REAL TIME FALLING LEAVES

Pere-Pau Vázquez and Marcos Balsa
MOVING Group, Dep. LSI, Universitat Politècnica de Catalunya

Keywords: Real-Time Rendering, Rendering Hardware, Animation and Simulation of Natural Environments.

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, leafs) and external forces such as wind are complex to model. In this paper we concentrate in the rendering of falling leafs at low cost. We present a technique that exploits graphics hardware in order to render thousands of leafs with different falling paths at real time and low memory requirements.

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 previous 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.

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. Finally, Section 6 concludes our paper with the results and future work.

2 RELATED WORK

There is a continuous demand for increasingly realistic visual simulations of complex scenes. Dy-

Vázquez P. and Balsa M. (2007).
REAL TIME FALLING LEAVES.
In *Proceedings of the Second International Conference on Computer Graphics Theory and Applications - GM/R*, pages 244-251
DOI: 10.5220/0002082002440251

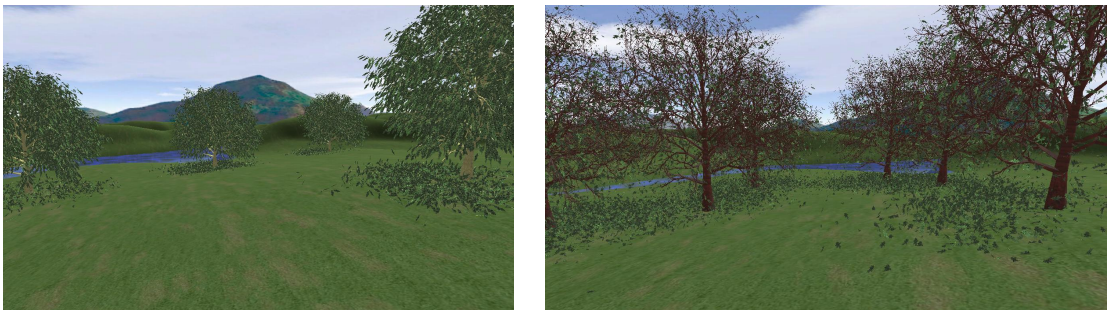

Figure 1: Two scenes with one million polygons and seven thousand (left) and 13.5 thousand leaves falling from the trees.

namic 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 it, 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 plants 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 (Bradley, 2004). 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 (Braitmaier et al., 2004) and focus especially in selecting pigments during the seasons that are coherent with biochemical reactions during the seasons. Deussen *et al.* (Deussen et al., 2002) 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 (Franzke and Deussen, 2003).

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 (Jakulin, 2000). Decaudin and Neyret render dense forests in real time avoiding the common problems of parallax artifacts (Decaudin and Neyret, 2004). 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.

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 (Wejchert and Haumann, 1991) 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.

Ota *et al.* (Ota et al., 2004) 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 (Reeves and Blau, 1985) proposed an approach based on a particle modeling system that made the particles evolve in a 2D space with the effect 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.* (Wei et al., 2003) 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.

3 RENDERING LEAVES

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 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 at real time, and thus, we get a differently looking path per leaf.

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 in a vertex shader. We have used the so-called Vertex Texture Fetch extension (Gerasimov et al., 2004), 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. Moreover, the vertex shader takes care of the path modification too.

Initially, a simple path is calculated and stored in a texture. We could use more than one path, and reduce the computations in the vertex shader, although this results in small gain because the current GPU implementations (before GF 8800) of vertex texture fetch are quite slow and the cost of texture accesses dominate over the rest of the work of the vertex. For each leaf, the vertex shader updates its 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 naturally looking fashion. Next, we explain the path construction process, the contents of the texture path, and overview the contents of our vertex shader. Section 4 deals on reusing data to produce different trajectories for each leaf.

3.1 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 (Wejchert and Haumann, 1991) or Wei *et al.* (Wei et al., 2003), have been developed. 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. 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.

3.2 Path Information

As we have already explained, we start with the precomputed data of a falling leaf and provide it to the vertex 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 easy the encoding, 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 vertex 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 vertex shader also receives other information such as the vertex position, its normal, and so on. Each frame, the vertex 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 vertex 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.

3.3 Overview

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

1. Looks for the initial frame f_i (different per leaf)
2. Seek current position in path (f_i+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.

3.4 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 vertex 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 vertex 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

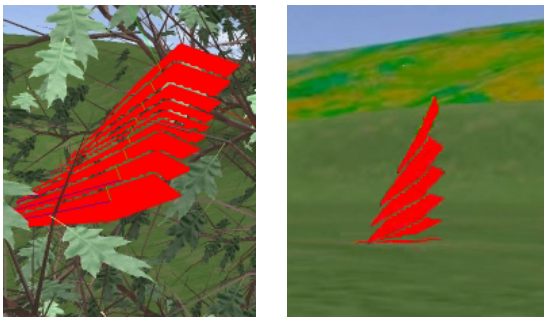

Figure 2: Several superposed images of a marked leaf starting to fall (left) and adapting to a planar position on the ground (right).

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 vertex 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).

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).

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 vertex 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 that is computed as follows: $permAngle = mod(center\_obj.x * center\_obj.y * center\_obj.z * 17., 360.)$. The product has been chosen empirically.

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: $scale\_x = mod(center\_obj.z * center\_obj.y * 3.0, 3.)$

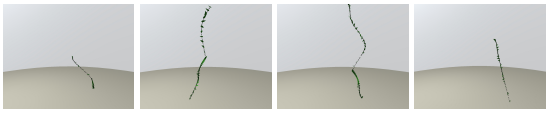

Figure 3: Different variations of the initial path.

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

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

and $scale\_z = mod(center\_obj.x * center\_obj.y * 3.0, 3.)$ respectively. This results in a non uniform distribution of the leaves at a *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 (Schödl et al., 2000). 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 must be similar in order to preserve continuous animation.

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 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 vertex 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: $mod((centre_obj.y * centre_obj.x * 37.), float(MAX\_F - 1)$ where *MAX_F* is the maximum number of frames of the animation. These continuity points are chosen among the ones with slow velocity and orientation roughly parallel to the ground, in order to ensure the soft starting 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 display 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 cost incurred in the vertex shaders. As all thrown leaves are processed by the vertex shader even if they already are on the floor. The vertex processing cost increases because the accesses to texture are determined by the current frame. When the computed position is *under* the floor, then the vertex 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.

We have implemented a simple solution that consists in removing the leaves that are on the ground from the list of leaves that are treated by the vertex 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). The correct position on the ground is computed on the CPU and updated, and the leaf is removed from the list of falling leaves. This step is not costly, as our experiments show that for one thousand falling leaves, 5 to 13 leaves need to be erased at each frame. This allows us to keep the frame rate constant for long exploration paths and thousands of falling leaves.

We have further taken advantage of graphics hardware in a different way: we use occlusion queries to lazily remove leaves from the pipeline for a small set of frames, for instance 4 frames which causes no noticeable artifacts. We proceed the following way: At each frame, the potentially visible leaves are thrown and use an *occlusion query* per leaf to determine if they were finally visible. In case they were not, these are removed from the visible list for 4 frames, and then rendered again with the occlusion query. As we are going to skip up to 4 frames, we only check for a subset of the leaves each of the frames, which allows us to balance the cost of occlusion queries throughout the frames. This improvement has noticeable results especially when the occlusions are important, as when the walthrough goes among the trees, or the trees are densely populated. Note that we cannot use view frustrum culling for leaves as their initial position is not the same than the one they will be rendered to.

The impact of those modifications can be seen in Figure 6 where the framerates of a complex navigation path is shown. We compare the initial method without optimizations (green) with the first optimization (blue), that consists in progressively determine the leaves which are on the floor and compute their

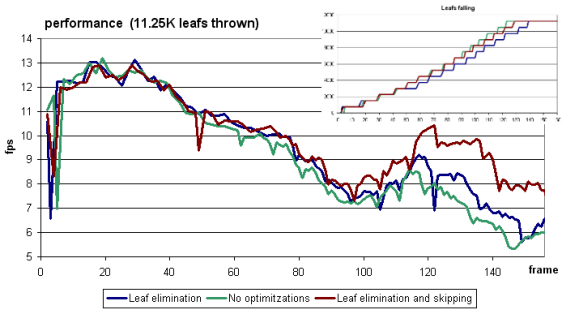

Figure 6: Results with a complex path in a scene of 1M polygons. Top right shows the amount of leaves that have been thrown, up to 11250, throughout the path. Note the maintenance of framerate for the optimized (brown) version of our method.

actual position and erase them from the set of leaves that are rendered using the vertex shader, and the one that also includes occlusion queries for selectively skip non visible leaves for up to 4 frames (brown). The framerate especially decays with the increase of falling leaves if we do not wipe out from the pipeline when they already are on the ground. Note how the optimized version (brown) maintains the framerate for a large number of falling leaves, and can even double the results of the non optimized version.

6 CONCLUSIONS

6.1 Results

We have implemented the presented algorithm in a 3.4GHz PC equipped with a GeForce FX 6800 ultra graphics card and 1Gb of RAM memory. Our results show that we can render several thousands of leaves at real time. We have experienced with different configurations, some results are shown in Table 1. In order to determine the cost of the rendering process, we have to compare with the walthrough that throws no leaves. Note that the framerate impact of falling leaves is influenced not only by the number of falling leaves but the number of polygons that the leaves have. We can see that even for large scenes (768K polygons) and a high number of leaves thrown, we maintain the framerate with small (around 20% for 5000 leaves) cost penalization.

6.2 Conclusions and 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

Table 1: Results obtained with our algorithm for different amounts of falling leaves and trees. Average fps are obtained from different walthroughs of several hundred frames that surround or cross the set of trees. All examples are rendered at full screen (1280×1024) resolution.

Trees/Pols	Triangles per leaf	Leafs	fps
1/128K	6	0	36.97
1/128K	6	1000	34.4
1/128K	6	2000	31.88
1/128K	6	5000	28.09
4/512K	6	0	18.79
4/512K	6	1000	17.49
4/512K	6	2000	16.66
4/512K	6	5000	14.27
6/768K	6	0	10.9
6/768K	6	1500	10.04
6/768K	6	3000	9.56
6/768K	6	6000	8.63

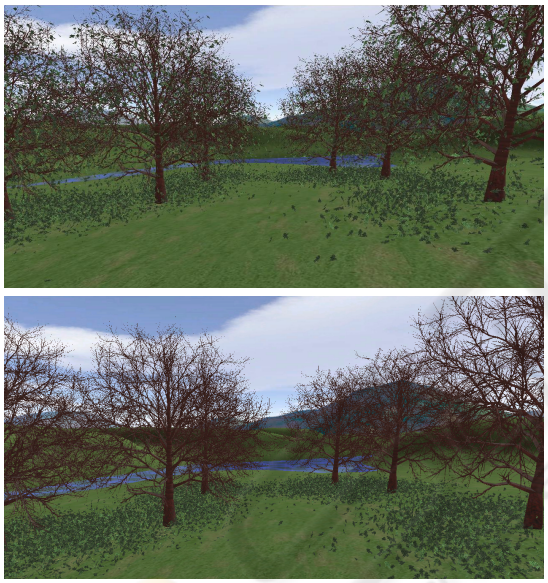

Figure 7: Two different snapshots. Top: 1M polygons, 3 thousand leaves. Down: 1.2 M polygons, 24.7 leaves.

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 20-30 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.

ACKNOWLEDGEMENTS

This work has been supported by TIN2004-08065-C02-01 of Spanish Government.

REFERENCES

Bradley, D. (2004). Visualizing botanical trees over four seasons. In *IEEE Visualization*, page 13.

Braitmaier, M., Diepstraten, J., and Ertl, T. (2004). Real-Time Rendering of Seasonal Influenced Trees. In Lever, P., editor, *Procceedings of Theory and Practice of Computer Graphics*, pages 152–159. Eurgraphics, UK.

Decaudin, P. and Neyret, F. (2004). Rendering forest scenes in real-time. In Jensen, H. W. and Keller, A., editors, *Rendering Techniques*, pages 93–102. Springer-Verlag.

Deussen, O., Colditz, C., Stamminger, M., and Drettakis, G. (2002). Interactive visualization of complex plant ecosystems. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 219–226, Washington, DC, USA. IEEE Computer Society.

Franzke, O. and Deussen, O. (2003). *In Plant Modelling and Applications*, chapter Accurate graphical representation of plant leaves. Springer-Verlag.

Gerasimov, P., Fernando, R., and Green, S. (2004). Shader model 3.0, using vertex textures, whitepaper. http://developer.nvidia.com.

Jakulin, A. (2000). Interactive vegetation rendering with slicing and blending. In de Sousa, A. and Torres, J., editors, *Proc. Eurographics 2000 (Short Presentations)*. Eurographics.

Ota, S., Tamura, M., Fujimoto, T., Muraoka, K., and Chiba, N. (2004). A hybrid method for real-time animation of trees swaying in wind fields. *The Visual Computer*, 20(10):613–623.

Reeves, W. T. and Blau, R. (1985). Approximate and probabilistic algorithms for shading and rendering structured particle systems. In Barsky, B. A., editor, *Computer Graphics Proceedings (Proc. SIGGRAPH '85)*, volume 19, pages 313–322.

Schödl, A., Szeliski, R., Salesin, D. H., and Essa, I. (2000). Video textures. In Akeley, K., editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 489–498. ACM Press / ACM SIGGRAPH / Addison Wesley Longman.

Wei, X., Zhao, Y., Fan, Z., Li, W., Yoakum-Stover, S., and Kaufman, A. (2003). Blowing in the wind. In *Proc. of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 75–85, Switzerland. Eurographics Association.

Wejchert, J. and Haumann, D. (1991). Animation aerodynamics. *SIGGRAPH Comput. Graph.*, 25(4):19–22.