A Three Dimensional Image Cache for Virtual Reality

Gernot Schaufler and Wolfgang Stürzlinger

GUP, Johannes Kepler Universität Linz, Altenbergerstr.69, A-4040 Linz, Austria/Europe
schaufler@gup.uni-linz.ac.at

Abstract

Despite recent advances in rendering hardware, large and complex virtual environments cannot be displayed with a sufficiently high frame rate, because of limitations in the available rendering performance.
This paper presents a new approach of software accelerated rendering which draws from the concepts of impostors, hierarchical scene subdivision and levels of detail. So far software optimization in real-time rendering has merely considered individual objects. This work is actually optimizing the rendering of the whole virtual environment by implementing a three dimensional image cache. It speeds up rendering for large portions of the scene by exploiting the coherence inherent in any smooth frame sequence.
The implementation of the three dimensional image cache is discussed and the savings in rendering load achievable on a suitable hardware platform are presented.

Keywords: viewing algorithms, geometric algorithms, object hierarchies, virtual reality.

1 Introduction

The basic capability of any virtual reality system is to provide the user with a view of the environment. As the user navigates within the environment this view must be updated at least several times per second and recent user movements must be reflected in the view with minimal latency [6]. Usually high-end graphics workstations are used to meet these performance requirements. The environment is modelled with textured polygons which can be rendered with hardware support on these workstations. Typically the number of potentially visible polygons must not exceed several thousands if frame rates of 20 Hz or more should be achieved. A lot of work has been published how to determine the potentially visible polygons for each frame and how to make best use of the available rendering performance in order to generate images of good quality with an acceptable frame rate.
The severe limitation in the number of polygons is due to the fact that every frame is rendered from scratch: from the current point of view the set of potentially visible objects is determined and the images of these objects must be rendered within the available frame time. It would be desirable to reuse most of the image data generated during previous frames if the changes to these images can be neglected in the current frame. So far two systems have been presented which are capable of reusing previously generated images - a hardware solution called the virtual reality address recalculation pipeline [8] and a software approach called dynamically generated impostors [9].
This paper presents a new approach to software accelerated rendering which improves and generalizes the concept of impostors. Unlike dynamically generated impostors the new method can handle intersecting objects and indoor as well as outdoor scenes. It does not rely on the scene to be actually organized into objects but groups together spatially close primitives automatically. A hierarchical three-dimensional image cache stores and allows to reuse image data of parts of the scene, which was generated during previous frames. An update algorithm ensures that images are replaced with new ones if necessary. As a result, those parts of the scene for which images are already available from previous frames need not be rendered from scratch in the

current frame. This paper's examples are based on polygonal rendering. However, the 3d image cache can accelerate any kind of rendering such as free-form surface rendering or ray tracing as well.
The two sections to follow familiarize the reader with previous work in the field of real-time rendering and with the concept of dynamically generated impostors. In section 4 the hierarchical image cache is introduced. The idea is to store the scene in a hierarchical space partitioning data structure, e.g. a k-d-tree, in which images generated for any subtree can be cached and reused if appropriate. The error introduced by this caching can be limited to pixel resolution as will be shown in section 5. Conclusions are drawn in section 6 which will lead on to some possible future work.

2 Previous Work

When generating images of polygonal scenes performance can be improved if only those parts of the scene are sent to the rendering hardware which are visible in the final image. Finding the visible parts can be efficiently implemented as a traversal of a hierarchical space partitioning data structure [3].
If a uniform frame rate is desired, an appropriate level of detail can be chosen for every visible object thereby trading rendering speed for image quality [4]. Automatic algorithms exist to generate such levels of detail from polygonal objects [5][10].
Multiple frame buffer hardware can be used to overcome the drawback of having to render every visible object for every frame [8]. Observing that the images of distant objects move slower on screen than the images of close objects, the objects are rendered into different frame buffers based on their distance from the viewer. The frame buffers containing distant objects can be updated less frequently. The final image is obtained by overlapping the buffers front to back.
Object images are generated as a preprocessing step and are used during rendering as appropriate [7]. For every object images from any viewing direction must be stored in the scene database so that images from any point of view can be rendered (at the cost of high memory requirements).
Such object images can also be generated on the fly for individual objects in sparse outdoor scenes [9]. Instead of the original object model a transparent polygon is rendered with an opaque image of the object mapped onto it (an impostor). However, errors occur in the final images if objects are too close together so that visibility is not resolved correctly by rendering impostors (see figure 10 upper right). The correct image without impostors is shown in the upper left and a difference image in the lower left of figure 10.
Rendering based entirely on images was proposed for walkthroughs of static scenes [2]. The correct view can be extracted from a number of environment maps by real-time image processing. Intermediate frames can also be interpolated from available keyframes [1] and an appropriate image is shown to the user when he moves between the points for which environment maps are available.
As the image caching proposed in this paper is based upon impostors a short introduction to them follows.

3 Dynamically Generated Impostors

The concept of dynamically generated impostors is depicted in figure 1 (for a more in depth discussion of dynamically generated impostors refer to [9]):
A complex object is replaced by a textured polygon facing the viewer (see figure 1) which cannot be distinguished from the original object from the proper point of view. The texture on the polygon is generated by finding a rectangle surrounding the object (or the object's bounding box for convenience) and rendering the object into this rectangle using the graphics hardware. The resulting image is read from the frame buffer and used to define the texture on the impostor rectangle.

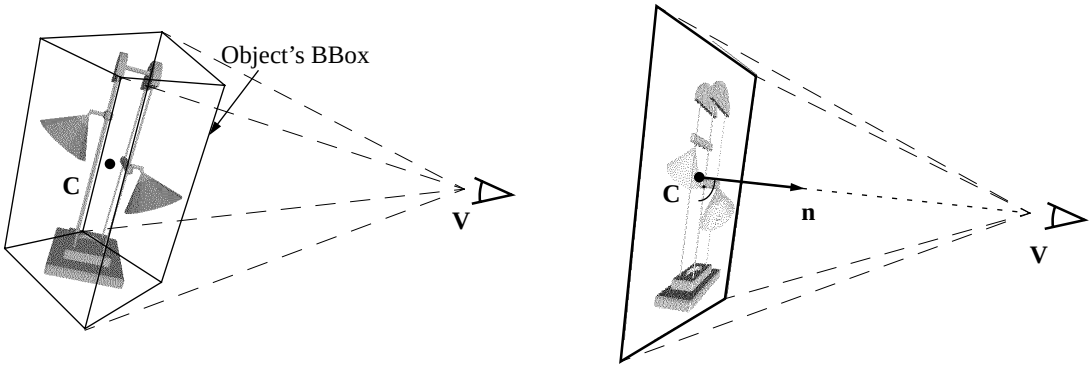

Figure 1. Original Object and Impostor

An impostor can be used as soon as the texels of the texture map are not visible in the final image. This fact can be expressed using angles under which the user sees one texel ($\alpha_{\mathbf{tex}}$), one pixel on screen ($\alpha_{\mathbf{screen}}$) and the whole screen (**fov**):

$$\alpha_{\mathbf{tex}} < \alpha_{\mathbf{screen}} = \frac{\mathbf{fov}}{\mathbf{screenres}} \tag{1}$$

As long as the user only changes his direction of view the images on the impostors need not be changed. This observation allows fast display updates during user head rotations which keeps the display lag short.
When moving sideways in front of an object the image on the impostor and the object's actual image will eventually become different. As soon as the differences would be visible in the final image a new impostor must be generated.
The need for an update can be determined by comparing the angle $\alpha_{\mathbf{trans}}$ to $\alpha_{\mathbf{screen}}$ (see figure 2). $\alpha_{\mathbf{trans}}$ is the angle under which extreme points on the object's bounding box ($\mathbf{B_1}$ and $\mathbf{B_2}$) and their image on the impostor (**B'**) are observed by the user.

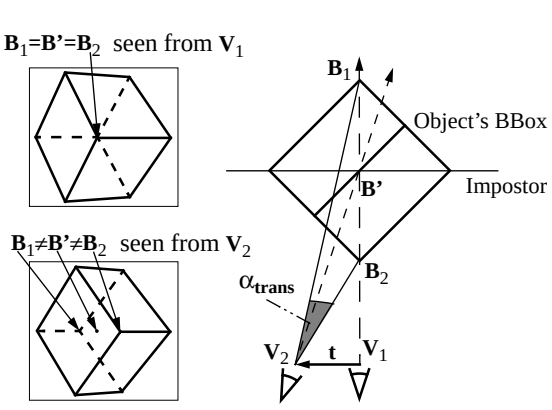

Figure 2. Error angle $\alpha_{\mathbf{trans}}$ due to translation

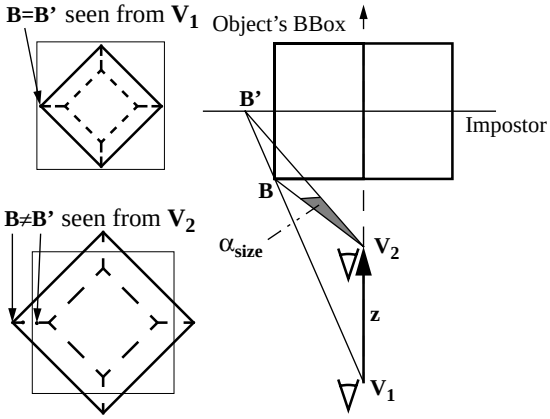

Figure 3. Error angle $\alpha_{\mathbf{trans}}$ due to move-in

A similar situation occurs when moving towards an object. In this case $\alpha_{\mathbf{size}}$ must be compared to $\alpha_{\mathbf{screen}}$ as shown in figure 3.
From those two orthogonal cases one can see that it is necessary to generate a new impostor every time either $\alpha_{\mathbf{size}}$ or $\alpha_{\mathbf{trans}}$ becomes larger than $\alpha_{\mathbf{screen}}$ (see equation (1)).
Problems occur when objects in a cluster or intersecting objects are replaced individually by impostors. For such objects impostors cannot resolve the visibility correctly and the object on the closer impostor will erroneously occlude the other object. These problems are especially annoying in indoor scenes. Examples are books in a shelf or chairs put in under a table (see figure 10 upper right for an extreme case).
Another problem occurs in scenes with large numbers of objects which all must be checked for a necessary impostor refresh. Considerable overhead results especially if the objects do not consist of many polygons. In addition some scenes are given as unstructured polygon lists which makes the use of dynamically generated impostors impossible.

4 A Three Dimensional Image Cache

In order to maintain a three dimensional image cache methods are needed to generate and store image data for parts of a scene. The current implementation of the 3d image cache generates and stores impostors for the contents of axially aligned bounding boxes (bboxes). In a hierarchy of bboxes image data in lower layers can be used in a bottom up fashion to generate images of the contents of higher nodes (considering several impostors as the contents of a bbox).
This hierarchy of bboxes is stored in a hierarchical space partitioning data structure. A k-d-tree is preferable to an octree, because it offers more flexibility: for the 3d image cache this flexibility is used to obtain approximately cubic bboxes. Such bboxes are desirable because they let scene subdivision adapt to overall scene shape and local scene complexity. Moreover the k-d-tree allows to tune the number of sub-boxes contained in one bbox to the overhead involved when generating an impostor from several impostors. Texture maps must be quadratic in this implementation which also makes approximately cubic boxes desirable as impostors then tend to be quadratic as well.
First a bbox for the whole scene is calculated. This bbox is recursively subdivided by the k-d-tree along it's longest side. Primitives (or object models) are partitioned into sub-boxes. If more than one sub-box is intersected, the primitive is stored in all the intersected sub-boxes.
Subdivision terminates if the number of primitives in a bbox falls below a given threshold or if the maximum subdivision level is reached. The number of primitives per bbox must compensate for the overhead in generating an impostor (which depends on the hardware).

4.1 Generating Impostors for BBox Contents

An impostor for the contents of a bbox can be generated in almost the same way as an impostor for an object. In a preprocessing step the part of the scene lying within the bbox is identified and stored with the bbox. Instead of clipping the primitives (lines, polygons, ...) to the bbox boundaries during preprocessing, hardware clipping planes can be used during creating the impostor. Such clipping planes are available on current graphics workstations. Cracks in objects crossing bbox boundaries can be avoided by using a slightly bigger bbox for clipping.
The generated impostor stores an image of the bbox's contents when viewed from the point of view for which the impostor was generated. Rendering the impostors for the bboxes of the k-d-tree correctly resolves visibility for all the primitives as the bboxes in the tree are disjoint.

4.2 Generating Impostors from Impostors

When the impostor of a bbox is invalid for the new point of view, it must be regenerated. If the bbox is not a leaf of the hierarchy, then a new impostor can be generated from the impostors of it's sub-boxes. As for an object a rectangle enclosing the bbox is determined and the sub-boxes' impostors are rendered into it. No clipping needs to be done in this case, as the bboxes form a hierarchy. Rendering the few impostors for the sub-

boxes is much cheaper than rendering all the primitives contained in the bbox. Re-rendering primitives is avoided until impostors in the leaves of the hierarchy become invalid. However, leaf-boxes are small which makes their error angles small as well and therefore their impostors are valid for more frames in general.

4.3 Using the 3d Image Cache

Rendering the impostors of the 3d image cache for the living room scene gives the image shown in figure 11 upper left. Compared to an image obtained by pure polygonal rendering (figure 10 upper left) any differences are due to resampling and are limited to an extent of one pixel as can be seen in the difference image (figure 11 upper right).
The contents of the 3d image cache are visualized in a bird's eye view for two different viewpoints (bottom row of figure 11). The k-d-tree used has a branching factor of eight and a depth of three. Impostors which were generated in the current frame are shown with a red border, impostors which were generated from the node's children are shown with a green border. In the right view parts of the floor and the ceiling close to the viewpoint could not be replaced by impostors due to the texture resolution limit.
For every frame to be rendered the k-d-tree is traversed to determine for which nodes an impostor can be used because the maximum texture resolution will be invisible in the final image and it is also determined which impostors need to be regenerated. Subtree traversal can be pruned if a valid impostor is encountered (as impostors of sub-boxes are always valid for more frames because their images and corresponding error angles are smaller). Pseudocode is shown in figure 4:

```
UpdateCache(Cache *c)
if(IsLeaf(c)) {
   if(ImpostorPossible(c) and ImpostorInvalid(c)) {
      GenerateImpostorFromPrimitives(c)
   }
} else {
   if(ImpostorPossible(c)) {
      if(ImpostorInvalid(c)) {
         forAll(c->child) {
            UpdateCache(c->child)
         }
         GenerateImpostorFromChildren(c)
      }
   } else {
      forAll(c->child) {
         UpdateCache(c->child)
      }
   }
}
```

Figure 4. Pseudo Code to Update the Image Cache

To render the final image the k-d-tree is traversed a second time and impostors are drawn when encountered in the tree which prunes subtree traversal. For bboxes close to the user (which cannot be replaced by an impostor), the contained primitives are drawn and clipped to the bbox boundaries (see figure 5 for the pseudo code). Limiting the cache updates to the visible portions of the k-d-tree is also a possible variation. If the viewing direction is rotated, the impostors for previously invisible bboxes must then be generated during the rotation. This option was turned off during the tests in section 5.

```
DrawCache(Cache *c)
if(Visible(c->bbox)) {            /* clip to frustum */
   if(ImpostorPossible(c)) {
      DrawImpostor(c)
   } else {
      if(IsLeaf(c)) {
         ClipTo(c);
         DrawPrimitives(c);
         ClipOff();
      } else {
         ForAll(c->child) {
            DrawCache(c->child)
         }
      }
   }
}
```

Figure 5. Pseudo Code to Draw the Image Cache

If the primitives in the scene are organized into objects and multiple levels of detail are available for each object, then the objects could be drawn with an appropriate level of detail based on their size on the screen. Such a selection scheme would guarantee that the same level of detail is selected for one object in each bbox it intersects and would adapt the work for drawing a bbox's contents to the distance from the viewer. Moreover rendering an impostor for a node in the tree implies a certain distance to the user. Therefore, only one level of detail per object must be stored per node and this level of detail is always used for regenerating the impostor for this node. With levels of detail the caching of images can be limited to one layer in the k-d-tree, namely the first layer of nodes for which impostors are useful. The texture memory necessary to store impostors of lower levels can then be saved.

5 Results

The department of computer graphics at the local university does not own a graphics workstation which supports texture mapping in hardware. Therefore, no measurements of execution times on a suitable hardware platform can be presented. However three different kinds of investigations have been carried out: first, a visualization of the 3d image cache updates was done for a flat environment so that the update events in the cache can be shown in two dimensions. Second, a frame sequence was rendered on a graphics workstation having support for gouraud shaded polygons (an INDIGO R3000 capable of 39k triangles/second) at a small screen resolution to minimize the effects of the software implementation of textured polygons (where the pixel fill rate is the bottleneck for triangles covering more than a few pixels). Third, the performance of the 3d image cache on a suitable hardware platform was simulated by replacing the operations not available on the R3000 system by other operations which were selected to match the execution speed of the desired operations on the required high-end graphics workstation. The statistics for all three test are shown in figures 6 to 8.

5.1 Visualization of cache behaviour

In the following investigation a very large and complex scene is assumed to be stored in the k-d-tree of the 3d image cache. For illustration purposes the scene is assumed to be flat enough so that no subdivision occurs along the projecting z-axis. As a worst case the scene is of homogenous complexity and the k-d-tree is always subdivided to the predetermined maximum. (In a typical walkthrough scenario some of the bboxes will be empty and therefore will not be present in the k-d-tree.) The diagram of figure 6 and the images of figure 9 address the question: which impostors need to be regenerated during a diagonal walkthrough of the scene.

Views from above show the necessary impostor updates for every 6^{th} frame of a total of 120 frames. A red square indicates a leaf bbox in the k-d-tree for which a new impostor was generated in the current frame. A green square indicates an intermediate bbox for which the impostor could be updated by reusing the impostors of it's sub-boxes. White squares indicate bboxes for which an available impostor was used. In the black area impostors cannot be used because the maximum texture resolution of 256 by 256 pixels would have been exceeded. Note how the used level of the k-d-tree adapts to the viewer position and accounts for the lower amount of coherence near the user. This pattern is entirely due to the chosen maximum texture resolution and the decision whether a bbox can be replaced by an impostor or not.
Some green squares in figure 9 contain smaller squares signifying that the corresponding impostor was generated by combining impostors of subnodes. The black bordered squares indicate the level of nodes in the k-d-tree used to generate the final image.
For each of the 120 frames the number of reused impostors, the number of impostors which could be generated from sub-box impostors (combined impostors in figure 6) and the number of impostors which needed to be regenerated (new impostors) is shown and should be compared to the total number of 341 bboxes in the k-d-tree. The total amount of memory required for the texture maps of the impostors is given in 100k bytes.
As can be seen a small fraction of impostors in the k-d-tree actually needs regeneration. Only those objects or parts of the scene, which lie inside such a bbox need to be rendered for the current frame. Combining impostors from impostors one level down the k-d-tree is a cheap operation and saves drawing the whole bbox contents.
In all simulations no clipping to the viewing frustum was assumed. If the 3d image cache is held up to date without considering viewing direction or clipping to the viewing frustum, rapid rotation of the viewing direction can be performed efficiently because most of the needed image data is already in the cache - only the scene portion around the user needs to be drawn as well as the impostors.

5.2 Simulation of cache behaviour

The next two tests were run for a scene of a procedurally generated forest consisting of 100 trees (approx. 105000 polygons, see figure 10 lower right). The first frame sequence generated depicts a sideward translation in front of the forest with 100 generated frames. The k-d-tree of the 3d image cache had a maximum depth of four with a total of 585 bboxes. The left graph of figure 7 compares new impostors, combined impostors and reused impostors. The amount of bboxes which could not be replaced by an impostor is shown in blue. The right graph of figure 7 shows new impostors, combined impostors and reused impostors of a total of 585 for a zoom from one corner of the forest towards its centre. The amount of bboxes which could not be replaced by an impostor is also given. Although it seems that the number of used impostors is comparable to the number of bboxes drawn using the primitives it will be obvious from the drawing times that a far larger part of the scene is covered by these impostors.
To approximate the performance of the 3d image cache on a suitable graphics workstation as close as possible the functionality not supported in hardware on the available R3000 workstation were replaced by operations which are equivalent in execution time to the required operations on texturing hardware. From data sheets of high end graphics workstations it can be seen that textured polygons are about twice as costly to draw as gouraud shaded polygons because of the additional access to texture memory for every pixel. Therefore textured polygons were replaced by drawing a gouraud shaded polygon with the same vertices twice. The texture definition operation used for an impostor update was replaced by two read operations of the image data from the frame buffer - the second transfer accounts for the copying of the image to the texture memory.
Although this test was run on a machine with no hardware support for texture mapping the frame times for the proposed rendering method were only longer a few times than without caching. Moreover, changes of the viewing direction would have been possible at almost no cost, as the necessary image data is in the cache and is valid. The left graph of figure 8 compares the times to render the frames for both rendering without caching and rendering using the 3d image cache (in 1/10 of a second) for the translation sequence, the right graph for the zoom in sequence. Drawing times (shown in figure 8) using the 3d image cache stay below the frame times

of usual polygonal rendering even though the impostors were also updated for the part of the scene not currently in the user's field of view.

6 Conclusions and future work

This paper presented the three dimensional image cache, a hierarchical data structure to speed up any kind of rasterized rendering on a graphics workstation which supports texture mapping in hardware. The 3d image cache uses dynamically generated impostors to store image data for the primitives contained in the nodes of a k-d-tree. During rendering these impostors are reused for several frames and are updated only if the differences to the image of the original primitives would exceed a pre-specified threshold (i.e. one pixel).
Impostor updates for leaves of the k-d-tree involve drawing the primitives. The impostors for intermediate nodes are refreshed from the image data stored in their subnodes.
If levels of details are available for the objects in the scene, a static level of detail selection algorithm can be employed to bound the amount of rendering load per node content. In this case the hierarchical caching of image data need not be used resulting in less texture memory requirements.
As the system has been implemented using the portable graphics library OpenGL future work will be to tune the system to a graphics platform supporting the required operations in hardware. Such platforms include the Pixel Flow system developed at the University of North Carolina and the SGI Reality Engine system.
In the future the system will be employed in a large environment navigation tool which pages in the environment portions around the user from secondary storage automatically.
Moreover the 3d image cache will be incorporated into an existing animation system based on ray tracing as the presented algorithm is independent of the method used to render primitives. Considerable speed up should be possible for this kind of rendering as well.

References

[1] Chen, Shenchang Eric and Lance Williams, *"View Interpolation for Image Synthesis"*, Computer Graphics (SIGGRAPH `93 Proceedings) (Aug. 1993) pp 279-288.

[2] Chen, Shenchang Eric, *"Quicktime VR - An Image-Based Approach to Virtual Environment Navigation"*, Computer Graphics (SIGGRAPH `95 Proceedings) (Aug. 1995) pp 29-38.

[3] Clark, James H., *"Hierarchical Geometric Models for Visible Surface Algorithms"*, Communications of the ACM 19 10 (Oct. 1976) pp 547-554.

[4] Funkhouser, Thomas A. and Carlo H. Séquin, *"Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments"*, Computer Graphics (SIGGRAPH `93 Proceedings) 17 (Aug. 1993) pp 247-254.

[5] Heckbert, Paul and Michael Garland, *"Multi-Resolution Modeling for Fast Rendering"*, Proceedings of Graphics Interface `94 (May 1994) pp 43-50.

[6] Helman, James L., *"Designing Virtual Reality Systems to Meet Physio- and Psychological Requirements"*, Applied Virtual Reality Course (ACM SIGGRAPH `93) (Aug. 1993) pp 115-1 - 5-20.

[7] Maciel, Paulo W. and Peter Shirley, *"Visual Navigation of Large Environments Using Textured Clusters"*, Symposium on Interactive 3D Graphics (April 1995) pp 95-102.

[8] Regan, Matthew and Ronald Post, *"Priority Rendering with a Virtual Reality Address Recalculation Pipeline"*, Computer Graphics (SIGGRAPH `94 Proceedings) (July 1994) pp 155-162.

[9] Schaufler, Gernot, *"Dynamically Generated Impostors"*, MVD'95 Workshop (Nov. 95) pp 129-136.

[10] Schaufler, Gernot and Wolfgang Stürzlinger. *"Generating Multiple Levels of Detail for Polygonal Geometry Models"*, 2nd Eurographics Workshop on Virtual Environments 95 (Jan. 1995) pp 53-62.

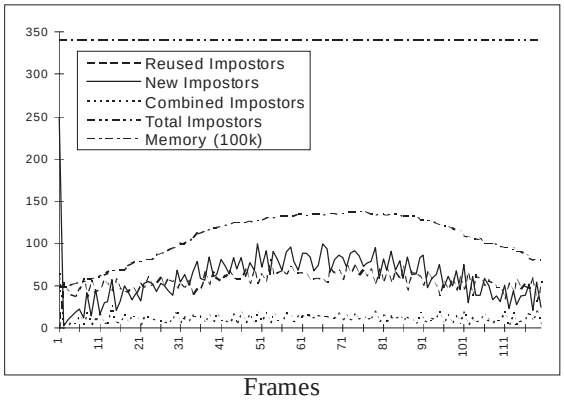

Figure 6. Statistics to Simulation of Frame Sequence in Section 5.1

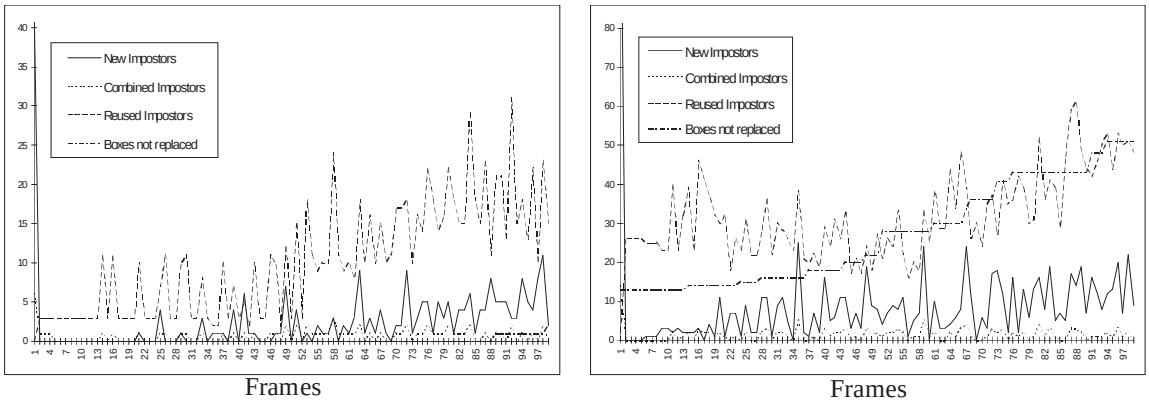

Figure 7. Statistics to Simulation of Frame Sequence in Section 5.2

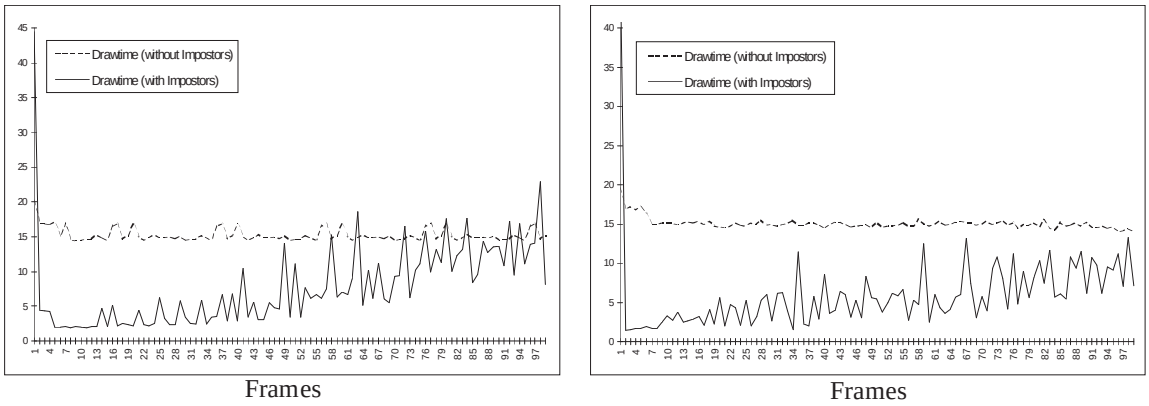

Figure 8. Comparison of Drawing Times in Section 5.2

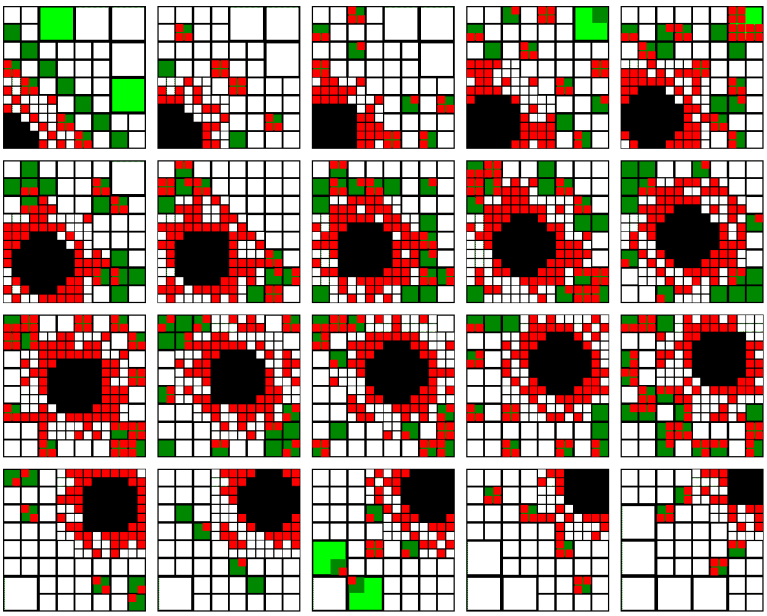

Figure 9. Selected Frames of Simulation Sequence Described in Section 5.1

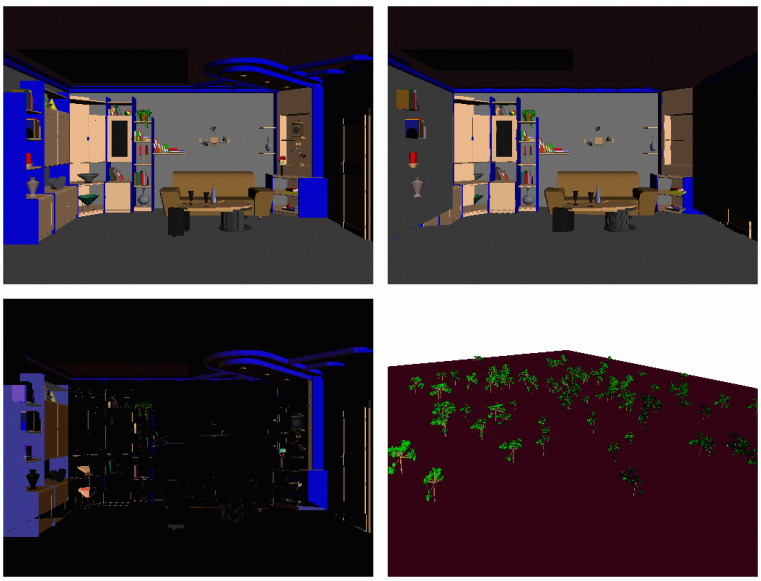

Figure 10. upper left: Original Living Room Scene; upper right: Problems with Object Impostors; lower left: Difference Image; lower right: Tree Scene

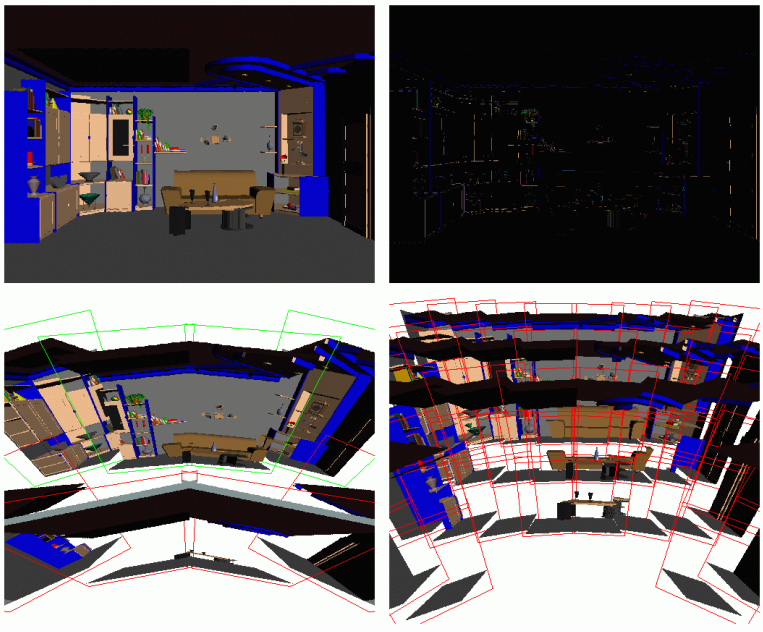

Figure 11. upper left: Result Generated with 3d Image Cache; upper right: Difference Image; lower row: Sample Bird's Eye Views of Cache Contents