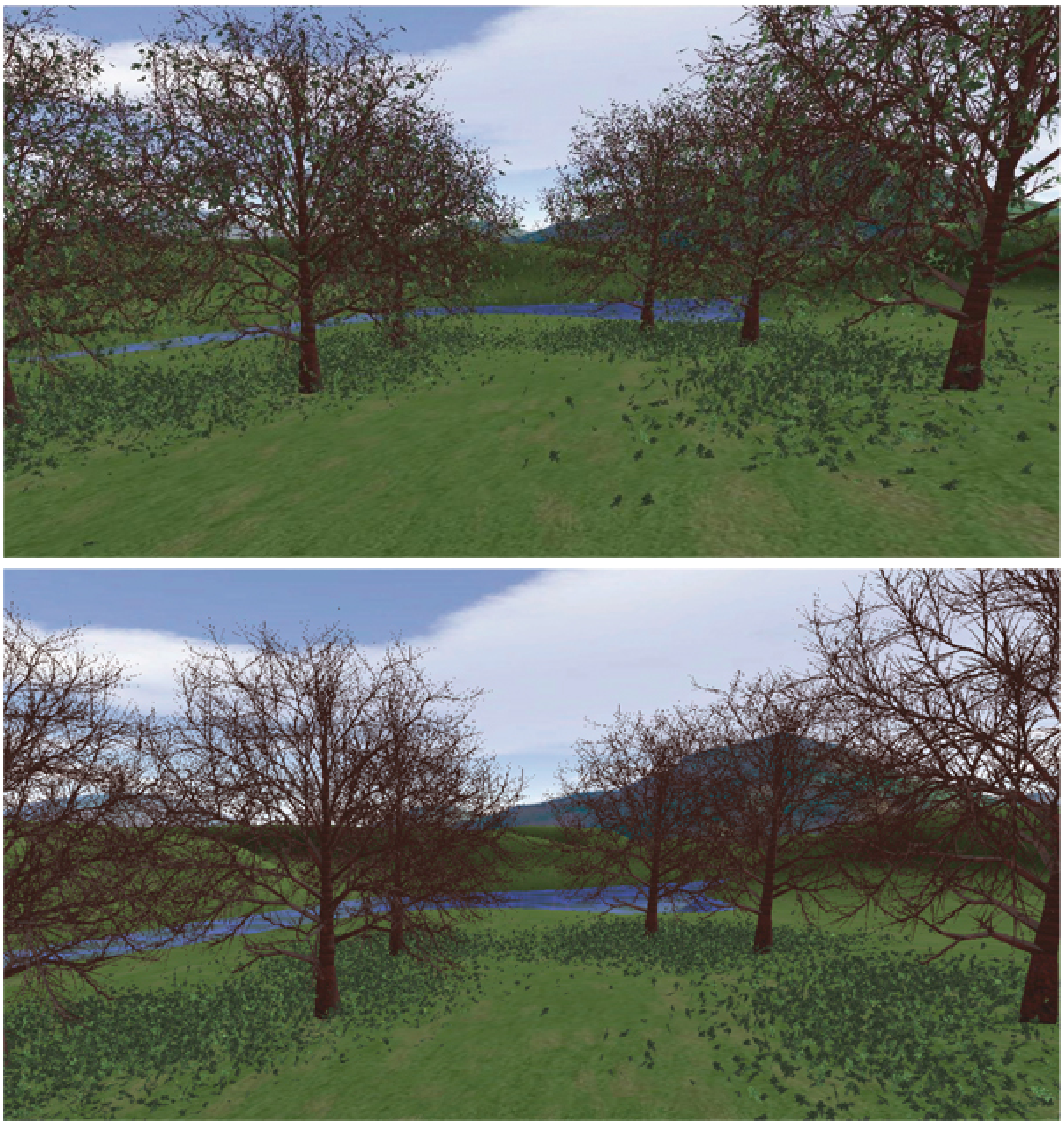

Figure 8: Two different snapshots. Top: 1M polygons, 3 thousand leaves. Down: 1.2 M polygons, 24.7K leaves.

[FO03] O. Franzke and O.Deussen, *Proceedings of the International Symposium on Plant growth Modeling, simulation, visualization and their Applications, PMA'03*, ch. Accurate graphical representation of plant leaves, Tsinghua University Press, 2003, ISBN 7-302-07140-3.

[GFG04] Philipp Gerasimov, Randima Fernando, and Simon Green, *Shader model 3.0, using vertex textures*, whitepaper, 2004, available from http://developer.nvidia.com, last visited March 13th, 2008.

[Jak00] A. Jakulin, *Interactive vegetation rendering with slicing and blending*, Proc. Eurographics 2000 (Short Presentations) (A. de Sousa and J. C. Torres, eds.), 2000.

[OTF$^+$04] Shin Ota, Machiko Tamura, Tadahiro Fujimoto, Kazunobu Muraoka, and Norishige Chiba, *A hybrid method for real-time animation of trees swaying in wind fields*, The Visual Computer **20** (2004), no. 10, 613–623, ISSN 0178-2789.

[RB85] William T. Reeves and Ricki Blau, *Approximate and probabilistic algorithms for shading and rendering structured particle systems*, Computer Graphics Proceedings (Proc. SIGGRAPH '85), vol. 19, 1985, ISSN 0097-8930, pp. 313–322.

[RGR$^+$07] C. Rebollo, J. Gumbau, O. Ripolles, M. Chover, and I. Remolar, *Fast rendering of leaves*, Computer Graphics and Imaging 2007 (Innsbruck, Austria), 2007, ISBN 0-88986-644-7.

[SF92] M. Shinya and A. Fournier, *Stochastic motion - motion under the influence of wind*, Computer Graphics Forum **11** (1992), no. 3, 119–128, ISSN 0167-7055.

[SSSE00] Arno Schödl, Richard Szeliski, David H. Salesin, and Irfan Essa, *Video textures*, Siggraph 2000, Computer Graphics Proceedings (Kurt Akeley, ed.), 2000, ISBN 1-58113-208-5, pp. 489–498.

[WH91] Jakub Wejchert and David Haumann, *Animation aerodynamics*, SIGGRAPH Comput. Graph. **25** (1991), no. 4, 19–22, ISSN 0097-8930.

[WZF$^+$03] Xiaoming Wei, Ye Zhao, Zhe Fan, Wei Li, Suzanne Yoakum-Stover, and Arie Kaufman, *Blowing in the wind*, Proc. of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation (Switzerland), 2003, ISSN 1727-5288, pp. 75–85.

Citation

Pere-Pau Vázquez and Marcos Balsa, *Rendering Falling Leaves on Graphics Hardware*, Journal of Virtual Reality and Broadcasting, 5(2008), no. 2, March 2008, urn:nbn:de:0009-6-12957, ISSN 1860-2037.