Interactive Stereoscopic Rendering of Voxel-based Terrain

Ming Wan, Nan Zhang, Arie Kaufman, and Huamin Qu

Center for Visual Computing (CVC)
and Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400

Abstract

We present an interactive stereoscopic rendering algorithm of voxel-based terrain. It provides unambiguous depth information of a terrain scene by generating perspective images for a pair of eyes with a horizontal parallax. The left-eye image is generated using a fast ray casting algorithm accelerated by exploiting a specific ray coherence method in the voxel-based terrain scene. The right-eye image is obtained by exploiting the frame coherence between the two views. Most of the pixel values are directly obtained from the left image by reprojection. The remaining pixels are computed by ray casting, while further accelerated with ray coherence. An A-buffer is employed to reduce image error caused by reprojection to non-integer pixel locations. Image-based task partitioning schemes are explored to effectively parallelize our algorithm on a multiprocessor.

Keywords: Terrain visualization, voxel-based modeling, ray casting, parallel rendering, virtual environment, visual flythrough, stereoscopic rendering, antialiasing.

1 Introduction

Stereoscopic displays play an important role in scientific visualization applications and virtual reality environments. In such systems we often generate two images of a scene which differ in their horizontal positions by using stereoscopic rendering techniques, and display them with stereoscopic displays. As a result, the observer sees a merged image with binocular parallax that appears truly 3D. Compared to the conventional display mode employing single-image cues, stereoscopy provides unambiguous depth information with greater robustness [1].

Although stereoscopic rendering provides impressive true 3D visual effects, its popularity is often limited by its slow rendering rates when compared to those of a single image rendering. In the worst case, the rendering time is doubled. Therefore, fast stereoscopic rendering approaches have been proposed which exploit the coherence between the two views, so that the second image of the stereo image pair can be generated in a fraction of the time of the first image. Most of the previous work was based on traditional geometric rendering schemes, such as ray tracing, where Adelson and Hodges computed the second view with as little as 5% of the computation time required to fully ray trace the first view [2]. With the increasing interest in volumetric data and volume rendering techniques, a fast stereo volume rendering technique for ray casting was proposed by Adelson and Hansen [3] and further accelerated by He and Kaufman [4]. However, both papers assumed parallel projection. Our work focuses on the more complicated perspective stereo projection. In particular, we are interested in interactive stereoscopic rendering of voxel-based terrain, which is critical to our virtual flythrough system.

In general, the topographical and textural features of the terrain are obtained from a 2D elevation map and a corresponding color aerial or satellite photograph. Two kinds of terrain modeling have been adopted for different terrain rendering algorithms: surface-based modeling [5], which uses a set of tiny triangles to cover the elevation grid, and voxel-based modeling [6, 7, 8], where terrain is represented as a view-independent volume buffer of volume elements (in short, *voxels*) above a *base plane* or sea plane with an elevation value of 0. In our virtual flythrough system, we have chosen the voxel-based approach for terrain modeling and rendering due to several considerations. First, the format of the elevation map lends itself to generating a very high resolution 3D volume of terrain and multiresolution volumes. Also, texture mapping for voxels is much simpler, higher quality, and can be preprocessed. More importantly, the voxel-based model is somewhat scene complexity independent, and it is easy to incorporate clouds, haze, flames, and

other amorphous phenomena and volumetric objects [9].

In our previous work, we developed a fast single image rendering method for voxel-based terrain [8]. The ray casting method has shown several attractive features supporting fast terrain rendering:

1. Early ray termination: This is performed by terminating ray sampling along each ray once the closest terrain surface is hit, since the terrain surface is opaque.

2. Ray coherence: If two rays cast from the viewpoint are projected onto the same line on the base plane of the terrain, the higher ray hits the terrain surface at a position farther away from the view point [7]. Ray casting can be dramatically accelerated by skipping most of the empty space above the terrain surface.

3. Scalability: As an image-order rendering approach, the ray casting algorithm can be effectively parallelized by using image-based task partitioning schemes. Its performance can be scalable with the number of processors when a load-balancing image partitioning scheme is employed.

As a result of these features, we reached interactive rendering rates at more than 10 frames per second for single terrain images [8]. However, such an encouraging result makes our fast stereoscopic rendering even more challenging. Therefore, we propose an interactive stereoscopic rendering approach, where most of the pixel values of the right image are effectively reprojected from the left image. Remaining pixel values are computed by performing additional ray casting. A perspective projection geometry is adopted so that the reprojection can be efficiently done in both images in the same scanline order. Ray coherence is exploited to accelerate the additional ray casting procedure. Different image-based task partitioning schemes are employed for further speedups.

Our stereoscopic rendering method is different from the previous stereo volume rendering approaches [3, 4]. We use a depth image from the left view of the *opaque* terrain, while the methods in [3, 4] recorded all intermediate sampling information along each ray from the left view to produce *transparent* objects. Furthermore, our work focuses on the more complicated perspective projection. By the way, although Adelson and Hasen [3] rendered on a parallel machine, they did not discuss specific load balancing solutions for stereo algorithms. In fact, our method is more similar to Adelson and Hodges' work on stereo ray-tracing [2], although they dealt with complex surface models rather than 3D volume data. In our study, we are more concerned with exploiting specific properties of the volume data and the terrain scene. For example, we process image scanlines from top down and process pixels in each scanline from right to left. The rules of visibility determination for reprojected points also differ.

2 Our Stereoscopic Rendering Algorithm

To generate a true 3D terrain scene viewed by two eyes with horizontal parallax, we first put the camera at the left eye to get the left image, by using our previously proposed perspective terrain rendering algorithm for single images [8]. Then, we shift the camera to the right eye position to generate the right image. Since the eyes are so close to each other, the frame-to-frame coherence between these two images can be exploited to save computation time. Specifically, when we generate the right image, instead of performing full ray casting as we did for the left image, we directly obtain most, if not all, of the pixel values from the left image. The basic steps of our serial stereoscopic rendering pipeline are as follows:

1. Generate the left image as a normal single-image and record the depth value for each pixel.

2. Establish a perspective stereoscopic projection geometry so that pixels in a scanline of the left image are reprojected to the same scanline in the right image.

3. Reproject each pixel in the left image to the right image in scanline order, and employ an A-buffer to alleviate the aliasing caused by reprojecting to non-integer pixel locations.

4. Remove invisible reprojections due to different fields of view or occlusion.

5. Fill the "holes" in the right image by additional ray casting which is accelerated by exploiting ray coherence.

The rest of this section describes more specific details of these steps. The parallelization of this serial stereoscopic rendering algorithm is described in the Section 3.

2.1 Generating the Left Image

The left image is generated by using our previously proposed terrain rendering algorithm [8], where a sequence of equidistant resamplings are taken along the ray cast from each pixel, until the ray hits the terrain surface or exits the volume. If the current resampling value is greater than the density threshold, we regard the ray as having reached terrain surface and thus stop the ray traversal. If a 2D color photograph is available as the terrain texture, it is pre-aligned with and pre-assigned to the terrain voxels. We can then easily calculate the terrain color at this sample point using bilinear interpolation from the related four neighbors

in the photo and map it back to the source pixel. Otherwise, we apply a local lighting model.

A key advantage of ray casting over other volume rendering techniques is that the ray coherence property can be efficiently exploited for acceleration. According to ray coherence, a higher ray always hits the terrain at a greater distance than that of the ray below it. Therefore, when the image columns are vertical to the horizontal direction, we can accelerate the traversal of a higher ray by space leaping according to the intersection information (or depth value) of the previous ray which was emanated from a lower neighboring pixel on the same image column.

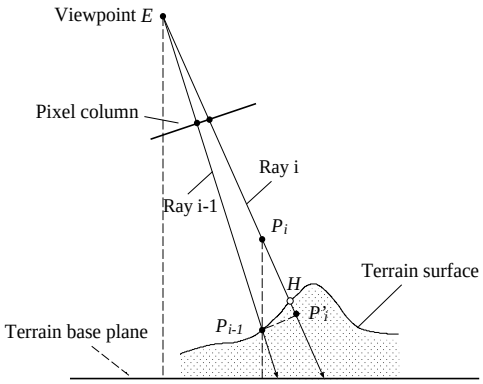

Figure 1. Terrain space leaping based on ray coherence.

Specifically, assume that ray i and ray $i-1$ are cast from two adjacent pixels in the same image column, and d is the distance from viewpoint E to stopping point P_{i-1} (hit point) on ray $i-1$ (see Figure 1). According to ray coherence, the traversal along the higher ray i can start from the sampling point P_i, which is located right above P_{i-1}. An alternative to P_i is P_i', which has the same distance d from the viewpoint. Choosing P_i' would be more efficient for skipping the empty space when the rays are more perpendicular to the base plane. However, P_i' may not be a conservative estimation for hit point H on ray i, as it may already pass H. Therefore, the traversal may need to retrace from P_i' to find the terrain surface.

2.2 Stereoscopic Projection Geometry

The fundamental technique of stereoscopic rendering is the establishment of *correspondence* – that is, the pairing up of pixels in the two images such that each pixel in a pair of points is the image of the same point in space. In our algorithm, we are more interested in finding the corresponding right image pixel for each non-background left image pixel. Figure 2 illustrates that there is no need to search the entire right image for the corresponding pixel P_r of an arbitrary pixel P_l in the left image. P_r is constrained to lie on the straight line L_r that is the projection of the line through P_l and the left center of projection E_l. Likewise, P_l is also constrained to be located on the straight line L_l that is the projection of the line through P_r and the right center of projection E_r. L_l and L_r are called the corresponding *epipolar lines* in the computer vision community. Also, the line connecting the two centers of projection is called the *baseline* and a plane through the baseline is termed a *epipolar plane*. Evidently, the epipolar plane passing through pixel P_l intersects the left and right images along epipolar lines L_l and L_r. Accordingly, any pixel on an epipolar line has its corresponding pixel on the corresponding epipolar line. In

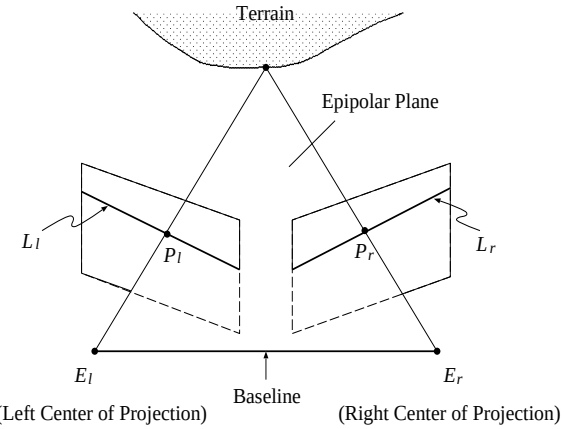

Figure 2. General case for stereo correspondence.

general, the epipolar lines in each image are different from the image scanlines. An intuitive proof is that the epipolar lines are not parallel to each other. Nevertheless, the two image planes can be chosen to be coplanar and parallel to their baseline; therefore, two corresponding epipolar lines L_l and L_r become collinear. Then, each pixel in a scanline of the left image can only be reprojected to the same scanline in the right image. As a result, the reprojection problem is simplified in 2D. The reprojection procedure from the left image to the right one can be efficiently conducted in a scanline order. Usually, since the two eyes are closer to each other than to the objects in space, the above rectification of image planes works fine. (The distance of the two projection centers E_l and E_r in Figure 2 has been magnified for legibility.)

Figure 3 illustrates the perspective projection geometry used in our stereoscopic rendering algorithm. E_l and E_r, separated by distance e, are located to the same side of the projection plane with the same distance h. We define the left-hand image space coordinate system as follows. Origin O is E_l. Axis Z is perpendicular to the image plane.

The main view vector or boresight of the left view is along axis Z. Axis X is along the horizontal direction pointing right, passing through E_l and E_r. Axis Y is orthogonal to both axes X and Z and points upward. By using this stereoscopic projection geometry, our single-image rendering algorithm can be employed to generate the left image without any modification. That is why we place the left center of projection at the origin and place the right center of projection to its right, instead of placing them symmetrically about the origin. The stereo images generated from these two kinds of projection geometry would be very similar, since the angly α between a pair of left and right rays is very small (often less than 2 degrees [4]).

2.3 Reprojecting Left Image Pixels

We assume that every pixel in the left image is the perspective projection of an object point on the terrain, and that each left image pixel has at most a single unique corresponding pixel in the right image. A left image pixel may not have its correspondence in the right image when the related object point is only visible to the left eye. Our assumption also excludes transparent objects from the scene. However, if a left image pixel does have a reprojection on the right image, the reprojection must be located in the same scanline in the right image according to our stereoscopic projection geometry. Therefore, our reprojection procedure is performed in a scanline order. For each scanline, the reprojection of left image pixels is conducted in the same XZ plane in image space. Thus, our reprojection computation only involves x and z coordinates, and we only care about x value of the reprojection for the right view.

Assume that the current pixel in question is at position (i, j) of the left image, and its 3D location in the image space is at $P_l(x_l, y_j, h)$ on the projection plane. We are interested in the corresponding pixel in the right image (n, j) of the same image scanline j, whose 3D image space coordinates are $P_r(x_r, y_j, h)$. These two pixels must be the images of an object point $P(x, y, z)$ in the image space, where x, y, z values can be obtained from the depth value of P_l in the left image (see Figure 3). Line segments AB and CD are the parts of the projection scanline respectively covered by the left and right images. P_l and P_r respectively fall into AB and CD. When the two eyes are close enough, AB and CD can overlap. From $(x_r - e)/(x - e) = h/z$, we get

$$x_r = e + h(x - e)/z \tag{1}$$

Then, the unknown n of the reprojected pixel position (n, j) in the right image can be easily computed from x_r:

$$n = (x_r - c)/w = (e + h(x - e)/z - c)/w \tag{2}$$

where c is X coordinate of the left-most pixel of the right image on the current scanline of the projection plane, and w is the physical width of one pixel on the projection plane. Both c and w are constants.

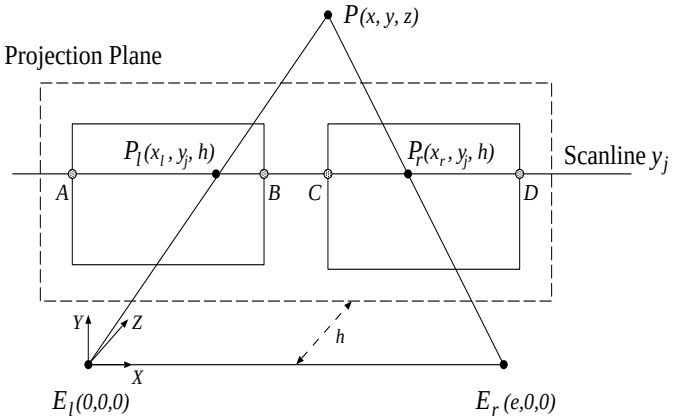

Figure 3. Stereoscopic perspective projection geometry (XZ plane).

In general, the value of n calculated from Equation 2 is non-integer. If we simply truncate or round it, serious aliasing may arise. P may be reprojected to a position in the right image that is off from its true reprojection by as much as near one pixel width. To fix this, rather than performing a time-consuming supersampling in the left view by casting multiple rays per pixel, we define an A-buffer in the right image. Unlike a common A-buffer where a 4×8 area mask is defined to represent the area of a pixel, a 1×8 mask is sufficient in our algorithm, since we are only concerned about the X coordinate of the reprojection. Thus, a pixel in the right image is represented by merely one byte, so that efficient bit operations are performed. Consequently, the maximum positioning error is reduced from nearly one pixel width to less than one eighth of that.

2.4 Hidden Reprojection Removal

An object point visible to the left eye is most probably also visible to the right eye. However, this is not always true for two reasons. First, the two eyes have different fields of view. Second, objects occlusion is viewpoint dependent. In order to generate a correct right view from the left view, we have to determine which reprojected left image pixels to utilize, as some of them might be hidden in the right view. On the other hand, there are some object points that are only visible to the right eye. Their projection on the right image can not be obtained from the left view. How to fill these "holes"(gaps) in the right image is discussed in Section 2.5. Here, we focus on a viable reprojection determination by a method we call *hidden reprojection removal*.

Figure 4 illustrates the different fields of view from the two eyes. Scene areas I and II are only visible to the left

eye. Scene areas IV and V are only visible to the right eye. Scene objects in area III are visible to both eyes.

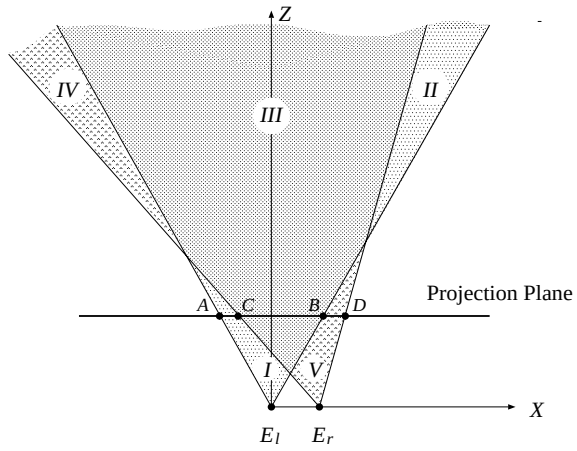

Figure 4. Different fields of view from two eyes.

If a pixel P_l in the left image is reprojected to a position outside the scope of $[C, D]$ on the current scanline, the related object point P in space must be located in either I or II, and invisible from the right eye. Therefore, the pixel value of P_l has no contribution to the final right image, and its reprojection is ignored. Otherwise, the reprojection of P_l is located between C and D and its related object point P is in area III. Although P is now in the field of view of the right eye, it may still not be viable in the right image, since it can be occluded by other scene objects located in front of P from the right view. These occluding objects are located either in area III or V. If in area V, we can cast rays from the right eye through the right image pixels between B and D on the scanline. The hitting test along these rays is performed only in that portion of area V that extends outward from the projection plane. If object points are found in this area, the reprojected pixels beyond them are ignored. Since this portion of area V is very small, the hitting test can be done very fast. In fact, this area is so tiny that the scene objects have a very low chance of appearing there. For the simplicity of description, we focus on the case when the occluding objects are in area III. In particular, we are interested in the situation where the reprojection of the current left image pixel is occluded by the reprojection of another left image pixel which is closer to the right eye.

Since the right eye is positioned to the right of the left eye, the reprojection of the current pixel can only be occluded by the reprojections of those pixels to the right of the current pixel on the same image scanline. Accordingly, we process each pixel in the current left image scanline in an order from right to left. We only need to record the up-to-date left-most reprojection on the current scanline on the projection plane, and keep updating it when necessary. Specifically, assume that P_l is the current pixel in the left image, P_r is its reprojection to the right eye, P_l and P_r are the images of an object point P, and P_s is the left-most reprojection so far on the current scanline. There are two possibilities for the relation of P_r and P_s: P_r is either to the left of P_s or not. Normally, P_r is to the left of P_s, which means P is visible to both eyes (see Figure 5a). Therefore, we update P_s by P_r. However, sometimes P_r is located on or to the right of P_s, which means P might be occluded by some object points which are projected to the right of P_l in the left image, as shown in Figure 5b. In this case, we simply ignore P_r and keep the current P_s untouched. It is worth pointing out that even in this special case, P may not be blocked to the right eye if it happens to be viewed through some gap on the terrain contour as illustrated in Figure 5c. Since the terrain contour is mostly continuous, this special situation rarely occurs. Therefore, even if P is actually visible to the right eye, we still ignore it for the simplicity of the implementation. In the following section, we discuss how to fill such gaps in the right image to ensure a correct image.

2.5 Filling "Holes" for the Right Image

During the above reprojection procedure, once we have determined that the current reprojected left image pixel is visible to the right eye, we fill the gap between the current reprojection P_r and the current left-most reprojection P_s on the current right image scanline. There are three different situations depending on whether P_r is the left-most, right-most, or middle reprojection on the right image row. We still use C and D to respectively represent the left-end and right-end points of the current right image scanline.

1. Left end gap: first, we examine whether P_r is the left-most (first) viable reprojection on the current right image row. If it is, we cast rays through each image pixel between P_r and D to fill this left end gap, and set P_s to P_r.
2. Middle end gaps: if P_r is not the left-most reprojection, then P_s is available. We calculate the distance between P_r and P_s. If this distance is less than a user-defined gap threshold, we interpolate the colors of the pixels between P_r and P_s. Otherwise, we cast rays through these pixels to fill the gap between P_r and P_s. Then, assign P_r to P_s.
3. Right end gap: if P_r is already the right-most reprojection, we cast rays through the pixels between C and P_r to fill the right end gap.

Figure 6 presents a pseudo-code of our serial stereoscopic algorithm. Note that in our implementation, the reprojection and gap filling procedure are performed on the

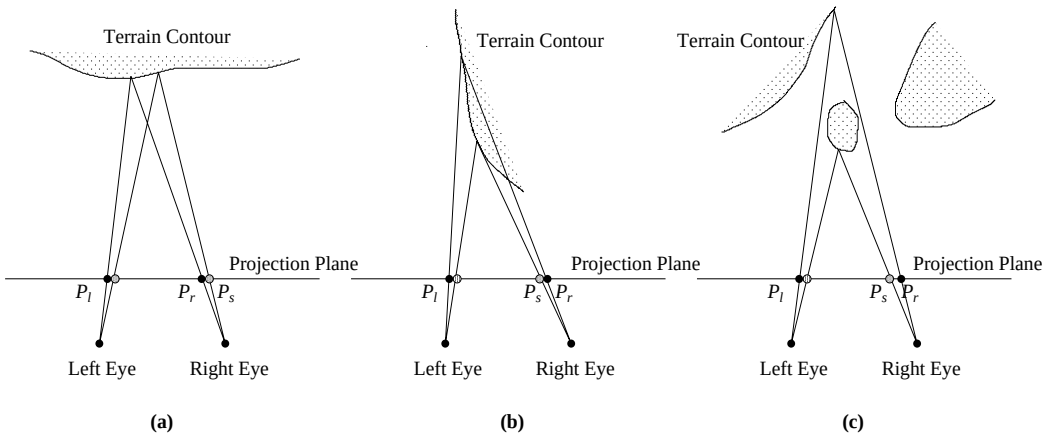

Figure 5. Hidden reprojection removal.

finer A-buffer with 1×8 area mask for each image pixel. Interpolation between two neighboring reprojections are conducted on this A-buffer. Image pixel colors are finally summed from the A-buffer. Yet, additional ray casting is directly performed on the right image to find the unknown pixels in the gaps. For the simplification of the algorithm description, we do not mention the A-buffer in the pseudo-code. In addition, we set the gap threshold to be two-pixel width. A larger threshold may lead to a faster speed, but may also lose some details between the two neighboring reprojections. In order to accelerate the additional ray casting procedure, we also make an effort to skip the empty space along each ray. A straightforward method to find the depth of a ray casting pixel seems to be interpolating the depth from those of the two neighboring reprojections. However, there are two problems that may arise. First, additional ray casting is usually performed where the two neighboring reprojections are so far away that the interpolated results (for colors and also for depth) may be unacceptable. Second, there is no guarantee that the depth of a ray cast pixel is always between those of its neighboring reprojections, except when the two neighbors are very close to each other. We realize that accelerating the additional ray casting procedure by exploiting ray coherence is still critical for our fast stereoscopic rendering algorithm. The solution is discussed in the next section, incorporated with our image-based task partitioning scheme.

3 Parallelism Acceleration

In our previous work, we parallelized the full ray casting algorithm for a single-image on a Silicon Graphics Power Challenge, a bus-based shared-memory MIMD (Multiple Instruction, Multiple Data) machine. We used a static image-based task partitioning strategy for parallelism. The image was treated as a pool of columns, and each processor selected and processed a fixed number of image columns in an interleaved order. Ray coherence was exploited to skip most of the empty space along a set of rays cast from the same image column. The total traversal distance along each set of rays cast from each image column was almost the same, so that each set of rays had approximately an equal amount of work to perform for ray traversal [8].

In order to maintain an interactive rate for the stereoscopic terrain rendering, we need to parallelize the generation of the right image. We would like to modify the serial stereoscopic rendering algorithm given before, by separating the additional ray casting operation from the reprojection and interpolation operations. Thus, the algorithm is separated into two parts. In the first part, pixels in the left image are reprojected to the right view in scanline order. Whenever ray casting is needed to fill a gap in the right image, we simply assign a special color to those pixels inside the gap, instead of casting rays through these pixels right away. We call these *ray-casting pixels*. When all pixels in the left image have been processed, we enter the second part where ray casting is performed on those pixels having a special color in the right image. The first part is called the reprojection procedure, while the second part is called the additional ray casting procedure.

3.1 Parallelization of Reprojection

In the reprojection procedure, the pixels in the left image row are reprojected to the right image and a linear interpolation is conducted if the distance between two neighboring reprojections is no more than a user-defined threshold. Since this threshold is often quite small (about two-pixel width), the interpolation computation between the two neighbors does not change much. Accordingly, the amount of computation work on each scanline is approximately proportional to the number of pixels in the left image row.

```
//assume points are represented in 3D image space.
for(each row in left image){
   //end points on right image row
   compute C and D
   START = FALSE;
   END = FALSE;
   //right to left
   for (each non-background left image pixel Pl){
      calculate the reprojection Pr of Pl;
      //discard an object point in area II
      if (Pr >= D)      //to the right of D?
         continue;      //yes, discard
      if (Pr < C)
         END = TRUE;    //discard points in area I
      if (START==FALSE){
         cast rays between [Pr, D]; //left-end gap
         Ps = Pr;
         START = TRUE;
         continue;
      }
      if (END==TRUE){
         cast rays between [C, Ps]; //right-end gap
         break;         //go to the next row
      }
      distance = Ps - Pr;
      if (distance <= 0)
         continue;      //discard occluded points
      if (distance <= GAP_THRESHOLD)
         interpolate pixel colors between [Pr, Ps];
      else
         cast rays between [Pr, Ps];//middle gaps
      Ps = Pr;
   }
}
```

Figure 6. Serial algorithm for stereoscopic terrain rendering.

Therefore, we can treat the left image as a pool of rows, and assign a fixed number of image rows to each processor in an interleaved order.

Although we do not perform additional ray casting in this reprojection procedure right now, we need to do some preparation beyond assigning a specific color to each ray-casting pixel. As mentioned, we can not find the depth information for each ray-casting pixel from its neighboring reprojections on the same scanline. Fortunately, in the subsequent additional ray casting procedure, we can exploit ray coherence for all the ray-casting pixels in the same image column. Thus, we can skip most of the empty space along a ray, according to the depth of the ray-casting pixel in the same column below it. But how about the lowest ray-casting pixel P_w in an image column? There are two different situations. If P_w is the bottom pixel of the column, we have to traverse along the ray from the beginning as we did in the left view. Otherwise, P_w has a neighbor P_n below it in the column, whose color is obtained by interpolation rather than ray casting (see Figure 7). Since interpolation is performed on P_n, the two neighboring reprojections of P_n must be very close. Therefore, we can safely find the depth of P_n by interpolation from the depths of its two neighboring reprojections, and use it to accelerate the ray casting of P_w.

In our implementation, we establish a 1D array for each processor, called a *depth buffer*. Each element of the depth buffer corresponds to one column of the right image, with several components to record the position of the lowest ray casting pixel P_w, the depth of its lower neighbor P_n (if exists), and the number of ray-casting pixels in that image column. Note that the computation of the depth value for an interpolated pixel is necessary only when its upper neighbor is a ray-casting pixel. Therefore, we constrain the reprojection procedure in a scanline order top down, so that we can compute depth values only when necessary. After all processors have completed their reprojection work, their depth buffers are combined into one, and only the lowest ray-casting pixel survives for each column. The number of ray-casting pixels from the same column are summed up. This combination requires synchronization, but no inter-processor communication due to the shared-memory.

3.2 Parallelization of Additional Ray Casting

To exploit ray coherence for ray-casting pixels in the same column, we treat the right image as a pool of columns when we parallelize the additional ray casting procedure. However, unlike the generation of a full ray casting image, the amount of computation work differs dramatically among different columns, due to the different number of ray-casting pixels in each column. A static interleaved partitioning scheme, such as the one we used to generate the left full ray casting image, can not guarantee a good load balance. Instead, a dynamic partitioning scheme is more appropriate. First, a task queue is created which contains those right image columns possessing at least one ray-casting pixel. Then, all the image columns in the queue are sorted in an order of decreasing number of ray-casting pixels. After

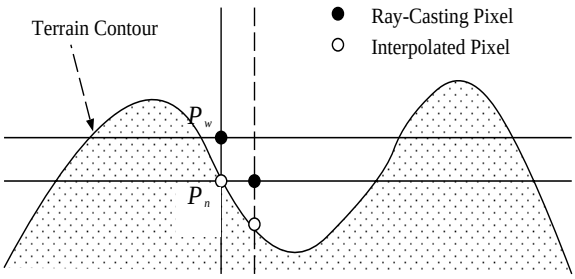

Figure 7. Depth Information from the lower interpolated pixel.

that, each processor takes and processes one column from the head of the task queue. Once a processor completes its column, it takes another one from the head of the queue until all columns are exhausted.

For each column, additional ray casting is completed in the following steps. First, find the location of the lowest ray-casting pixel and the corresponding depth from the depth buffer generated in the previous reprojection procedure. Cast a ray from this pixel and use the corresponding depth to skip the empty space along the ray. When the hit position is found along the ray, save the new depth value in the depth buffer and update the old one. Second, move upward along the column to the next ray-casting pixel with the specific color and cast a ray through it. The updated depth value is used to skip the empty space along the ray; it is updated again once the hit point is found. Then, repeat the second step, until no ray-casting pixels are left or the current ray does not intersect with the terrain surface. Finally, search upward along the column for all remaining ray-casting pixels, and directly assign their colors to be background color. No ray casting is performed through these pixels, since they have no chance intersecting with the terrain surface.

4 Implementation and Experimental Results

We have implemented our interactive stereoscopic terrain rendering algorithm as the core of our virtual flythrough system on both SGI graphics workstations and a virtual reality environment using a Responsive Workbench. The Responsive Workbench provides a powerful tool by immersing the user within the computer-generated virtual navigation environment with a superior 3D interaction. The user can conveniently control the viewing position and orientation by moving and rotating the Ascension tracker with six degrees of freedom. Arbitrary stereoscopic perspective views over the terrain are generated using our interactive stereoscopic algorithm. These views are displayed on-the-fly by shutter glasses for several people working collaboratively, with both immediate visual feedback and high-definition photo-realistic images.

To demonstrate the performance of our stereoscopic algorithm, we have tested several terrain data sets on an SGI Power Challenge with 16 MIPS R10000 processors (194MHz) and 4 GB RAM. Figure 8a and 8b show a pair of stereoscopic images of a terrain in Southern California, generated by our algorithm. Each image size is 500×400. Our terrain model consists of a 3D terrain volume with a resolution of $512 \times 512 \times 64$ and a corresponding registered aerial photo. In Figure 8, instead of mapping the color photo to the terrain, we used a lighting model. Our system provides such a rendering option, considering that a color photo may not always be available for terrain data sets. Table 1 presents measured rendering times for both the left and right images of Figures 8a and 8b generated by our algorithm with different number of processors. Near linear scalability is shown as the number of processors increases, which is ascribed to our effective task partitioning schemes.

Table 1. *Stereoscopic rendering times (in sec) of a terrain in Southern California.*

processors	1	4	8	12	16
Left image	2.92	0.79	0.41	0.30	0.21
Right image	0.28	0.08	0.04	0.03	0.02

The speedup ratio between the left and right images is affected by several factors, such as how many non-background pixels we work on, and how many pixels are computed by the additional ray casting. For an arbitrary view as shown in Figure 8, the average time saving of the right image is about 90% of the left image for a different numbers of processors. In Figure 8c, we marked with red colors those pixels in Figure 8b which were rendered by ray casting. The remaining non-background pixels were generated by interpolation. The user-defined gap threshold was set to a two-pixel width. The ratio between the number of ray-casting pixels and the interpolated ones is 4.3%. However, since the amount of computation for ray casting is much more expensive than the reprojection and interpolation operations, the additional ray casting time takes about one fourth of the entire rendering time for the right image. Thus we believe that speedup for the additional ray casting procedure is very important and have made efforts to accelerate it by exploiting ray coherence. We measured the additional ray casting time for Figure 8b without considering ray coherence, and found it increased by about a factor of four.

In order to show the accuracy of the right image in Figure 8b, we rendered a full ray casting image from the same right view in Figure 8d and compared these two images pixel by pixel. Figure 8e shows the differences of pixel colors between Figure 8b and 8d. The differences are measured as Euclidean distances in RGB ($256 \times 256 \times 256$) space. The average difference is 3.1%. (The intensities shown in Figure 8e were magnified from scope 1–50 to 1–255.) The error is small because we employed A-buffer for antialiasing. When we further experimented with integer truncation without using an A-buffer, the error increased. The difference map is shown in Figure 8f, where the average difference increased to 5.7%. By the way, we did not find noticeable increase of the rendering time when A-buffer was used. This is because the numbers of both reprojected pixels and the ray-casting pixels did not change, which dominated the rendering time. This result proved the effectiveness of the A-buffer.

Figure 9 gives another stereoscopic view of the same ter-

rain data set as for Figure 8. This time we rendered the terrain with texture obtained from the registered pre-mapped color photo. Table 2 presents the rendering times for both the left and right images generated by our algorithm with different numbers of processors. The average time saving is about 72%, which is less than the 90% shown in Table 1 — bilinear interpolations for texture mapping is much less time-consuming than lighting computation [8]. Although the time saving for the right image decreased, the total rendering speed for the stereoscopic image pair increased. As a result, we reached a stereoscopic perspective rendering rate at more than $10Hz$ on 16 processors. Similar results have been shown on other terrain data sets.

Table 2. *Stereoscopic rendering times (in sec) of a terrain (with texture).*

processors	1	4	8	12	16
Left image	0.93	0.26	0.14	0.10	0.07
Right image	0.25	0.07	0.04	0.03	0.02

5 Conclusions and Future Work

We have presented an interactive stereoscopic rendering algorithm of voxel-based terrain with natural perspective projection. It provides unambiguous depth information of a terrain scene during navigation in our virtual flythrough system. The generation of the right half of the stereo image pair is vastly accelerated by exploiting the frame coherence between the two views. Most of its pixel values are directly obtained from the left image by reprojection. A small number of rays are cast through the remaining pixels to fill the gaps, and further accelerated by exploiting ray coherence. High image quality is not only ascribed to the accurate additional ray casting, but also to the antialiasing A-buffer which reduces the image error caused by the non-integer reprojection problem. In addition, load balancing task partitioning schemes lead to good speedups when the algorithm is run on a multiprocessor, and interactive stereoscopic rendering rates have been reached.

Our current work includes multiresolution rendering, representation and manipulation of other 3D volumetric objects, such as stationary buildings, trees, clouds, and moving planes and vehicles on top of the terrain. We are in the process of combining these 3D voxelized objects with the terrain to generate richer and more practical stereoscopic perspective views in the context of a scene graph.

Acknowledgments

This work has been partially supported by NRL grant N00014961G015, NSF grant MIP9527694, ONR grant N000149710402, and NASA grant NCC25231. Thanks to Edmond Prakash, Lichan Hong, Suya You, Baoquan Chen, Milos Sramek, Kenneth Gordon, Bai Wang, Silvia Piquet, and other colleagues in the virtual flythrough project.

References

[1] L. Hodges (1992). "Tutorial: Time-Multiplexed Stereoscopic Computer Graphics". *IEEE Computer Graphics and Applications*, Vol. 12, No. 2, March 1992, 20–30.

[2] S. Adelson and L. Hodges(1993). "Stereoscopic Ray-Tracing". *The Visual Computer*, Vol. 10, No. 3, December 1993, 127–144.

[3] S. Adelson and C. Hansen (1994). "Fast Stereoscopic Images with Ray-Traced Volume Rendering". *1994 Symposium on Volume Visualization*, October 1994, 3–9.

[4] T. He and A. Kaufman (1996). "Fast Stereo Volume Rendering". *Proc. IEEE Visualization '96*, October 1996, 49–56.

[5] P. Lindstrom, D. Koller, W. Ribarsky, L. Hodges, N. Faust, and G. Turner (1996). "Real-Time, Continuous Level of Detail Rendering of Height Fields". *Proc. SIGGRAPH' 96*, August 1996, 109–118.

[6] J. Wright and J. Hsieh (1992). "A Voxel-Based, Forward Projection Algorithm for Rendering Surface and Volumetric Data". *Proc. IEEE Visualization '92*, October 1992, 340–348.

[7] D. Cohen-Or, E. Rich, U. Lerner, and V. Shenkar (1996). "A Real-Time Photo-Realistic Visual Flythrough". *IEEE Transactions on Visualization and Computer Graphics*, Vol. 2, No. 3, September 1996, 255-265.

[8] M. Wan, H. Qu, and A. Kaufman (1998). "Virtual Flythrough over a Voxel-Based Terrain". *Proc. IEEE Virtual Reality '99*, March 1999, 53-60.

[9] A. Kaufman, D. Cohen and R. Yagel (1993). "Volume Graphics". *Computer*, Vol. 26, No. 7, July 1993, 51–64.

[10] M. Levoy (1988). "Display of Surface from Volume Data". *IEEE Computer Graphics and Applications*, Vol. 8, No. 5, May 1988, 29-37.

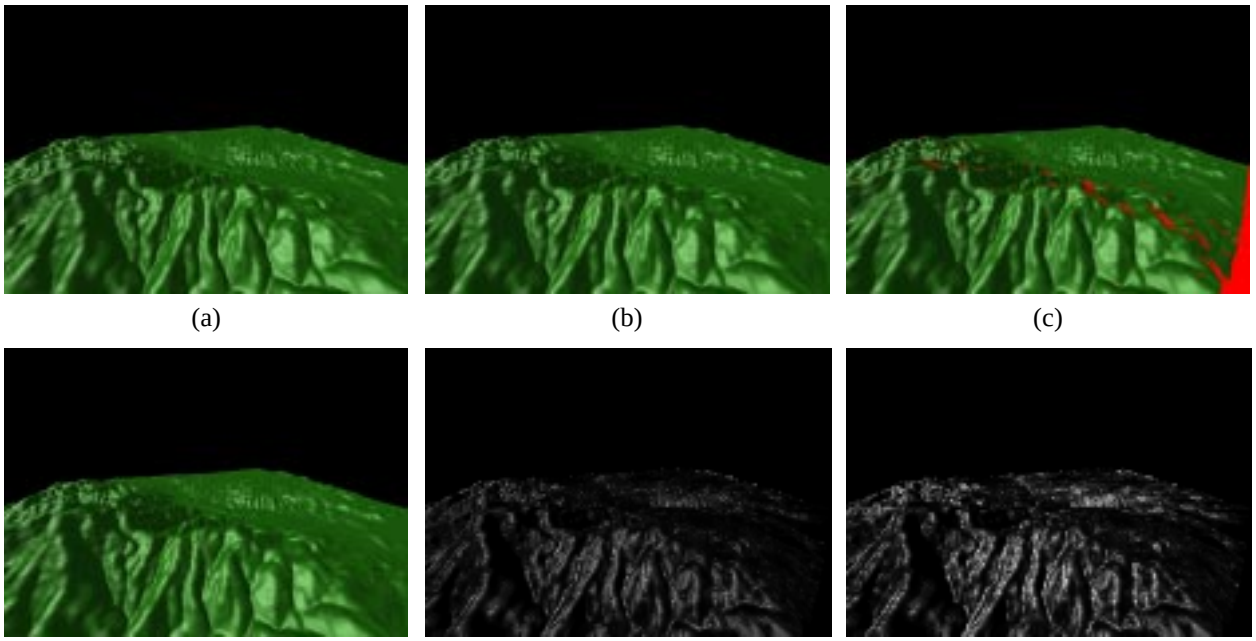

Figure 8. Stereoscopic views of shaded terrain and difference maps. *(a) Left image by full ray-casting. (b) Right image by our method. (c) Right image with ray-casting pixels shown in red. (d) Right image by full ray-casting. (e) Intensity-magnified difference map of (b) and (d). (f) Intensity-magnified difference map of (b) and (d) without A-buffer.*

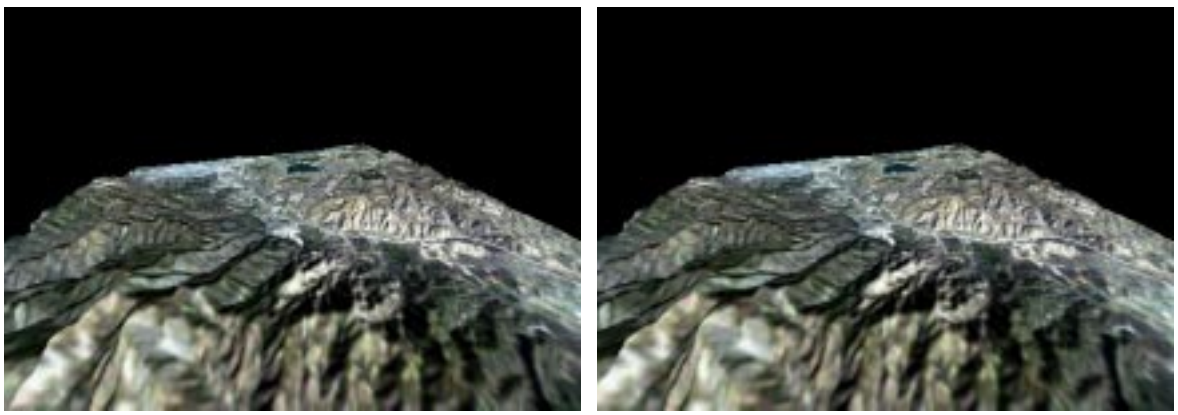

Figure 9. A stereo pair from a flythrough of texture-mapped terrain on the Responsive Workbench. *(a) Left image by full ray-casting. (b) Right image by our method.*