# Rendering of Forest Scenes

Paul Guerrero

September 19, 2006


## Abstract

I will discuss and compare two methods to render large forest scenes. One involves pre-generated impostors with different levels of coarseness, the other one batches up large parts of the scene according to an octree grid. Additionally, a method for LOD blending without transparency and a method for tree shading will also be presented.


## 1 Introduction

Rendering a forest scene involves rendering thousands of trees, each one having a complex geometry when fully modeled. To render such a scene without geometry simplification of any sort would be exceedingly slow, even modern graphic cards can't render hundreds of millions of polygons at interactive frame rates.

Another problem with the naive approach is that instancing trees requires thousands of render calls per frame, which stalls rendering even when rendering only few polygons per tree[1].

[1]Instancing trees requires one render call per tree, although this problem may be alleviated with hardware instancing.

The unique material of forests, when seen from a distance, with its strong parallax effects, make simplification of groups of trees difficult.

Various attempts have been made to cope with those difficulties. One approach described in [1] is to automatically generate multiple LOD levels for each tree and choose one of them for display based on the viewing distance.

[2] propose using a specially tiled 3D texture to represent the forest, which is then displayed in layers. This method does not work well for close-ups of individual trees, but gives good frame rates of realistic forests when viewed from a distance (e.g. for flight simulators).

In [3] trees are rendered using points of increasing coarseness instead of triangles, if screen size or visibility is low. Clustering and averaging of finer points is used to construct the coarse points. This method can be used for a near view at a forest (walk through) as well as for a landscape view of it.

The goal of this project is to display a forest scene consisting of around 75.000 trees at interactive frame rates while maintaining good visual quality.

I have implemented and tested two different methods for forest rendering in Ogre [4], one based on [5], using an impostor for each octant of an octree covering the scene, the other one involving image-based LODs, LOD blending, and special tree shading.

## 2 Impostor Method

The idea of this approach (described in [5]) is the following: Organize the scene in an octree and build one impostor for each octant in each level of the octree. An impostor simply contains six colours, represent-

ing the octant as viewed from each of its six sides. If area of on octant on screen is small enough, the impostor is displayed instead of the contained geometry. The same principle is applied to higher levels in the octree: if the higher octant is small enough, its coarser impostor is displayed instead of the finer child octant impostors.

This method can be implemented efficiently by recursing through the octree, starting at the root and continuing deeper into the octree until either the screen area of an octant is small enough or a leaf octant is reached[2]. In the former case, the octant impostor is drawn and the subtree of this octant is disregarded. In the latter case, the original geometry is drawn.

Impostors are generated in a pre-processing step by rendering the contained geometry of each octant from six sides and averaging each resulting image to one colour. Coarser Impostors in higher levels of the octree can be constructed by averaging the colours of the finer impostors in the child octants.

Impostor Rendering saves rendering time by discarding distant geometry and using impostors of increasing coarseness instead. This speedup comes at the cost of using image-based impostors and the associated loss of visual quality.

## 2.1 Loose and Tight Octrees

In Ogre, the standard scene manager uses an octree to manage the scene and do culling. It is a loose octree, meaning that octants in the same octree level overlap by a factor of 0.5. This octree organization assures that each object is inside the volume of only one octant. The single octant containing the object is determined by the center and extent of the object bounding box. This way, objects can be managed easily and don't have to be split up.

However, loose octrees are not well suited for the impostor method described above, since overlapping impostors cause visible artifacts, like too much opacity on the overlapping areas when using transparent impostors, depth fighting, and a loss of resolution since the octant screen area will be larger than in a standard octree.

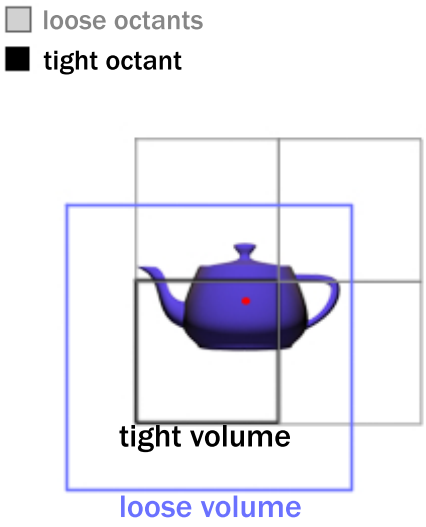


Figure 1: The teapot object has one loose octant and three tight ones. The red dot marks the object center.

To avoid having to use two octrees for the same scene, the existing octree has been adapted to act as loose octree as well as standard octree. I will call the standard octree 'tight' octree from now on to avoid confusion.

To achieve this, each object in the scene is assigned to exactly one loose octant and possibly one or more tight octants, if the object is also inside the tight volume[3] of these octants. In Figure 1, the center of the teapot object determines its single loose octant. The object is completely included in the octant loose volume. If the object would be bigger, it would have to be assigned to an octant at a higher level of the octree. The teapot also has three tight octants, since it is partially included in their tight volumes.

Tight octants of an object are always at the same octree depth as the object loose octant. Octants are created on demand, so there are no empty octants in

[2]It is also possible to define a maximum octree depth to keep the number of impostors low. Impostors are only generated up to this depth.

[3]Tight volumes are the non-overlapping volumes of octants.

the octree.

### 2.2 Impostor Generation

In a preprocessing step, each leaf octant (or octant at maximum depth) is rendered from six sides. The rendered colours are then averaged to calculate the colours of the octants at higher octree levels, resulting in six colours for each octant in the octree.

One difficulty in storing rendered colours is that texture readbacks have to be done. This is generally not recommended and can be very slow. To avoid having to do one readback per render, many octant sides are rendered into one large texture, which is read back afterwards.

### 2.3 Impostor Types

Two types of impostors have been implemented, sprite impostors and cube impostors.

Sprite impostors consist of a sprite that is always facing the camera. The single colour of the sprite is interpolated from three of the six side colours, depending on the viewing angle.

Cube impostors are colour cubes. Each side of the cube has a single colour.

Both impostor types have the extent of the octant tight volume.

To reduce memory consumption, one material is used for as many impostors as possible. The impostor colours are packed into large textures and texture coordinates for impostors are adapted.

### 2.4 Clustering

When clustering, more than one impostor is generated for each octant. Impostors are arranged in a $clusterSize \times clusterSize \times clusterSize$ grid inside the octant. This is more efficient than simply increasing octree depth since the whole cluster is treated as a single mesh and just one render call is needed. The drawback is that only the whole cluster can be shown or hidden, not parts of it.

When building a cluster, $6 \times clusterSize$ sides have to be rendered. Each image contains colour information for $clusterSize \times clusterSize$ impostor sides.

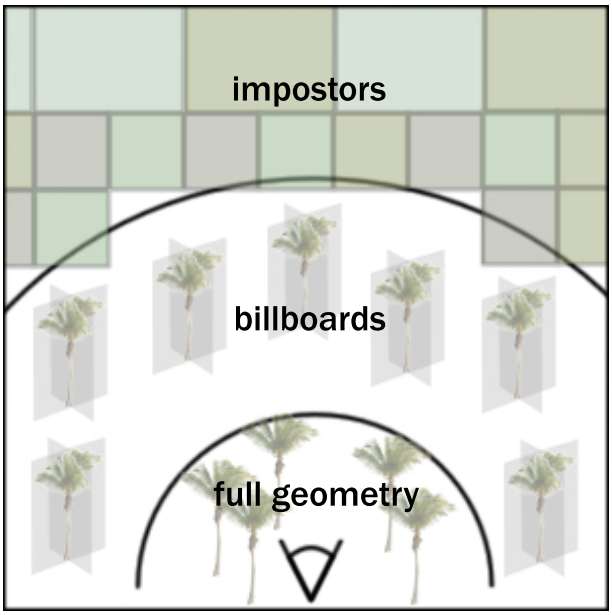


Figure 2: Impostors are rendered at far distances. Between impostors and full geometry, billboards are used.

Parts of the cluster that are hardly visible (too little alpha) are left out. This requires having a separate mesh for each cluster, increasing memory consumption, but, especially in a tree scene, with a relatively thin layer of trees above the terrain, large parts of the cluster geometry can be omitted.

Impostors require at least $6 \times clusterSize^3$ colours for each octant. In the forest scene, this amounts to 138MB, when using 32 bit colours and standard settings.

Impostors in the forest scene are typically rendered at far distances (see Figure 2). Billboard trees are used for the large area between the original geometry and the impostors.

## 3 Static Geometry Method

This method achieves better visual quality by simply continuing to render billboard trees at far distances. The main bottleneck when rendering each instance of a tree separately is the large number of render calls

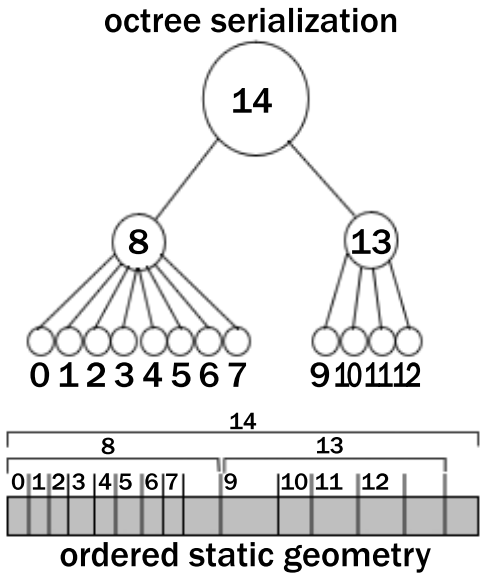


Figure 3: Octree serialization defines the ordering of the static geometry.

issued.

Since trees are assumed to be static, it is possible to group all trees in the scene into one large soup of polygons, which can be rendered by issuing a single render call [4].

The problem with this approach is that software culling and LOD switching can't be done, since only all of the billboard trees can be rendered or none.

One solution could be to use a grid of smaller tree groups and display or hide each group as needed. But the choice of grid resolution would be problematic. When choosing a fine resolution, too many render calls are needed. When choosing a coarse one, there is too little control over the border between static geometry and instanced geometry.

A better method is to use an octree to group the trees in multiple grid resolutions. This is done as follows:

[4]Due to hardware limitations (16 bit indices), more than one render call is needed when rendering large amounts of trees, but the number of render calls needed is very low when compared to separately rendering each tree.

First the octree is serialized by recursively numbering its octants depth-first. Then, the billboard trees are added to the single polygon soup in the order defined by the octree serialization (see Figure 3). For each octant, two indices into the polygon soup are stored, marking the beginning and the end of the octant geometry.

Since only two pointers have to be stored per octant, only minimal additional memory is required compared to the grid method.

Now it is possible to render the geometry contained in any octant volume with one render call. By choosing an octree level, the resolution of the tree group grid can be varied.

The algorithm to render the static billboard trees, starting at the octree root, is described in Figure 4.

```
draw_geometry(octant) {

  if (not_visible(octant))
    return;

  if (dist_to(octant.bound) < LOD_dist)
    for each child octant
      draw_geometry(child octant);
    return;

  if (partially_visible(octant)
      and
      octant_depth < max_cull_depth)
    for each child octant
      draw_geometry(child octant);
    return;

  draw_octant_geometry();
}
```

Figure 4: The algorithm used to render static billboard trees.

An octant is either split up if it is close enough to the camera, so that some of its trees may be drawn using full geometry instead of billboards, or if part of it can be culled.

`draw_octant_geometry` requires one render call. `max_cull_depth` defines the octree depth, after which

octants are no longer split up to do visibility culling. Splitting up octants to do culling only helps to improve performance up to a certain depth, since it also increases the number of render calls needed and the number of culled polygons is smaller close to the leaf octants.

For readability, the non-static billboard trees (billboard trees that are close to the camera and drawn one at a time) are not included in the pseudo-code.

Later, I will show that using static geometry in the forest scene is actually faster than the impostor method described in the last section, while achieving better visual quality.

### 3.1 Implementation in Ogre

The Ogre static geometry class has been modified to support indexing into regions of static geometry. Each octant stores two indices into such a region, pointing to the beginning and end of the octant geometry.

## 4 LOD blending

At far distances, trees are represented as crossed billboards. Crossed billboards were used instead of sprites (billboards facing the camera), since crossed billboards avoid a problem with sprites; when two sprites are close enough to overlap, popping of one sprite in front of the other can be seen at some angles, when rotating around the sprites. However, shading of crossed billboards is more difficult. I will address this problem in the next section.

The LOD blending implemented does not use transparency, instead small texels of the tree are blended in or out, according to a noise pattern (similar to the technique used in SpeedTree[6]).

First, a grey-level noise texture is created in multiple resolutions. This texture is then used by the two blending LOD levels to determine which texels are visible. A blend factor ranging from 0.0 to 1.0 determines the overall visibility of a LOD level. Only texels with an intensity smaller than the blend factor are rendered.

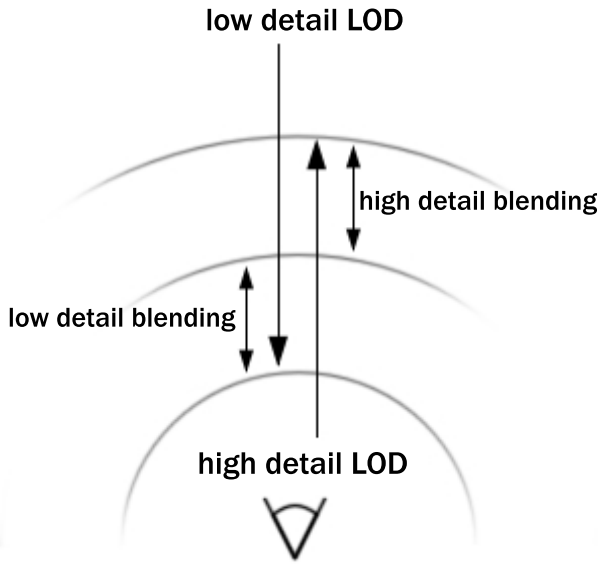


Figure 5: LOD blending is not done in parallel.

The resolution of the noise texture is very important. If it is too small, the texel areas are too large and become obviously visible. If it is too large, the texture becomes blurred to only a few grey values.[5]

Figure 5 shows how LOD levels are blended over the distance. LOD levels are not blended in parallel. First one LOD level is completely blended in, then the other one is blended out. This is done to avoid that the background leaks through.

This kind of blending helps to reduce the 'ghosting' effect seen when using transparency for blending. Since trees usually have a noisy texture (due to the great number of leaves), noise blending is harder to notice in this noisy environment. Additionally, in contrast to transparency blending, no depth sorting or alpha blending has to be done.

LOD blending is implemented in Ogre by extending the entity class to calculate the blend factor. The actual blending is done by vertex and fragment programs.

[5]It may also be necessary to turn off texture filtering to avoid blurring of the texture to few grey values if the texel screen size is small.

# 5 Tree Shading

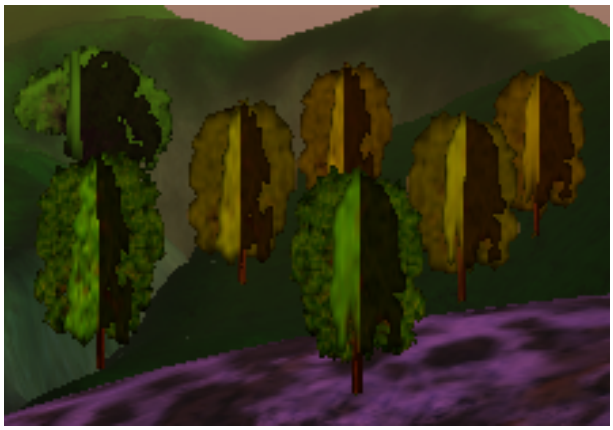

Figure 6: Billboard trees shaded by using the normal orthogonal to the billboard plane.

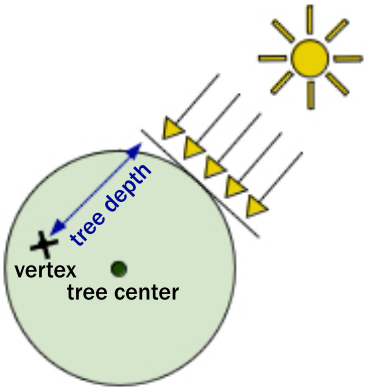


Figure 7: The tree depth of each vertex is used for shading.

The default shading of the crossed billboards (with normals orthogonal to the billboard plane) reveals their flatness (see Figure 6) and the default shading on high detail tree models doesn't take self-shadowing into account, a property that is important for realistic shading in (dense) trees. On a tree, leaves that are on the side opposed to the sun are usually darker since the sunlight has less chance to get through to them.

Taking this property into account, the diffuse factor for a vertex in a tree is determined by its distance from the tree center along the sun direction (see Figure 7).

This shading works well for full geometry trees, but still shows some artifacts when applied to billboards.

First, shading is incorrect when looking at the dark or light side of the tree, since all vertices of the facing billboard are at about the same depth as the tree center, resulting in a medium lightness for the whole billboard. This error occurs, because billboards are located at the center of trees, while in real (dense) trees, we hardly ever see the leaves or small branches at the center of the tree.

Second, the edges of billboards seen at a grazing angle are usually brighter or darker than the the underlying colour of the billboard facing the camera. This makes the edges of grazing angle billboards visible.

Both problems are shown in Figure 8, the encircled billboard edge has a different colour than the underlying billboard. The billboard marked with a red border has a medium lightness, even though it should be darker when viewed from the side opposite to the sun and lighter when viewed from the other side.

To correct these problems, all billboard vertices are projected onto a plane for lighting calculation. This plane has an offset towards the camera, usually half the radius of the tree.[6] This method results in a smooth gradient. Figure 9 shows the corrected shading model for billboards.

Figure 10 shows the results of using the corrected shading. The right image shows the lightness of the billboards. Note how the gradient is now much smoother.

[6]Since most trees have more of an elliptical shape than a spherical one, the tree radius is larger in the vertical directions when calculating the offset.

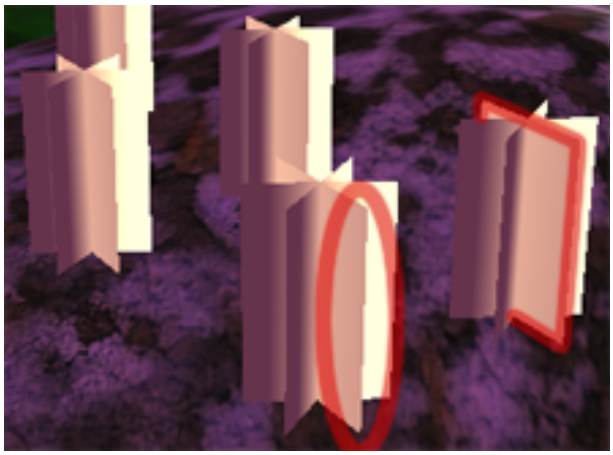

Figure 8: Billboard trees shaded without projecting the vertices to a plane. Only the lightness of the billboards is shown.

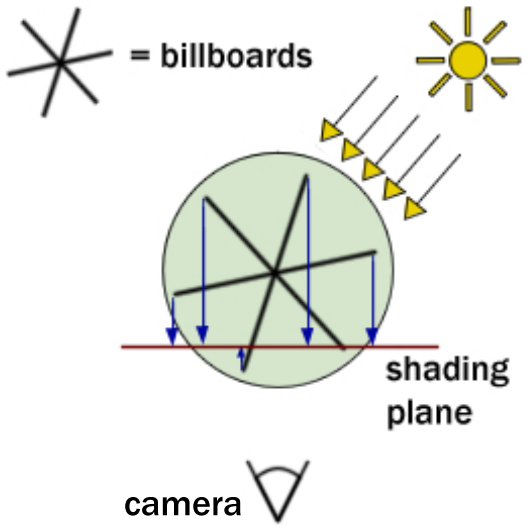


Figure 9: Each vertex of a tree billboard is projected to a plane for shading.

For trees close to the camera, this kind of shading may look too smooth (trees have a very noisy texture). To avoid this effect, in full geometry trees, the normals of each leaf are also taken into account when calculating the diffuse factor.

Note that billboard vertex positions in the static geometry polygon soup are relative to the polygon soup center, not the billboard center. The vertex positions relative to the billboard center are saved as texture coordinates in each vertex, which can then be read by a vertex program.

# 6 Results

The two methods were tested on an Athlon64 (running in 32-bit mode) at 2.2 GHz with a GeForce 7800 graphics card at a resolution of $800 \times 600$ pixels.

For the impostor method, cube impostors were used with a cluster size of 12 and a maximum octree depth of 8. For the static geometry method, a maximum culling depth of 7 was used.

The scene consists of about 75000 trees, a terrain and some smaller objects. Three tree models were used: a palm tree (about 340 vertices in full geometry), catalpa (about 650 vertices in full geometry) and cammelia (in three colours, about 2800 vertices in full geometry). [7] One low detail LOD level was used for each tree model, consisting of three crossed billboards.

Because impostor rendering requires more render calls, impostor rendering performs poorly when compared to static geometry rendering. To achieve the same frame rates, the impostor screen area has to be very large resulting in unacceptable image quality.

Figure 11 shows a comparison of static geometry to impostors.

When using static geometry organized in an octree structure, the frame rate is 35 to over 100 frames per second. When rendering almost all of the 75000 trees using billboards only, the frame rate is around 50 fps.

Figure 12, top shows most of the forest scene. Only billboards are rendered (no trees are close enough to show full geometry).

When getting closer, more render calls are needed

[7] The number of vertices was not the bottleneck in the application. Cammelia forests would render just as fast as palm tree forests. The number of render calls needed was a much larger factor for performance.

and the trees closer to the camera display full geometry. Figure 12, bottom shows a closer view at the trees. When displaying full geometry, the frame rate depends mostly on the number of different materials per tree, as each material needs a separate render call. When 'walking' among the trees, the frame rate is 35 to 100 fps, depending on the number of trees in view.

For comparison, here are results when using the other methods for tree grouping mentioned in section 3:

Drawing each tree separately results in 0.2 - 0.3 fps when viewing most of the 75000 trees. When flying through the scene, the framerate is rarely above 10 fps.

When using a grid with a resolution fine enough to do LOD blending [8] instead of the octree structure, the framrate is in the range of 10-35 fps (10 fps when viewing most of the 75000 trees).

# 7 Conclusion

The results have shown that impostor rendering as described in this document would only be efficient, if large parts of the scene were very distant, such that many trees would be mapped to one or only a few pixels. Although the memory requirements for impostors in such a scene would probably be too high.

A hierarchical static geometry structure organized in an octree gives the ability to hide or show regions of arbitrary size and location inside a static geometry polygon soup, while keeping the number of render calls low. Minimal additional memory is needed.

LOD blending with noise textures is usually less noticeable in tree environments than transparency blending and helps to increase performance, because no depth sorting or alpha blending is needed.

Tree shading that takes the tree depth relative to the sun direction into account hides the flatness of billboards and results in more realistic shading with minimal additional processing needed, although measures have to be taken to avoid that trees close to the camera appear too smoothly shaded.

[8] Each grid cell contains roughly 100 trees.

# References


[1] A. Fuhrmann, E. Umlauf, S. Mantler: Extreme Model Simplification for Forest Rendering. In *Eurographics Workshop on Natural Phenomena* (2005), pp. 57-66

[2] Philippe Decaudin, Fabrice Neyret: Rendering Forest Scenes in Real-Time. In *Rendering Techniques '04 (Eurographics Symposium on Rendering)* (June 2004), pp. 93-102

[3] G. Gilet, A. Meyer and F. Neyret: Point-based rendering of trees. In *Eurographics Workshop on Natural Phenomena* (2005) - http://www-evasion.imag.fr/Publications/2005/GMN05

[4] Object-Oriented Graphics Rendering Engine - http://www.ogre3d.org/

[5] Bradford Chamberlain, Tony DeRose, Dani Lischinski, David Salesin, John Snyder: Fast Rendering of Complex Environments Using a Spatial Hierarchy. In *Proceedings of the conference on Graphics interface* (1996), pp. 132-141

[6] www.speedtree.com, Interactive Data Visualization, Inc

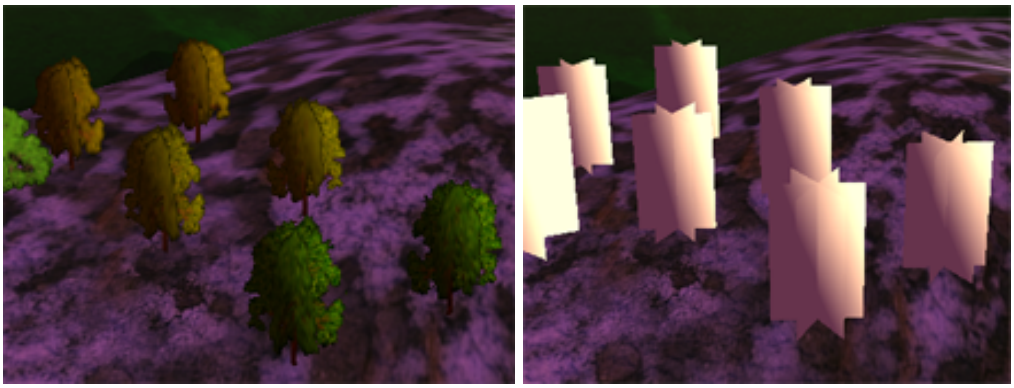

Figure 10: The final billboard tree lighting. Only the lightness is shown in the right image.

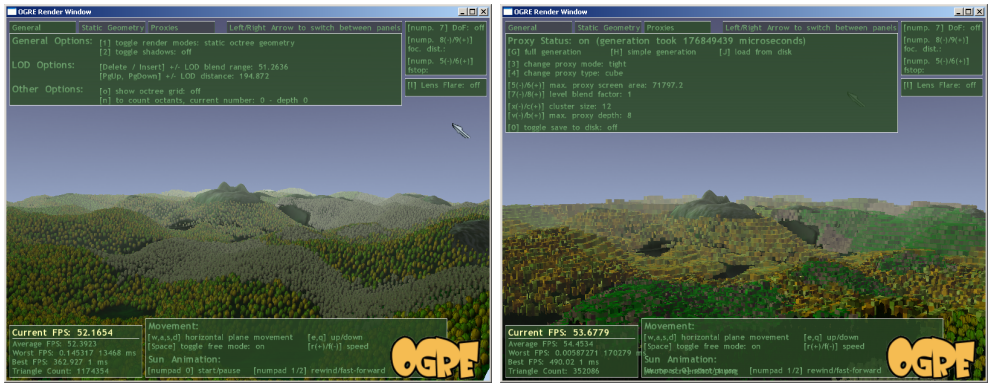


Figure 11: Comparing an image rendered using static geometry (left) to an image rendered using impostors (right). The impostor screen area was adjusted to get the same framerate.

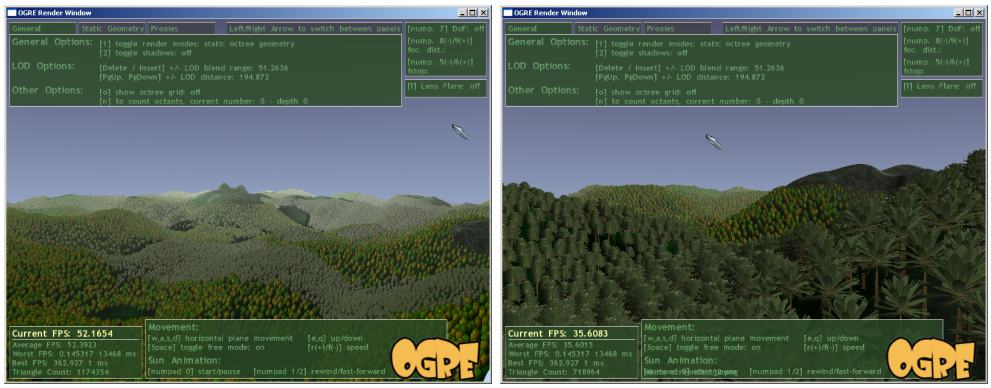


Figure 12: Overview of the scene (left) and a closer look with some full geometry trees (right).