Hardware Accelerated Rendering of Foliage for Real-time Applications

Gábor Szijártó
József Koloszár

Budapest University of Technology and Economics

Abstract

One of the major challenges in developing techniques for realistic and high performance visualization of outdoor environments is rendering of vegetation. The greatest problem is that convincing modeling of trees, bushes and undergrowth requires very large numbers of polygons that exceed the limits posed by rendering hardware today (and in the near future). A number of methods have been proposed in the past to address the issue, most of which are variants of multi-resolution modeling and level-of-detail algorithms. This paper reviews some of the landmark techniques used in real-time PC applications (mostly games and flight-simulators), and presents a solution that takes advantage of the programmable rendering pipelines available on most of the recent video cards. The algorithm uses view dependent 2.5 dimensional impostors to visualize trees in convincing quality for most levels of detail. Numeric results are presented to illustrate rendering efficiency, and specific, implementation related issues are also discussed.

Keywords: Tree Rendering, Outdoor Simulation, View Dependent Visualization

1 Introduction

Rendering some form of vegetation is a must, when it comes to outdoor scenes in virtual reality environments (unless one is modeling arctic ice fields or deserts). The omission of trees and bushes drastically reduces the feeling of realism, which is one of the key factors in immersing the user in the virtual experience.

The computer and video game industry has become the driving force behind the most recent advances in rendering real-time virtual environments. User demand for more visually intensive, better looking, more immersing software products, and PC systems required to run these applications, has resulted in the availability of very powerful graphics hardware for the mass market.

Mainstream video cards today boost more power than graphics workstations considered high-end just a few years ago. The performance of Graphics Processing Units (*GPU*s) seems to be increasing even faster than Moore’s Law.

For a long time consumer hardware acceleration of video cards has been geared to high throughput texturing, as most games and related applications relied (and to this day are still relying) on sometimes multiple layers of high quality textures for realistic visuals. However, the iterative rendering architecture realized in hardware has been very inflexible, thus limiting the range of applications able to take advantage of the underlying processing power. With the introduction NVIDIA’s *GeForce* series of cards, the concept of programmable rendering pipelines has been introduced. Though there are still limitations, the most recent boards from NVIDIA (the GeForce4 and FX series) and ATI (*Radeon* 9x00) [1] finally offer enough flexibility to make even demanding visualization problems feasible. The shear processing power that can be controlled by programming the geometry and texturing stages (vertex- and pixel-programs, respectively) has opened new frontiers from volume visualization to mass rendering.

The two leading hardware manufacturers mentioned above both offer fairly solid support for OpenGL and DirectX. For the research involved in this paper, Microsoft DirectX 9.0 [23] has been the platform of choice. DirectX supports accelerated hardware through various versions of pixel and vertex program profiles. Programs or *shaders* are written in an assembly-like language. NVIDIA introduced its C for Graphics (*Cg*) [13] programming language to provide developers with easy to use C-like syntax to develop advanced shaders. The Cg toolkit available free of charge from NVIDIA’s homepage provides tools to compile Cg code to the DirectX shader language, or the NVIDIA specific register-combiner extensions for OpenGL. For this research vertex and pixel programs were developed in Cg, compiled to DirectX shaders, which were fine-tuned by hand when modifications were deemed necessary.

With the abovementioned advances in graphics hardware programmability, real-time tree rendering can also be taken to a new level. The algorithm presented in this paper does not deal with geometry issues, and it has not been designed for high-fidelity vegetation rendering. It is a method for rendering realistic-looking scenes at frame rates appropriate for game and flight simulation engines.

2 Vegetation Rendering

2.1. Simulator and Game Applications

It is often useful to define specific scales of simulation at which a vegetation-rendering algorithm should provide the required level of realism. Most applications can be assigned to one or more of the following categories:

- **Insect scale**: The level of simulation where a consistent, realistic depiction of individual branches and leaves is expected. (The avatar can climb the tree.)
- **Human scale**: Scenes must look realistic through distances ranging from an arm's reach to some tens of meters. Consistency is desired but not required. (The avatar can bump into trees, even dash through bushes, but does not focus on specific details.)
- **Vehicle scale**: At this level vegetation serves as little more than a backdrop. Individual trees are almost never focused upon and consistency is not required. Viewing distance may exceed several hundred meters. (The avatar is usually moving through the environment at some altitude at faster than running speeds.)

Flight simulation applications probably have the longest history in outdoor rendering. Most flight simulators that do any vegetation rendering at all do so at vehicle scale. The problem is usually addressed by using one or a few (alpha-blended) textures to depict a forest cover from higher altitudes. While a dense forest canopy can be rendered quite convincingly using these methods, low-altitude simulation of sparse vegetation is yet a challenge unanswered. Some products warp randomly generated individual trees onto the landscape when the avatar descends to lower altitudes, but these are usually little more than billboards, not looking very realistic at all.

Image 1: *Textured tree models in EPIC's Unreal Tournament 2003 first person shooter.*

The first moderate successes in realistic outdoor simulation at human scale emerged only a few years ago. Though the proud history of first-person shooter – probably the most popular style of computer game today – now spans more than a decade, outdoor environments with lush vegetations have either been lacking, or have made a rather poor impression. Arguably some of the best results where the earliest: Novalogic's voxel based engine, which has since been abandoned in favor of accelerated iterative rendering, being an example worth mentioning. Most offerings usually rely on static, multi-layered aligned billboards. Though in some rare cases these algorithms yield satisfactory illusions, they are far from what can be potentially achieved with recent hardware.

Insects scale simulation has yet to make its debut in the gaming and simulation industry.

The focus of this research has been an algorithm for human and vehicle scale simulation, with possible application in low-altitude (helicopter and glider) flight, land vehicle simulators, and first person shooters.

2.2. Related Work and Research

Vegetation rendering has also enjoyed its share of scientific interest. The two broad fields associated with the topic are the generation of plants, and their visualization. The former – vegetation modeling – lies outside the scope of this study. Extensive research in the field has yielded a number of published results, and commercial tools for modeling are also available [16]. Only one aspect of modeling is stressed in the context of this paper: a good tree-rendering algorithm has to able to introduce variations of tree characteristics. One tree is never enough, and a forest consisting of identical trees is almost as unrealistic as one with no trees at all!

Visualization seems to be a nut harder to crack. There are two general approaches: geometry-, and image-based methods. As its name suggests, techniques of the former group use geometric representations of the foliage. As it takes roughly hundred thousand triangles to build a convincing model of a single tree, some form of Level of Detail (*LoD*) rendering technique must be applied to reduce the polygon count for a given frame to a reasonable level [17, 4]. Visually pleasing results can usually only be achieved with rather complex algorithms, or significant memory overheads. The authors suggest that for purposes of games and simulation, geometry based methods are not yet efficient enough. Image-based methods represent a trade-off of consistency and physical precision in favor of more photo realistic visuals.

A brief note on terminology: There is some inconsistency in literature regarding the exact definitions and usage of the image-based rendering related terms: *sprite, billboard* and *impostor*. In this paper the term sprite refers to a flat face with some static surface properties (usually a simple texture image) always facing the camera from a fixed position in space (Figure 1). A billboard is face that rotates around one fixed axis in space, trying to face that camera as much as possible (Figure 2). The term impostor – used mainly in game programming jargon – refers to a billboard or sprite, whose surface properties are not static (the texture used is also rendered to).

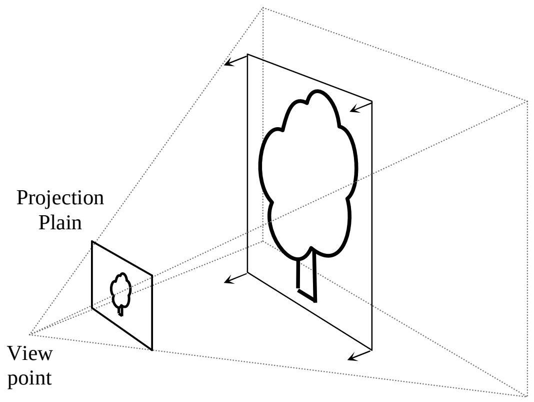


Figure 1: *Sprite Rendering. The textured polygon is always facing the camera.*

The most basic of all image based methods is sprite rendering. This technique, which, in the context of tree-rendering is analogous to using cardboard cutouts with a tree-like image painted on them, which are always facing the camera, has been around in computer games since the introduction of texturing. Though resulting visuals are far from satisfactory, the technique is used to this day to depict smaller plants.

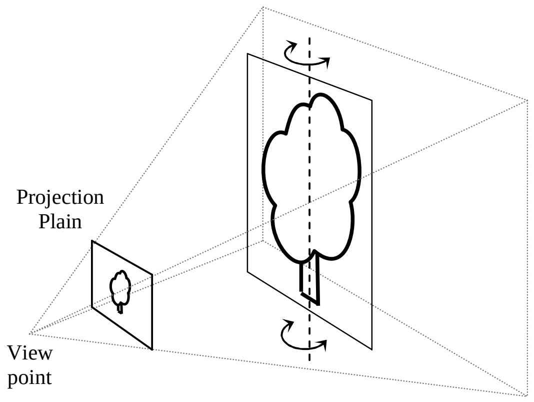


Figure 2: *Billboard with arbitrary position and orientation.*

Two obvious improvements introducing some form of view-dependence to the above method were soon to follow: sets of view-dependent sprites, and complex cutouts. The former method simply pre-renders a finite set (usually 4 or 8) of views of the same tree, and presents the one closest in alignment with the actual viewing direction. A popping artifact is visible when there is an alignment change. The latter algorithm uses texture transparency and blending to render more than one view at the same time onto properly aligned surfaces (Figure 3). Both methods yield surprisingly acceptable results in vehicle scale simulations, but fail to deliver quality in close-up views. It is also not trivial to introduce varying shapes and sizes of trees without overtaxing memory.

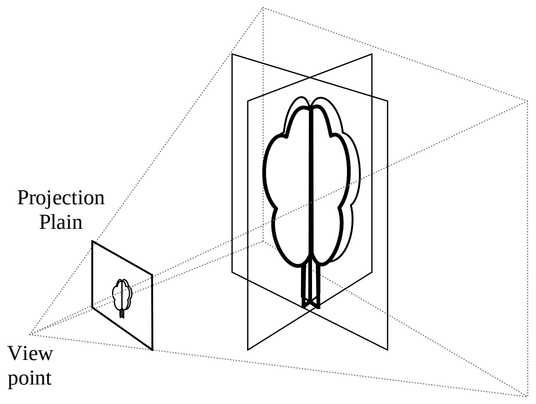


Figure 3: *Complex cutout with two faces.*

One of the most advanced methods actually implemented in commercial entertainment software is the basic free-form textured tree model (Image 1) with some LoD applied. Though the idea is quite straightforward, only in the last few years has hardware become powerful enough to handle the task. Resulting visuals are satisfactory, although due to the simple geometric model used, close-up views usually look artificial, and variations are usually introduced through new models (or through combining model-parts). Increasing the number of trees in a scene quickly bogs performance. Most recent human scale simulators rely on this technique, and the raw hardware power to cope with it.

Though more advanced techniques using billboarding [12], layered depth-images [19], and multi z-buffers [12] have been presented through scientific forums, most are computationally too expensive for implementation in games or simulators. It is important to stress that even though the quality of tree- and foliage rendering makes a huge impact on overall visuals, in an actual product it is but a small part of a complex rendering engine, and must share the available hardware resources with other computation-intensive tasks.

3 Introducing 2.5 Dimensional Impostors

3.1. The Algorithm

The technique proposed is an improvement to traditional impostor rendering. Impostor rendering has two stages. The first stage is a view-dependent render-to-texture operation (drawing the impostor), the result of which is used in the second stage usually as sprite or billboard texture. The technique has originally been introduced to overcome limitations of Z-buffer precision: complex object can be rendered to impostors, maintaining a correct depth precision locally, and great number of impostors can be rendered coherently in the second stage with global depth precision.

The most trivial application in tree rendering would be to render a correctly aligned single tree into the impostor, and then use sprites to render a forest in the second stage. However, the issue of the lack of variety has to be addressed.

The idea is to compose the final image of the foliage of more than just a single sprite, as shown on Figure 4. A cloud like image of leaves is rendered to an impostor, and applied to a number of properly positioned sprites to build the whole canopy. Thus sufficient variations can be introduced by perturbation of the relative positions of the sprites in space. Rendering two or three different textures for sprites can introduce even more diversity. Individual sprites can also be blended using different colors, again to introduce variations, and even to fake lighting. A canopy depicted using 10-100 sprites can look very convincing in still images, even if the same texture is repeated over all sprites. If textures are pre-computed or hand drawn, high-pass filtering should always be applied to eliminate low-frequency components from the texture, as those make repetitions apparent when the texture is tiled (or rendered to sprites over and over).

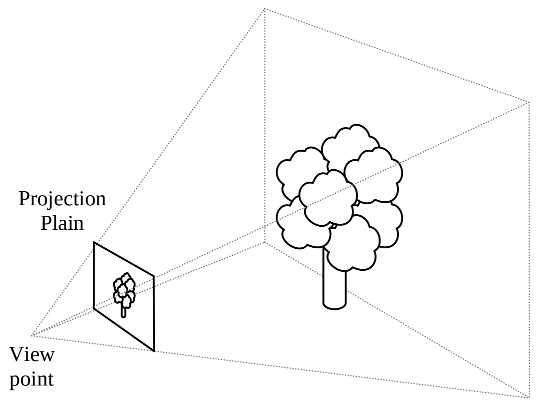


Figure 4: *Using multiple sprites to render the tree.*

However, the view-independence of sprites makes the above method quite useless for real-time rendering when the avatar is moving. As sprites are always turning to face the camera, if their appearance is constant, they upset the motion-parallax producing very unrealistic results. The introduction of view-dependence through the use of impostors updated every frame instead of static sprites successfully eliminates this problem.

An arbitrary group of leaves arranged randomly in space and assumed to be positioned in the center of mass of the canopy is rendered to a texture with correct alignment (Figure 5). The texture is then used as an impostor and applied to more sprites to build the canopy. Image quality and feeling of realism increase dramatically, even if the same impostor is used for the entire tree (Image 2). The fact that impostors are flat results in some disturbing artifacts, for example when branches rendered from geometry stab through the sprites.

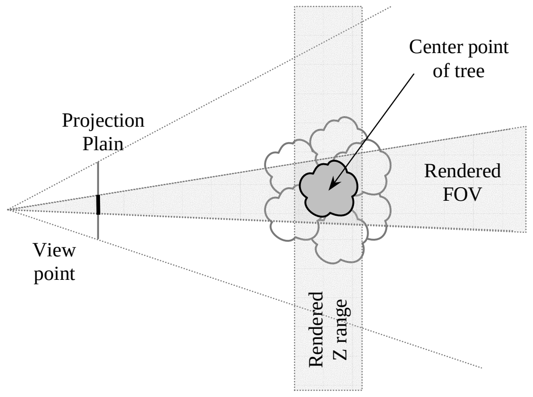


Figure 5: *Rendering the impostor texture from a group of leaves assumed to be positioned at the center point of the tree.*

To correct the effect, depth is introduced to the render-to-texture procedure. In the stage of impostor rendering, depth information is stored in the target textures alpha channel. The result is a 2.5 dimensional impostor. In the implementation the group of leaves rendered is assumed to be at some center point of the canopy, just as before, but the z-near and z-far planes are adjusted to approximate a reasonable bounding box for the rendered group of leaves, as shown on Figure 5. In the final phase of rendering the stored depth values are appropriately scaled and clipped to the final depth buffer before depth testing is performed, yielding a volumetric feel to the textured sprites, which can now inter-lap in a spatially coherent manner.

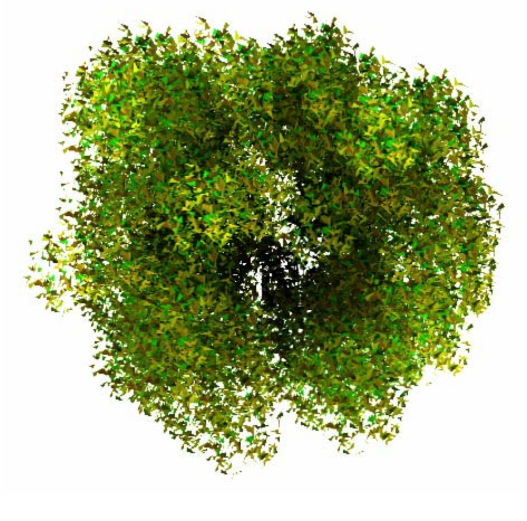


Image 2: *Tree canopy rendered using impostors.*

The artificial look resulting from repetition of the same image over many sprites is almost completely eliminated, as volumetric overlapping obscures arrangement to the point where it is almost impossible to discern any single impostor.

3.2. Implementation

The technique described above produces very convincing visuals, and can be efficiently implemented using only the GPU on recent video cards (Figure 6).
The ultimate version of the algorithm requires hardware support for the following features:

- Programmable pixel shaders.
- Render-to-texture operations.
- Limited flow control in pixel shaders.
- Explicit writes to the depth buffer from pixel shaders.

Note that the latter two features are only supported through DirectX9 Version 2.0 Pixel Shaders or above (*ps_2_0+*). Currently ATI's R3xx range of boards (Radeon 9700, 9500 and Pro variants) support ps_2_0 [2], and the GeForceFX is NVIDIA's only offering supporting ps_2_ext (NVIDIA has no boards supporting ps_2_0, the GeForce4's capabilities are limited to ps_1_3).

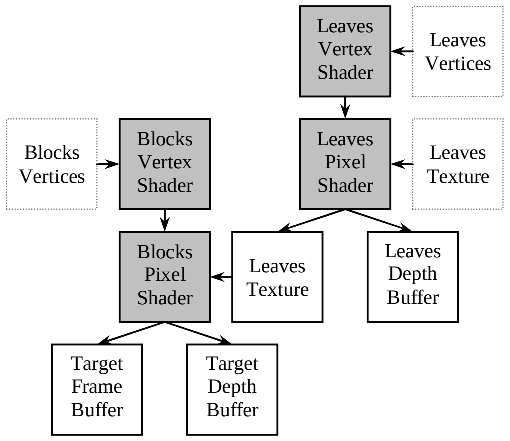


Figure 6: *Block diagram for implementation using two vertex, and two vertex programs. Depth information is stored in the alpha channel of Leaves Texture.*

A trimmed version of the algorithm without depth information in the impostors can be implemented on earlier hardware. When doing so, however, sprites have to be sorted back-to-front for rendering, introducing more off-board calculations. Resulting visuals are still very appealing.

3.3. Enhancements

A number of features can be added to the above algorithm with minimal or no impact on performance. The first three enhancements below are proposed to increase variations. The remaining ones aim to improve realism. Some of these techniques have already been mentioned in preceding chapters.

- **Tanning**: specific impostor textures can be blended with individual constant colors in the final rendering stage using pixel shader. For example, a slight brown shade could be applied to originally green leaves. Doing so requires an additional addition instruction in the pixel program.
- **Stretching**: some per-tree randomness may be introduced to slightly alter the size of sprites, at the added cost of one or two instructions in the vertex program.
- **More impostors**: as the name suggests, increasing the number of different impostor textures applied is probably the most straightforward way to introduce variations.
- **Textured leaves**: though great visuals can be achieved by just using flat-shaded triangles to represent leaves rendered to the impostor, the use of some leaf-like texture can greatly improve quality. Adding this feature is recommended in case there is need for detailed close-ups, which might be the case in human-scaled simulators.
- **Lighting**: fake lighting can be introduced, and impostors on the sunny side of the tree rendered in brighter shade than those on the lower or central part of the canopy. This feature requires about two additional instructions in the vertex program. Note, that some lighting can also be applied when rendering leaves to the impostor.
- **Ground shadows**: the impostor texture used in the canopy can also be projected to the ground using a different pixel shader in the final rendering stage. The addition of this feature requires revision of the main rendering loop in the application.
- **Dynamics**: the vertex program can be modified to introduce wave-like motion, simulating wind blowing through the canopies.

3.4. Performance Benefits

The technique presented in the preceding subsection has several advantages over others.

- It is a very simple approach, and can be implemented using the graphics hardware.
- Vegetation geometry can be kept at a reasonably simple level. As opposed to algorithms using LoD geometry, complex geometric representations for trees need not be maintained at all.
- The algorithm is fill-limited, meaning that performance is a function of the number of pixels filled. Trees in the distance only cover a small number of pixels, thus the cost of rendering them is minimal. Algorithmic decimation of model complexity is only required when frame rates are to be boosted beyond the edge.

3.5. Drawbacks and Limits

As mentioned in the introduction, the algorithm presented is geared to render convincing vegetation at very high speeds, not to render high-fidelity models of trees.

Though the technique can be applied as presented in any vehicle- and most human scaled simulators, extreme close-ups may look a little artificial, and further tuning may be necessary. Also, when the canopies are traversed, some artifacts may appear due to z-buffer related issues.

3.6. Rendering Forests

Though forests of only a couple of trees may be visualized by rendering individual trees in a loop, once the number of trees in a scene increases beyond fifty, modifications are required to maintain real-time frame rates. The reason is that switching render targets is rather costly even with hardware support for render-to-texture operations. As the abovementioned brute-force approach would render one impostor for every tree in the forest, the render target would be switched at least 201 times for scene with 200 trees.

To resolve the issue only a fixed number of different views for impostor textures are rendered. Screen space is partitioned using a regular grid of some size (5 by 5 is a reasonable choice). Leaves are rendered into impostors with alignments and orientations characteristic to individual screen sectors. In the final rendering stage all sprites in a sector share the appropriate impostor image. Note, that all enhancements mentioned in Section 3.3 are still applied on a per sprite basis, maintaining the illusion of many variations, even though a number of trees may be sharing the same impostor texture. Also, all impostors can be tiled on a larger target texture, and the appropriate image for a given sprite selected in the vertex program by specifying explicit texture coordinates. This reduces render-target switches to two, regardless of the number of trees in the forest.

Also note that the technique above improves image quality when looking at a single tree from a very close distance. Normally, only a single impostor texture would be rendered for a single tree. This would become disturbing in close-up views, because of the lack of perspective distortion. With a 5x5 grid, the above modification would render a tree filling the entire screen using 25 different textures.

4 Results

Tests were performed on a 1.2GHz Athlon based PC equipped with a GeForce4 graphics accelerator. Vertex and pixel programs were written in Cg and compiled to DirectX 9 shaders. As this hardware does not support some required pixel shader operations, only a trimmed version of the algorithm was extensively tested. Advanced features were tested for feasibility using software reference drivers, and some brief tests were also run on a Radeon 9700, but due to technical difficulties, extensive evaluation of performance was not possible.

Image 3 depicts impostor textures rendered from increasing number of leaves. Leaves are modeled by simple triangles arranged randomly. The basic version of the algorithm computes one impostor per tree. A single impostor texture can be created and rendered in very little time. Increasing the number of leaves even to the very dense level of 512 does not significantly impact performance. The time spent creating impostors is negligible compared to the other stages of rendering.

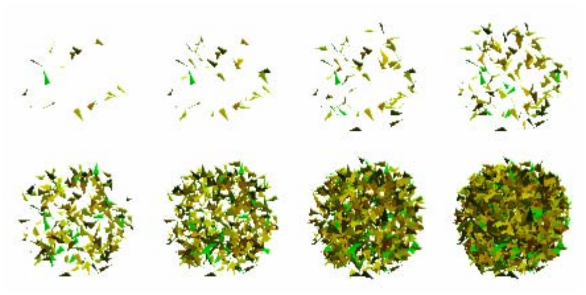


Image 3: *Different impostor textures rendered from 16, 32, 64, 128, 256, 512, 1024 and 2048 leaves.*

Image 4 shows a typical canopy for a single tree, with 128 leaves in the impostor texture, and 128 impostors for the tree. These numbers are ideal for simulating a fair number of trees at human-scale detail. Note that the visual quality of the image is almost equivalent to rendering a tree model with 16384 leaves defined in geometry. Rendering speeds recorded were around 150 frames per second (fps). Impostor texture size for all tests was fixed at 128x128.

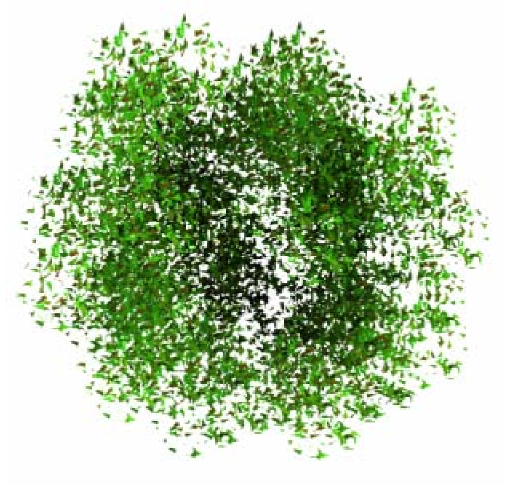
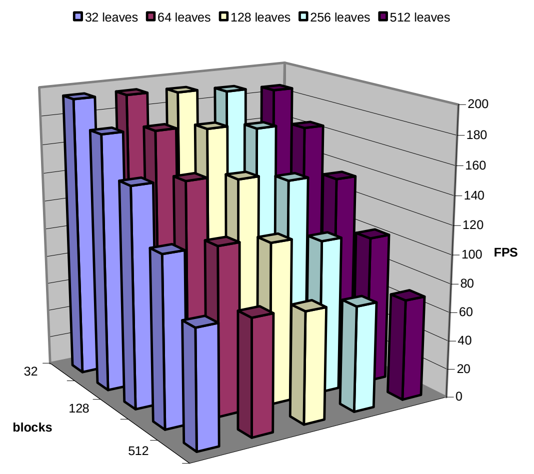


Image 4: *Canopy of a single tree (128 leaves per block, 128 blocks).*

Image 5 is a rendering of a forest canopy viewed from low altitude. Only brute-force forest rendering was tested: all trees were rendered from scratch individually. Though this method is far from optimal, it gives valuable insight into factors limiting performance. Please refer to Section 3.6 for details. Even brute-force forest rendering could achieve very pleasing visuals suited for vehicle-scale detail at up to 50 fps on the test system.

Image 5: *Forest canopy (approx. 200 trees).*

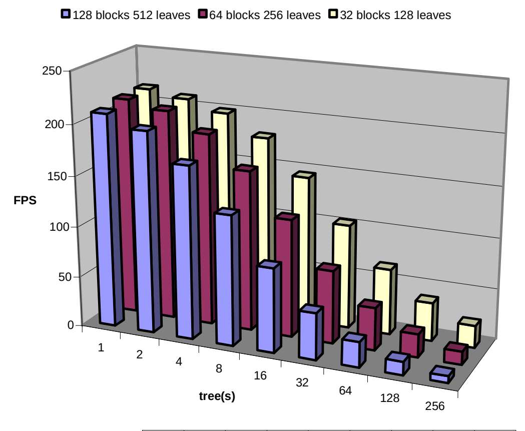
Table 1 below summarizes raw performance data from rendering a single tree filling most of the 768x768 frame buffer. Notice that increasing the number of leaves per impostor does not seriously impact performance.



	32	64	128	256	512
🟦 32 leaves	195	178	152	117	81
🟥 64 leaves	194	176	150	116	80
🟨 128 leaves	192	173	146	112	77
⬜ 256 leaves	189	169	140	107	73
🟪 512 leaves	186	165	136	103	70

Table 1: *Frame rates for rendering a single tree with increasing number of leaves/impostor (columns) and increasing number of impostors/tree (rows).*

Table 2 presents results from rendering forests of different size. As already discussed in Section 3, results indicate that performance is primarily fill-sensitive, and in case of the tested brute-force approach, limited by switching the render target (and rendering the impostor over) for every tree in the forest.



	1	2	4	8	16	32	64	128	256
🟦 128 blocks 512 leaves	210	198	170	128	83	46	25	13,1	6,7
🟥 64 blocks 256 leaves	214	207	189	158	116	72	42	23	13
🟨 32 blocks 128 leaves	215	209	199	179	145	103	65	38	22

Table 1: *Frame rates for forest rendering.*

5 Conclusions and Future Work

It must be noted that it is not possible to illustrate the most important visual property of the algorithm with still images. The power of the technique becomes obvious when set into motion: there are no popping artifacts typical to LoD based techniques, no sprites turning to face the camera, etc. On the other hand there is the illusion of a great deal of detail being depicted, as the number of leaves perceived can easily surpass one million for an average forest canopy.

The presented algorithm can be used in vehicle and human scale simulators to render realistic looking trees and forests in real-time. The technique takes full advantage of the programmable rendering pipeline available on recent graphics accelerators.

Though only discussed in the context of tree rendering, the method can easily be adopted to render bushes and undergrowth, even grass.

After extensive evaluation of implementations of the algorithm featuring all enhancements discussed in the paper, future work will focus on efficient handling and generation of tree trunks and branches and methods for rendering grass, ferns, pines and irregular tree shapes using modifications to the presented 2.5 dimensional impostor method.

6 Acknowledgements

This work has been supported by the National Scientific Research Fund (OTKA ref. No.: 42735), by IKTA-00159/2002, by the Sloven-Hungarian Action Fund and Intel Corp.

References

[1] ATI Technologies Inc., RADEON™ 9700 Pipeline Overview, 2002, www.ati.com

[2] ATI Technologies Inc., Performance Optimization Techniques for ATI Graphics Hardware with DirectX® 9.0, 2002, www.ati.com

[3] Ó. Belmonte, I. Remolar, J. Ribelles, M. Chover, M. Fernández, Efficient Implementation of Multiresolution Triangle Strips, Proc. of the Computational Science 2002 Conference, vol. 2, pp. 111-120, 2002.

[4] I. Remolar, M. Chover, J. Ribelles, Ó. Belmonte, View-Dependent Multiresolution Model For Foliage, WSCG 2003, pp.370-378, 2003.

[5] X. Decoret, G. Schaufler, F. Sillion, J. Dorsey, Multi-layered impostors for accelerated rendering, Eurographics'99, 18(3), 1999.

[6] L. De Floriani, E. Puppo, P. Magillo, A formal approach to multiresolution hypersurface modeling, Straber W., Kein R., Rau R., editors, Geometric modeling:theory and practice, Berlin:Springer, 1997.

[7] J. El-Sana, A. Varshney, Generalized View-Dependent Simplification, Proc. Of EUROGRAPHICS'99, pp. 131-137, 1999.

[8] J. El-Sana, E. Azanli, A. Varshney, Skip Strips: Maintaining triangle strips for viewdependent rendering, Proc. of Visualization'99, pp. 131-137, 1999.

[9] P. Heckbert, M. Garland, Survey of Polygonal Surface Simplification Algorithms, Siggraph'97 Course Notes, 1997.

[10] B. Lintermann, O. Deussen. Interactive modeling of plants. IEEE Computer Graphics and Applications, 19(1), 1999.

[11] D. Marshall, D. Fussell, A. T. Campbell III, Multiresolution Rendering of Complex Botanical Scenes, Graphics Interface '97, pp. 97-104, 1997.

[12] N. Max, K. Ohsaki. Rendering trees from precomputed Z-buffer views, Eurographics Workshop on Rendering 1996, pp. 165-174, 1996.

[13] NVIDIA Corporation, Cg Language Toolkit, 2002

[14] M. Peercy, M. Olano, J. Airey, and P. J. Ungar. Interactive multi-pass programmable shading. In Proceedings of SIGGRAPH 2000, pages 425–432,2000.

[15] K. Proudfoot,W. R. Mark, S. Tzvetkov, and P. Hanrahan. A real-time procedural shading system for programmable graphics hardware. In Proceedings of SIGGRAPH 2001, pages 159–170, 2001.

[16] P. Prusinkiewicz, A. Lindenmayer, The algorithmic beauty of plants, New York, Ed. Springer-Verlag, 1990.

[17] E. Puppo, R. Scopigno, Simplification, LOD and Multiresolution – Principles and Applications, Eurographics'97, Tutorial Notes, 1997.

[18] J. Ribelles, A. López, Ó. Belmonte, I. Remolar, M. Chover. Multiresolution Modelling of Arbitrary Polygonal Surfaces: A Characterization, Computers & Graphics, 26(3),pp.449-462, 2002.

[19] G. Schaufler. Per-object image warping with layered impostors. Eurographics Rendering Workshop 1998, pp. 145-156, 1998.

[20] Szirmay-Kalos, L. Theory of Three-Dimensional Computer Graphics, Publishing House of the Hungarian Academy of Sciences, 1995.

[21] J. Xia, J. EL-Sana, A. Varshney, Adaptive Real-Time Level-of-Detail-Based Rendering for Polygonal Models, IEEE Transactions on Visualizations and Computer Graphics 3(2), pp. 171-183, 1997.

[22] A. Watt, F. Policarpo, 3D Games Real-time Rendering and Software Technology, Addison-Wesley, 2001.

[23] Microsoft Corporation, DirectX 9.0, 2002