Hierarchical Image-Based Rendering using Texture Mapping Hardware

Nelson Max, University of California, Davis, max2@llnl.gov
Oliver Deussen, University of Magdeburg, deussen@isg.cs.uni-magdeburg.de
Brett Keating, University of California, Davis, brettk1@home.com

Abstract. Multi-layered depth images containing color and normal information for subobjects in a hierarchical scene model are precomputed with standard *z*-buffer hardware for six orthogonal views. These are adaptively selected according to the proximity of the viewpoint, and combined using hardware texture mapping to create "reprojected" output images for new viewpoints. (If a subobject is too close to the viewpoint, the polygons in the original model are rendered.) Specific *z*-ranges are selected from the textures with the hardware alpha test to give accurate 3D reprojection. The OpenGL color matrix is used to transform the precomputed normals into their orientations in the final view, for hardware shading.

1. Introduction

Image-based rendering using depth images has become a promising technique for rendering real-world scenes acquired as images, or models with very high polygon counts. Our strategy is to reproject for a new viewpoint one or more precomputed images. This involves using the depth at each pixel, together with the pixel address, to reconstruct a 3D surface point. This point is multiplied by a matrix Q, which is the product of the inverse of the viewing matrix for the precomputed view and the viewing matrix for the desired new view, and then projected into the new view.

Chen and Williams [1] achieve the reprojection using texture flow, which is coherent on large areas where the same smooth surface is visible. For scenes containing vegetation, where the depth varies discontinuously with high spatial frequency, it is more effective to transform each pixel independently, usually in software. McMillan and Bishop [2] showed that a single image can be processed in a "painter's" order that does not require a *z*-buffer for the output image, and Shade *et al*. [3] extended this to multiple *z*-layers at each pixel. Max [4] and Shade *et al*. [3] used the multiple layers to include more of the depth complexity in the pre-computed images, which might become "disoccluded" in the new view.

Recently, Schaufler[5, 6] has shown how to reproject single layer *z*-buffer images using texture mapping hardware, by storing the depth in the alpha channel, and testing it for equality during multiple passes through the texture. The alpha channel of the texture is loaded by reading back the depth buffer of the precomputed view. During rendering, it is scaled by a factor that compresses the 8 or 12 bit alpha range into a smaller number of integer values. For each of these integers n, a polygon covering the corresponding depth layer in the precomputed view is transformed into the new view by the matrix Q, and rendered with the alpha test set to transmit only those pixels with alpha equal to n. The color channels of the texture were used by Schaufler to store preshaded colors. Here we use them to store unshaded color (diffuse reflectivity) in one pass, and to store normal components in a second pass.

Max [7] reprojected precomputed images of various subparts of a tree, organized in a hierarchical model. These were adaptively selected according to the detail needed for the current viewpoint. This scheme requires shading during or after reprojection, because the subparts are used in the hierarchical model at various orientations. Westermann and Ertl [8] show how the color matrix can be used with a surface-normal texture to reorient the normal and shade the surface in hardware.

In the current work, we have combined the techniques of [5], [7], and [8] into a completely hardware-based method to reproject and shade hierarchies of images. The standard *z*-buffer hardware is used for final visibility determination, so the input can be processed in any order. The next section explains the basic reprojection scheme, and the following one deals with the shading.

2. The hierarchical reprojection algorithm

The hierarchical model uses the standard philosophy of constructing complex objects from repeated instances of subparts, each scaled, translated, and/or rotated into position by a 4 x 4 matrix. Several systems for modeling vegetation use this hierarchical philosophy, together with a substantial amount of randomness, so that many geometrically inequivalent subobjects are used at each level of the hierarchy. Here, since each subobject must have images precomputed in several views and used as textures, this randomness must be very limited, as in Brownbill [9], so that only one or two inequivalent objects are used at each level. Our system can parse and render from input files that contain both the hierarchical objects of Max [7] and those in the Rayshade format [10], such as produced by Brownbill[9] and Lintermann and Deussen [11].

In order to decide whether to reproject an object as a whole, the reprojected size of the precomputed pixel closest to the viewpoint is compared to a threshold, which is approximately the size of an output pixel. If the reprojected size is less than the threshold, reprojection will provide sufficient detail, so the whole object is reprojected. Otherwise, the algorithm goes deeper into its detailed description, in terms of subobjects. If a part of the model is so close to the viewer that no precomputed image has enough detail, the polygons in the original model are rendered.

In Max [7], the threshold was fairly small, in order to avoid gaps between reprojected pixels. Shade *et al.* [3] filled in these gaps by reprojecting each input pixel as an appropriately sized “splat” of several output pixels. Texture mapping hardware can automatically find the nearest precomputed texture pixel for a reprojected output pixel, so gaps solely from texture resampling are less of a problem. (Nearest neighbors must be used, rather than the smoother linear or mip-map texture interpolation, because interpolating or averaging the depth in the alpha channel could lead to meaningless results when the correct depth is discontinuous.)

The key step is to reproject a precomputed image, using the alpha test. As shown in [5], even though this alpha test is for equality to a single value, it can be made to give some overlap in the *z*-ranges for adjacent layers, in order to reduce visible gaps between them when viewing slanted smooth surfaces. However this does not completely eliminate gaps from surfaces which are at too steep a slant in the precomputed view, and cannot show surfaces that are occluded in the precomputed image. Both Schaufler [5, 6] and Max[4, 7] reproject several views to help solve these problems. In this paper we use all six views along the positive and negative directions on the *x*, *y*, and *z* axes. This guarantees that every surface will be positioned (although possibly not visible) in at least one pair of images at an angle which is not too slanted, and also that thin round objects like twigs can have all sides represented. The same six views were also used for image based rendering by Lischinski and Rappaport[12].

Shade *et al*. [3] and Max[4, 7] used multiple depth layers in the precomputed images, whose depths were stored individually in a sorted list at each pixel, and thus could not be easily generated in standard *z*-buffer hardware. To take advantage of hardware rendering, we slice the object by hardware *z* clipping into multiple slabs, and reproject each of them by the method of Schaufler[5]. Since each slab is viewed from both the front and the back (in two of the six fixed viewing directions) this completely handles slabs with a depth complexity of two (*i. e*. a viewing ray intersects at most two surfaces per slab). The RGBA images for the multiple slabs are placed together in a single texture for each precomputed view. These are each loaded as a separate texture object, using OpenGL texture binding.

A related technique was described by Meyer and Neyret[13]. They made many more *z*-clipped slabs and put each in a separate texture. In general, a smooth closed surface intersects a slab in a collection of strips, each bounding an interior region of pixels. Meyer and Neyret filled in these interior regions with opaque textured pixels, and thus eliminated the cracks visible in our method when the surface is viewed at a steep angle. However, since our bottleneck is in the loading of many different textures, this method is impractical for us.

We preprocess each frame by going once through the hierarchical model doing no rendering, but instead accumulating a list of reprojection matrices *Q* for the multiple instances of each precomputed view of each object. In a second pass, we go through all the precomputed views in order, and for each non-empty list, we bind the appropriate texture and reproject all the instances using it. This reduces texture swapping. We also render in the hardware pipeline the polygonal descriptions for parts of objects that are so close that resampling is inadequate. In a third pass, we similarly reproject or render all the surface-normal images and use them for shading, as described below. The scene parser must repeatedly read the same object description files, so these are first loaded once into memory, and then processed from there.

3. Normals and shading

To precompute a separate RGBA texture, with the *x*, *y*, and *z* components of the surface normal in the R, G, and B components of the texture, standard color interpolation is used while rendering the polygons in the model. The polygonal tessellations for curved surfaces must be fine enough so that when the vertex normals are interpolated in hardware, they remain approximately unit vectors. The polygon vertex colors are set by scaling and biasing the normal components (N_x , N_y , N_z), from the range [-1, 1] into the range [0, 1], as in Westermann and Ertl [8], so that

$$(red, green, blue) = (.5 + .5{*}N_x\,,\ .5 + .5{*}\,N_y\,,\ .5 + .5{*}N_z)\ . \qquad (1)$$

When the precomputed texture is reprojected into a new view, these normals must be rotated into their new positions, using M_{rot}, the rotation part of the reprojection matrix Q, after scaling and translation are removed. This is done using the 4 x 4 color matrix, an SGI OpenGL extension in the "imaging pipeline", which multiplies the RGBA values from an array in memory as they are being stored into the texture table.

If $L = (L_x, L_y, L_z)$ is the vector to a light source at infinity, we used the color matrix

$$CM = \begin{bmatrix} L_x & L_y & L_z & 0 \\ L_x & L_y & L_z & 0 \\ L_x & L_y & L_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} M_{rot} \ . \qquad (2)$$

To remove the scale and bias of equation (1), we used a post-color-matrix scale of 2, and a post-color-matrix bias equal to minus the sum of the entries in one of the identical first three row of *CM*. (Westermann and Ertl [8] did this instead by multiplying the product in equation (2) on the right by an appropriate constant matrix, but this assumed that alpha is always 1, while we are using alpha to store the depth.) The color data are automatically clamped to [0, 1] before being stored in the texture map. This sets the shading value to zero for surfaces facing away from the light source, and gives the shaded image that would result if all the model surfaces were perfectly diffuse white.

This black and white shaded image is multiplied by the unshaded color image by copying it over the color image, using a blending weight of zero for the source image, and a blending weight of the source color for the destination image. During this image copy (we used an image read and write) we used a bias of *ambient*, and a scale of (1. - *ambient*), in order to add ambient illumination to the final image.

The same lists of reprojection matrices Q for each needed object view are used for reprojecting the normal textures. Each texture is loaded into system memory only once per non-empty list, but since the color matrix is only in the imaging pipeline, not the fragment pipeline, the texture map has to be reloaded from memory with a different color matrix for each different matrix Q on the list.

4. Results and future work

Figure 1 (see color section at the rear of this volume) shows a view of a maple forest, produced in 5 minutes by this method, using one 195 Mhz R10000 processor on an SGI Onyx, InfiniteReality. Figure 2 shows the hardware rendering of the complete model of 955,871 subobjects with 17,205,476 polygons, which took 11 minutes. Much of this time was spent in the software traversal of the deepest levels of the hierarchy. It took about 3 hours to prerender the colors, normals, and depths in the 6 orthogonal views, each with 6 depth slabs, for the five levels in the hierarchical model. The complete polygonal models were used. Much of this time could be saved, at some cost in accuracy, if reprojections of prerendered lower levels of the hierarchy were used in prerendering the higher levels. The precomputed textures were stored on disc as run length encoded SGI image files, requiring 15.4 megabytes.

Our goal was to produce complex images like those in Deussen *et al*. [14], so we extended the parser of Max[7] to read Rayshade files. Figure 3 shows a forest containing a mixture of the maple trees in figure 1, with oak trees modeled by the system of Lintermann and Deussen[11]. It took 46 seconds to render, using only the precomputed images of the whole trees. A polygon rendering took 20 minutes. Figure 4 shows a closer view, which took 109 seconds because it also needed precomputed views of branches. Figures 5 is a close-up showing textured leaves, and figure 6 is a long view.

Because normals are represented in the precomputed images, it should be possible to use bump-mapped texture for the bark on the tree trunks and branches, which would make them look more realistic.

Currently, our leaf polygons have only one normal, which can become reversed when the wrong side of the leaf is visible. If the normals were transformed by M_{rot} alone, to form a normal image, instead of using equation (2) to take their dot product with L, then a software post-process could reverse the normals that point away from the viewer. Also, given a corrected normal image, table look-ups, perhaps using "pixel textures", could be used to produce the radiance-based shading of Max *et al*. [15]. Since the "shadow buffer" of Williams [16] can also be generated by image-based rendering, it should be possible to add post-process shadows to this algorithm, as in Max *et al*. [15], but the quantization of the depth onto the parallel planes used in the reprojection technique might cause artifacts in the shadows.

The accuracy in reprojecting using the alpha test depends on the spacing of the polygonal layers into which the depth is quantized. This depends on the total number of layers from all the slabs, but not on the number of slabs sliced from the object. So the cost in textured pixels ("fragments") is not increased with multiple slabs; only the texture storage and transfer costs increase. The speed/quality trade-offs involved in setting these two quantities should be further explored.

In the "standard" layered depth images of Shade[3] and Max[4, 7], the first layer at a pixel contains the surface, if any, closest to the viewpoint. The next layer contains the surface, if any, just behind the first one, and so forth. Each layer could potentially involve the entire depth range of the object, so the fragment cost could grow in proportion to the number of layers. However, no surfaces would be lost from occlusion, even if only one view were taken along each of the three positive x, y, and z axis directions, instead of the six positive and negative ones. It is difficult to precompute such images with current hardware, but possible in software. If the bottleneck is in texture loading rather than in fragment processing, as it is in our current implementation, the "standard" layered depth images should be considered. Future hardware which permits rendering from compressed textures, or at least loading texture maps from compressed data in memory, should alleviate this current bottleneck, since most of each slab or layer is empty and transparent. Concurrent prefetches of texture from disc should also help.

Acknowledgments

This work was performed under the auspices of the U. S. Department of Energy by Lawrence Livermore National Laboratory under contract W-7405-ENG-48. Mark Duchaineau provided the X-window management software, and Dan Schikore, Rüddiger Westermann, Gernot Schaufler, Brian Cabral, and Mark Duchaineau gave helpful information and suggestions. Jan Nunes and Ross Gaunt helped record the videotape.

References

1. Chen, Shenchang Eric, and Lance Williams, “View Interpolation for Image Synthesis”, ACM Computer Graphics Proceedings, Annual Conference Series 1993, pp. 279 - 288.

2. McMillan, Leonard, and Gary Bishop, “Plenoptic Modeling: An Image-Based rendering System”, ACM Computer Graphics Proceedings, Annual Conference Series 1995, pp. 39 - 46.

3. Shade, Jonathan, Steven Gortler, Li-wei He, and Richard Szeleski, “Layered Depth Images”, ACM Computer Graphics Proceedings, Annual Conference Series 1998, pp. 231 - 242.

4. Max, Nelson and Keiichi Ohsaki, “Rendering Trees from Precomputed Z-Buffer Views”, in “Rendering Techniques ’95 (Hanrahan and Purgathofer, eds.) Springer, Vienna (1995) pp. 74 - 81.

5. Schaufler, Gernot, “Per-Object Image Warping with Layered Imposters”, in “Rendering Techniques ’98” (Drettakis and Max, eds.) Springer, Vienna (1998) pp. 145 - 156.

6. Schaufler, Gernot, “Image-based Object Representation by Layered Impostors”, Proceedings of ACM Symposium on Virtual Reality Software and Technology ’98, Nov. 1998, Taipei, Taiwan, pp 99-104.

7. Max, Nelson, “Hierarchical Rendering of Trees from Precomputed Multi-Layer Z-Buffers”, in “Rendering Techniques ’96 (Pueyo and Schröder, eds.) Springer, Vienna (1996) pp. 165 - 174.

8. Westermann, Rüdiger and Thomas Ertl, “Efficiently Using Graphics Hardware in Volume Rendering Applications”, ACM Computer Graphics Proceedings, Annual Conference Series 1998, pp. 169 - 177.

9. Brownbill, Andrew, “Reducing the storage required to render L-system based models”, Master’s Thesis, The University of Calgary, 1996.

10. Kolb, Craig, “Rayshade”, http://graphics.stanford.edu/~cek/rayshade.

11. Lintermann, Bernd, and Oliver Deussen, “Interactive Modelling of Plants”, IEEE CG&A Vol. 19 No. 1 (1999).

12. Lischinski, Dani, and Ari Rappoport. “Image-Based Rendering for Non-Diffuse Synthetic Scenes, in “Rendering Techniques ’98” (Drettakis and Max, eds.) Springer, Vienna (1998) pp. 301 - 314.

13. Meyer, Alexander, and Fabrice Neyret, “Interactive Volume Textures” in “Rendering Techniques ’98” (Drettakis and Max, eds.) Springer, Vienna (1998) pp. 157 - 168.

14. Deussen, Oliver, Pat Hanrahan, Bernd Lintermann, Radomír Mech, Matt Pharr, and Przemyslaw Pruisinkiewicz, “Realistic modeling and rendering of plant ecosystems”, ACM Computer Graphics Proceedings, Annual Conference Series 1998, pp. 275 - 286.

15. Max, Nelson, Curtis Mobley, Brett Keating, and En-Hua Wu, “Plane-Parallel Radiance Transport for Global Illumination in Vegetation”, in “Rendering Techniques ’97” (Dorsey and Slusallek eds.) Springer, Vienna (1997) pp. 239 - 250.

16. Williams, Lance “Casting Curved Shadows on Curved Surfaces”, Computer Graphics, Vol. 12, No. 3 (1978 Siggraph Conference Proceedings) pp. 270 - 274.

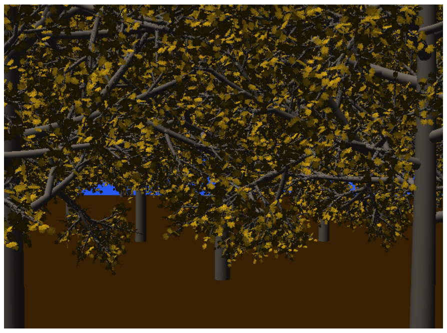
Fig. 1. Maple forest, rendered by IBR.

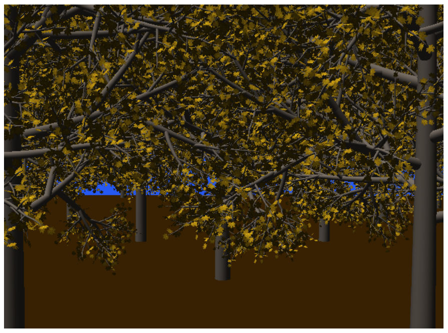
Fig. 2. Maple forest, rendered with polygons.

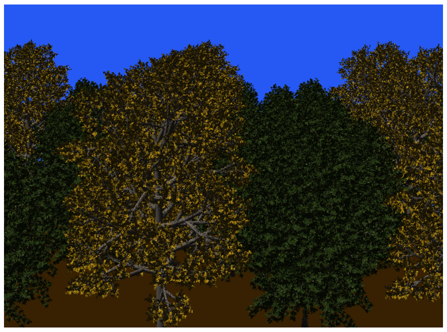
Fig. 3. Mixed oak and maple forest, by IBR.

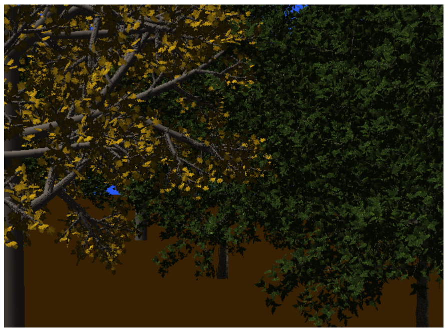
Fig. 4. Closer view of maple and oak forest.

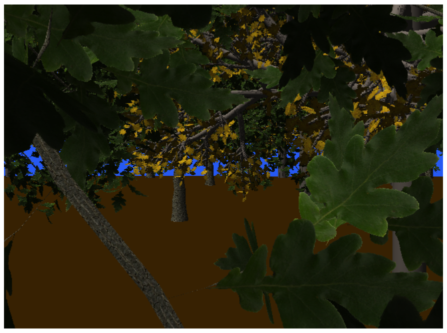
Fig. 5. A close-up, showing leaf texture.

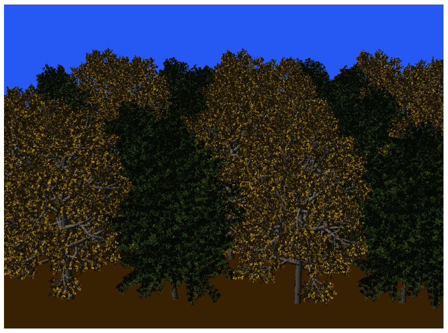
Fig. 6. A long view of the whole forest.