A Real-Time Photo-Realistic Visual Flythrough

Daniel Cohen-Or[1,2], Eran Rich[1], Uri Lerner[1], and Victor Shenkar[1]

[1]Tiltan System Engineering
Bnei-Brak 51204, Israel

[2]School of Mathematical Sciences
Tel-Aviv University, Ramat-Aviv 69978, Israel

Abstract

In this paper we present a comprehensive flythrough system which generates photo-realistic images in true real-time. The high performance is due to an innovative rendering algorithm based on a discrete ray casting approach, accelerated by ray coherence and multiresolution traversal. The terrain as well as the 3D objects are represented by a textured mapped voxel-based model. The system is based on a pure software algorithm and is thus portable. It was first implemented on a workstation and then ported to a general-purpose parallel architecture to achieve real-time performance.

Keywords: Terrain Visualization, Parallel Rendering, Flight Simulator, Visual Simulations, Voxel-Based Modeling, Ray Casting

1 Introduction

The quest for real-time photo-realistic rendering has been one of the major goals of 3D computer graphics in recent years. Techniques for adding realism to the image, such as shading, shadow, textures, and transparency have been developed. The generation of realistic images in real-time is currently being researched. Flight simulator applications have always led the way in real-time performance [3]. Special-purpose machines, dedicated to flight simulation have been developed. These machines generate images with reasonable realism in real-time, but are expensive (more than a few million US dollars) [7, 15]. The main contribution of the work presented in this paper is that the real-time performance was achieved on commercial general-purpose parallel architecture, as opposed to specialized rendering hardware.

Generating images of arbitrary complex scenes is not within the reach of current technology. However, the rate of image generation in flight simulation can achieve real-time because the scenes that are viewed from the sky are not too complex. Typical views contain terrains which are merely 2.5D, or 3D objects which are seen as relatively simple, featureless objects. Nevertheless, simulating photo-realistic aerial views in real-time is by no means easy [10, 8, 12, 13].

The term *visual flythrough* can be distinguished from flight simulation. Visual flythrough generates simulated images as seen from a video camera attached to a flying object. The camera generates *photo-realistic* images, although not necessarily in color, since many video cameras have a grey-level output. It should be emphasized that the generation of true photo-realistic images is critical for applications where the user needs to recognize the area or identify objects on the ground (i.e. targeting, mission rehearsal). See the photo-realistic impression of the images presented in Figure 1.

In a typical flythrough scenario the camera views a very large area, especially when the camera pitch angle is high (i.e. towards the horizon). In many applications the camera flies at high speed over long distances and the area covered during a few seconds of flight is vast. This suggests that it is not possible to load the entire terrain data onto the main memory. For some applications even a Gigabyte of RAM is not enough. It is safe to say that no size will ever suffice, since the application demands will always increase according to the availability of space. This suggests that flythroughs require a large secondary storage together with a fast paging mechanism.

An image of an aerial view gains its realistic impression by mapping a digital photograph onto the terrain model. In order to achieve high quality, full resolution of both the digital terrain model and the corresponding aerial photograph need to be employed. This causes a major load on conventional graphics hardware based on a geometric pipeline. First, a high resolution polygonal terrain model contains a vast number of polygons which need to be processed by the geometric pipeline, while processing tiny polygons loses the cost-effectiveness of the rasterization hardware. The high resolution photograph that needs to be texture-mapped during rasterization, creates a further problem since large photographic maps need to be loaded onto an expensive cache. For example, the Reality-Engine board cannot hold more than a few Kilobytes of texture [1], while larger textures need to be paged from the main memory in real-time [8]. To avoid that, many flight simulators use repetitive patterns as ground texture, but for some applications where a specific target area is a vital requirement, a true photograph has to be mapped on the terrain. These photographs are huge and must be loaded on the fly to the rasterization hardware, forming a serious bottleneck in the rendering pipeline.

Instead of using a polygonal model and a geometric pipeline we have favored a software solution where the model is represented by a voxel-based representation. The texture-mapping of the photograph over the model is a preprocessing stage, which is decoupled from the rendering stage [9]. Voxel-based modeling also lends itself to representing fine grained geometry. The voxel data is regular, internally represented in an array, and offers easy and fast access to the data [5]. Each voxel represents and stores some discrete local attributes of the model. Voxels representing the terrain contain a height value and a color value, while the voxels representing a 3D model contain a texture photograph as will be described below in Section 5. A voxel-based visual flight simulator with real-time performance has been developed at Hughes Training, Inc. Their flight simulator runs on special purpose hardware, but yields poor results on a graphics workstation [15].

The visual flythrough that we have developed is hardware independent, and thus portable. Portability is important, since it enables integration of the flythrough system with rapid progress in hardware platforms. However, a software rendering algorithm must be fast enough, around a second or two per frame running on a sequential machine, so that on a parallel machine with 32 processors it achieves a rate higher than 20 frames per second. That

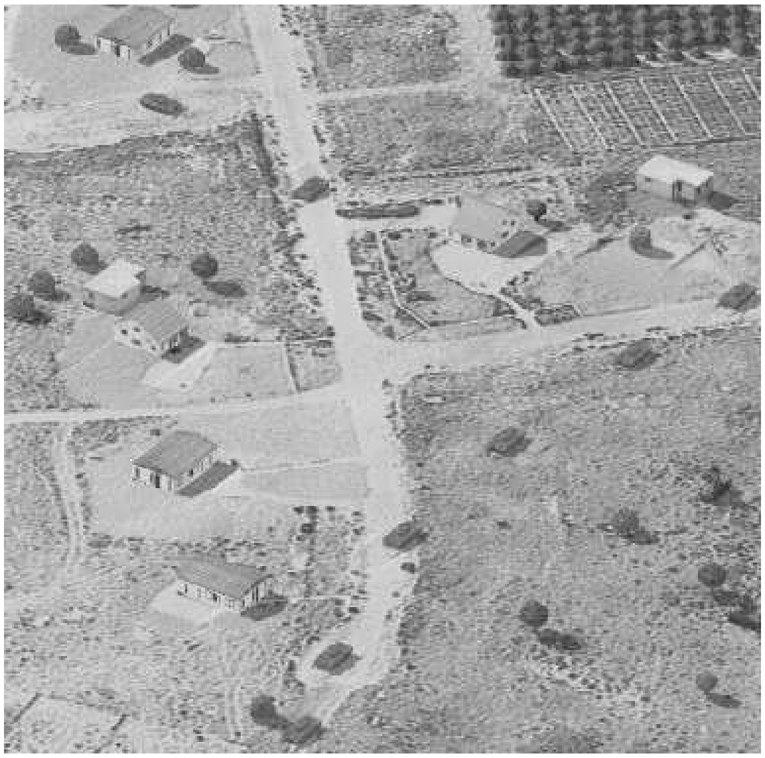

Figure 1: *Two aerial photo-realistic images generated by the flythough.*

is, of course, assuming little overhead is imposed on the parallel version of the algorithm.

Although we have employed a parallel machine, the real time performance is mainly due to an innovative rendering algorithm. The new algorithm generates a photo-realistic image such as in Figure 1 within two seconds, on a common workstation. The implementation of a parallel version of the algorithm on a 32-way multiprocessor architecture has sped up the rendering to achieve the desired real-time rates. It should be noted that other (hardware independent) ray casting algorithms have reached reasonable speeds ([13, 5, 11]), but just for point sampling. Avoiding aliasing artifacts is quite involved and time costly. The algorithm presented here resembles the principles of the projection algorithm in [15]. However, their algorithm is based on a forward mapping method and was designed to be implemented in hardware. The algorithm presented in the next section is a simple ray casting

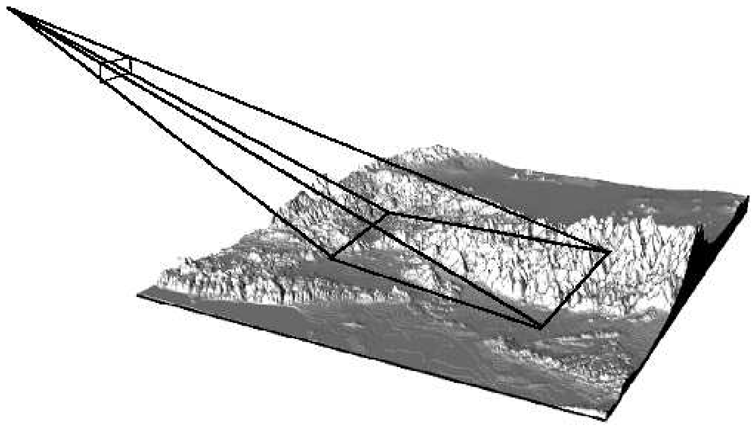

Figure 2: *The image footprint over the terrain is defined by the viewing parameters.*

(forward mapping) accelerated by ray coherence and multiresolution traversal and highly optimized for hardware independent implementation.

The remainder of this paper is structured as follows: Section 2 describes the rendering algorithm. Section 3 presents the IBM Power Visualization System, the current parallel platform of the parallel algorithm, implementation details concerning the parallelization of the algorithm, and some results. We discuss the generation of voxel-based objects in Section 5 and conclude with a brief discussion on our current activity and some final remarks.

2 The Rendering Algorithm

The sequence of images generated by the rendering algorithm is independent. Each image is defined by the location of the camera in 3D space, the camera field of view, and its orientation, namely the pitch, roll, yaw angles, and image resolution. Figure 2 depicts the *image footprint* defined by the projection of the image frame over the terrain. The terrain model is represented by a voxel-based map, generated from a discrete elevation map colored by a corresponding aerial photo map. The rendering algorithm is based on a backward mapping approach known as ray casting. The image is generated by casting a ray-of-sight emanating from the viewpoint through each of the image pixels towards the model (see Figure 3). The ray traverses above the

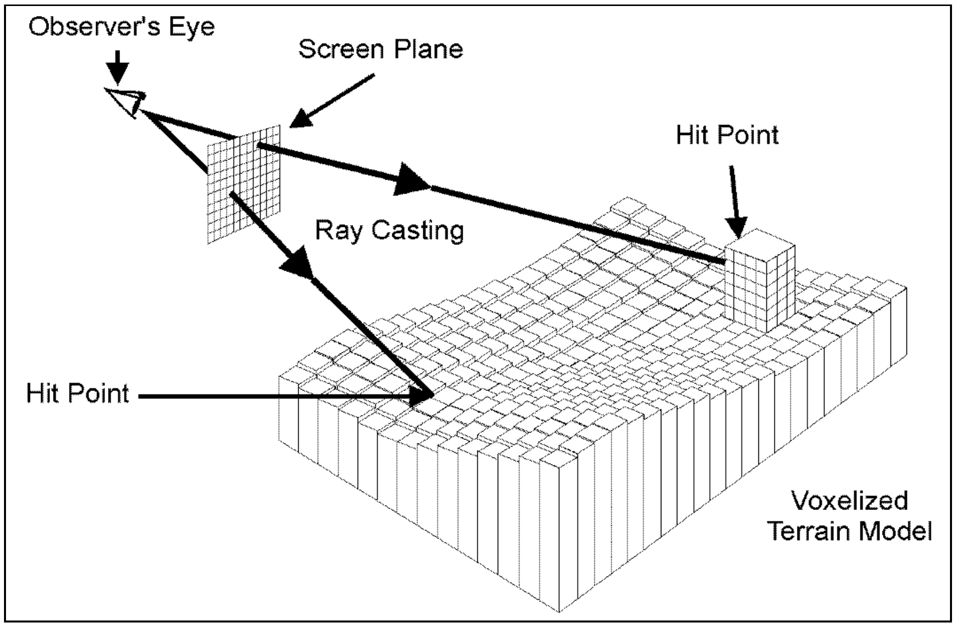

Figure 3: *Discrete ray casting of a voxel-based terrain*

terrain voxels until it intersects the terrain. The terrain color is sampled and mapped back to the source pixel. Since the model is discrete there is no explicit intersection calculation, but a sequential search for a "hit" between the ray and a voxel. The speed of the ray traversal is crucial for achieving real-time performance.

The technique we employ is based on a discrete grid traversal, where the steps along the ray are performed on the projection of the ray on the plane rather than in 3D. The heights along the ray are incrementally and uniformly sampled and compared to the height of the terrain below it, until a hit occurs and the color of the terrain at the hit point is mapped back to the source pixel. If there is no hit, then the background color of the sky is mapped. This apparently naive traversal is "flat" ([12]) in contrast to a "hierarchical" traversal ([5]). In [5] a pyramidal elevation map is used. The multiresolution pyramid is treated as a hierarchy of bounding boxes through which the ray traverses in a recursive top-down traversal. The number of steps of the hierarchical traversal is proportional to the height of the pyramid. When a binary pyramid is used, the number of steps is logarithmic to the terrain length, rather than linear to the terrain size as in the case of the flat traversal.

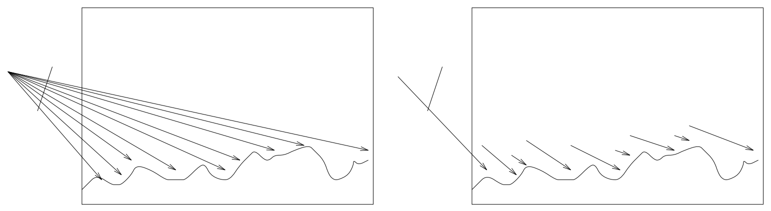

Figure 4: *Assuming the terrain has no caves, each ray can emanate from the previous hit point.*

Our algorithm is based on the incremental "flat" traversal, but, as will be shown, some rays are "hierarchically" traversed.

Since the terrain is a height field map we can assume that the terrain model has no vertical cavities or "overhangs" (i.e., a vertical line has only one intersection with the terrain). The traversal can be accelerated using ray coherence [6, 11]. The basic idea is that as long as the camera does not roll, a ray cast from a pixel vertically adjacent always hits the terrain at a greater distance from the viewpoint than that of the ray below it. The image pixels are generated column by column from bottom to top. A ray $i+1$ emanating above ray i will always traverse a distance not shorter than the distance of ray i (see Figure 4). Thus, ray $i+1$ can start its traversal from a distance equal to the range of the previous hit of ray i. This feature shortens the ray's traversal considerably.

The total number of steps required to generate one column is equal to the length of the column footprint, eliminating the factor of the number of pixel columns. In other words, a naive generation of one column has a time complexity of $O(ml)$, where l is the length of the column footprint and m is the number of pixels in the image column. Using ray coherence the time complexity is reduced to $O(l)$ only, providing an order of magnitude speed-up. The rays emanating from the bottom of the column cannot gain from a previous hit and are thus accelerated by a hierarchical traversal [5].

Using the above vertical ray coherence between consecutive rays, each terrain voxel is virtually traversed once. The time complexity of the traversal is proportional to the number of voxels in the image footprint. This is still a huge number since the image footprint can extend to the horizon. Moreover, this number is view-dependent and causes instability of the frame generation rate.

Due to perspective projection, the rays diverge with the distance, caus-

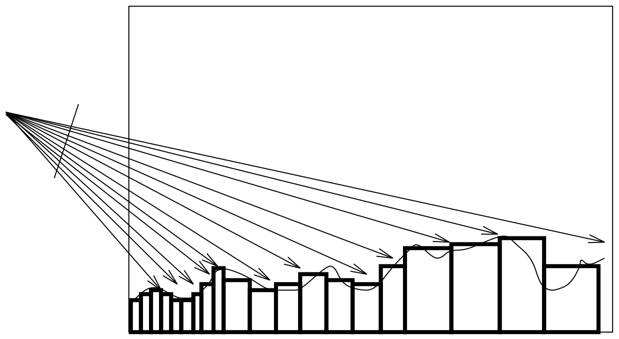

Figure 5: *Multiresolution traversal. The voxel map resolution corresponds to the sampling rate.*

ing a non-uniform sampling rate of the terrain voxels by the rays. The rays emanating from the bottom of the image frame hit the terrain at a closer range than the upper rays. Assuming that the terrain dataset is represented in a single resolution, then, close voxels tend to be oversampled while far voxels are undersampled. Using a hierarchy of data resolutions improves the sampling, since the rays can adaptively traverse and sample voxels of an appropriate size, proportional to the pixel footprint (see Figure 5). Optimally, in every step one pixel is generated. In multiresolution traversal the voxel sampling rate becomes proportional to the number of rays (i.e. pixels), and the number of steps becomes independent of the viewing direction. That is, the number of steps over the terrain is in the order of the image space rather than the object space. Thus, the adaptive hierarchical traversal not only speeds up the rendering, but also helps to stabilize the frame generation rate.

In our implementation, we use a pyramid of data resolutions where each level has half of the resolution of the level below it. Using more resolutions can be even more successful, in the sense of uniformity of the sampling, but then it would use more space. Another advantage of a binary pyramid is the simplicity of alternating between consecutive levels, where the step sizes are either multiplied or divided by two, taking advantage of the integer arithmetic of the traversal [5]. Moreover, the pyramid offers a fast first hit for the first rays which emanate from the bottom row of the pixel array. Those rays cannot benefit from coherency with the previous rays. For those rays a top-down traversal of the hierarchy speeds up their first hit [5].

One important issue that must be taken care of in a real-time hierarchical rendering is creating a soft transition when switching between levels. A sharp transition is very noticeable and causes an aliasing effect of a wave that sweeps over the terrain. A simple solution is to interpolate between adjacent hierarchies [14] where the interpolation weights are defined by the distance from the viewpoint to the sampled voxels. Since the range gradually changes, so do the weights, causing a soft transition.

Synthetic objects such as trees, buildings, and vehicles can be placed over the terrain. The 3D objects are represented by *sticks* (a run of voxels) of three types: *uniform* sticks which are colored by a single color like a terrain voxel, *textured* sticks which contain a vertical sequence of colored voxels, and *complex* sticks which are textured sticks, but contain some semi-transparent or fully transparent voxels (see [15]). Synthetic objects are then described by a set of adjacent sticks. A ray which hits a textured stick climbs onto the stick and maps back the stick texture to the screen. When a semi-transparent value is encountered, a secondary ray continues through the voxel. The results of the secondary ray are then blended with the values of the primary ray according to the value of the semi-transparent voxels. In many cases the transparency value indicates a cavity in the stick; in this case no blending is performed and the colors of the secondary rays are directly mapped to the pixels.

Since cavities cause the spawning of secondary rays it is clear that they slow down the rendering process. One way to reduce cavities is to fill them up at coarse resolutions, assuming the cavities are small enough and their contribution to the final image is insignificant. One should note that in typical scenes only a small fraction of the sticks need to be complex. For example, when viewing woods only the trees at the boundary needs to be fully represented with their non convex parts, while most of the other trees are hidden and only their tops can be seen.

A typical scene contains many replicated objects placed at different locations and orientations. Thus, many sticks are common to many objects. A complex voxel contains a pointer instead of a color which points into a stick table. Each stick consists of a header and a sequence of values. The header contains several attributes like the stick type and the stick length.

2.1 The Basic Algorithm

In this section we present in detail the basic algorithm that generates a single column of the image. The algorithm is based on a fast traversal of the column footprint over the terrain. The voxels along the footprint are tested for visibility and the colors of the visible ones are sampled and mapped back to the image column. The pseudo-code is shown in Figure 7.

Let E be the location of the eye and P the location of a column pixel. The parametric equation of a ray emanating from E and passing through P is $v = P + t(P - E)$. Denote the ray direction $P - E$ by $Q = (Q.x, Q.y, Q.z)$. Then for a given x, the coordinates along the ray are explicitly given by:

$$z = P.z + (x - P.x)(Q.z/Q.x) \tag{1}$$

and

$$y = P.y + (x - P.x)(Q.y/Q.x). \tag{2}$$

Assuming the ray is X major, i.e. $Q.x > Q.y$, then the sequence of the voxel coordinates (x, y) along Q is generated by a forward differences evaluation of the line equation:

$$z_{i+1} = z_i + (Q.z/Q.x) \tag{3}$$

$$y_{i+1} = y_i + (Q.y/Q.x) \tag{4}$$

where $x_{i+1} = x_i + SIGN(\text{Q.x})$.

Using fixed point arithmetic the integral coordinate of y, denoted by $\lfloor y \rfloor$, is retrieved by a shift operation on the binary representation y, while the fraction part, $w = y - \lfloor y \rfloor$, is used for linear interpolation at the sampling point (see below). The hit between the ray Q_j and the terrain is detected by comparing $height(x, \lfloor y \rfloor)$, the height of the terrain above $(x, \lfloor y \rfloor)$, against z. If $z > height(x, \lfloor y \rfloor)$ then x, y and z are incrementally updated, otherwise a hit has been detected. The terrain color at $(x, \lfloor y \rfloor)$ is sampled and mapped to the pixel P_j, and the process proceeds to the next ray Q_{j+1} emanating from P_{j+1}.

Since the terrain is a height field, the ray Q_{j+1} does not hit the terrain before it reaches the hit point of Q_j. The algorithm continues to evaluate the sequence of the (x, y) coordinates, and their heights need to be compared to the height of ray Q_{j+1} (see Figure 6). The slope $(Q.z/Q.x)$ and the height

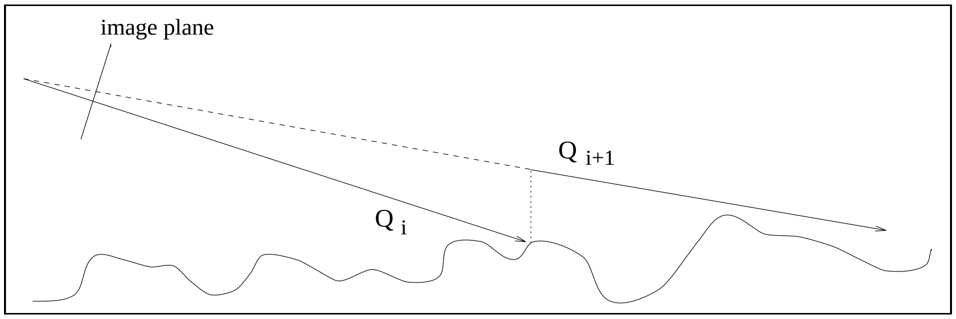

Figure 6: *Climbing from the hit point of ray Q_i to ray Q_{i+1}.*

of ray Q_{j+1} above x is evaluated by Equation 3. Note that a small error is introduced since the plane defined by the rays emanating from a column of the image plane is not perpendicular to the main plane and may be slightly slanted due to the perspective projection. However, when the field of view is small, say under 10 degrees, the error is insignificant.

Let $\vec{E}$ be the location of the eye.
Let $\vec{P}$ be the location of the bottom pixel of the column.
Let $\vec{Up}$ be the vector direction of the image columns.
Let $\vec{Q} = \vec{P} - \vec{E}$ be the direction of the ray emanating from P.
Assume $Q.x > Q.y$ and E is above the terrain, and $x = E.x$ and $y = E.y$.
Let n be the distance between x and the end of the terrain.

```
**while** (n - -){ // while not reaching end of terrain
        **while** ($z < height[x, \lfloor y \rfloor]$){ // test for a hit
                $w = y - \lfloor y \rfloor$; // yield the subvoxel weight
                Color = Sample($x, \lfloor y \rfloor, w$); // sample the voxels
                Pixel(j++) = Color; // back map the results
                **if** column done return;
                $\vec{P} + = \vec{Up}$; // move up to next pixel
                $\vec{Q} = \vec{P} - \vec{E}$; // climb to the new ray
                z = $P.z + (x - P.x) * Q.z/Q.x$;
                }
        // Move on to the next voxel along the ray
        x += SIGN(Q.x); // move along the major axes
        y += Q.y/Q.x; // incrementally update the Y coordinate
        z += Q.z/Q.x; // incrementally update the ray height
}
**if** (n) // the sky is seen
            color the rest of the pixels with the sky color;
```

Figure 7: *The integer base incremental traversal.*

The function Sample($x, \lfloor y \rfloor, w$) samples the terrain colors at the integer coordinates of x. However, the resolution of the fixed point values is higher than that of the voxel space, and the fraction value, denoted by w, yields the subvoxel location of the hit point. The exact hit point lies on the vertical grid line between $(x, \lfloor y \rfloor)$ and $(x, \lfloor y \rfloor + 1)$, (see Figure 8). Thus, the voxel colors of $x, \lfloor y \rfloor$ and $x, \lfloor y \rfloor + 1$ are linearly interpolated at w. Since the size of the

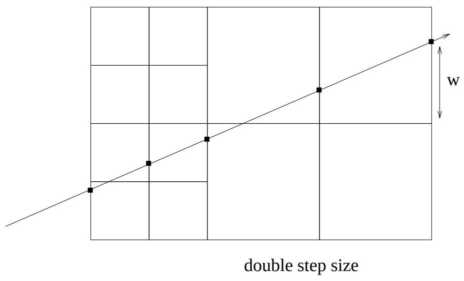

Figure 8: *The samples are always on the vertical grid lines, where w indicates the subvoxel vertical sample location. The switch to a double step size must occur at an even step.*

pixel footprint is about the size of a voxel, this simple filter is satisfactory.

The traversal algorithm has to switch to a lower resolution at some point. Since the steps are of unit size along the major direction it is rather simple to double the step size, and respectively, the ray vector and its slopes. To preserve the property that the steps are always at integer coordinates of the major axes, the switching to a double step size at the lower resolution must occur at an even step of the current resolution (see Figure 8).

The switch to a lower resolution occurs at the distance where the voxel footprint in the image is narrower than a pixel. In other words we avoid undersampling the voxels. Since the vertical field of view is not equal to the horizontal field of view, we consider only the horizontal one allowing vertical oversampling or undersampling to occur in some rare cases. In particular, when the viewing pitch angle is low (i.e., shallow), the pixel footprint tends to elongate and may cause significant undersampling. Vertically supersampling the pixels to compensate for elongated footprints is not too costly since it does not require accessing a larger number of voxels. We have implemented a variation of supersampling where each pixel has been supersampled by parallel rays. The relaxed assumption that the rays cast from a single pixel are parallel enables efficient implementation without any significant loss of quality.

3 Parallel Implementation

Sequential implementation of the rendering algorithm cannot deliver the desired real-time rates on contemporary workstations. It is vital to use a powerful parallel machine, not only to speed up the rendering but also to support the processing of very large databases. The application requires flying over thousands of square kilometers, including many 3D objects. Taking into account the hierarchical data structures, the total amount of data is over 35 Gigabytes (see below). Moreover, the relevant data, i.e. the image footprint, must be continuously loaded into main memory. Thus, the machine needs to have very large first and secondary memories, and high speed channels between them. All these requirements need the support of a machine with high speed and very large storage capacity, with large bandwidth busses. However, a postprocessor is used to further accelerate the image generation rate and to enhance the image quality (described below).

A block diagram of the system is illustrated in Figure 9. The IBM Power Visualization System (PVS) is the parallel machine described below. It is controlled by an IBM RS/6000 Support Processor which also serves as a connection to the external world. It reads the commands from the user's control stick and sends control command from the PVS to a Post Rendering Processor (PRP) (see below) through an Ethernet LAN. The images generated by the PVS are sent via an HIPPI (100MB/Sec) channel to the PRP and are displayed on a standard NTSC monitor.

3.1 The IBM Power Visualization System

The IBM Power Visualization System (PVS) was designed to provide computational power, high-speed memory and I/O to realize very large amounts of complex data. The PVS is a shared memory architecture consisting of up to 32 parallel processing units, and up to 2.5GB of internal local and global memory.

The architecture consists of up to eight processor cards. Each processor card consists of four processor elements, each composed of an Intel i860XR or i860XP microprocessor operating at 40 or 45 MHz.

Processor storage consists of 16 MBytes of local memory per processor and global memory which can be increased to 2048 MBytes. The global memory consists of up to four memory cards. Each card is designed to

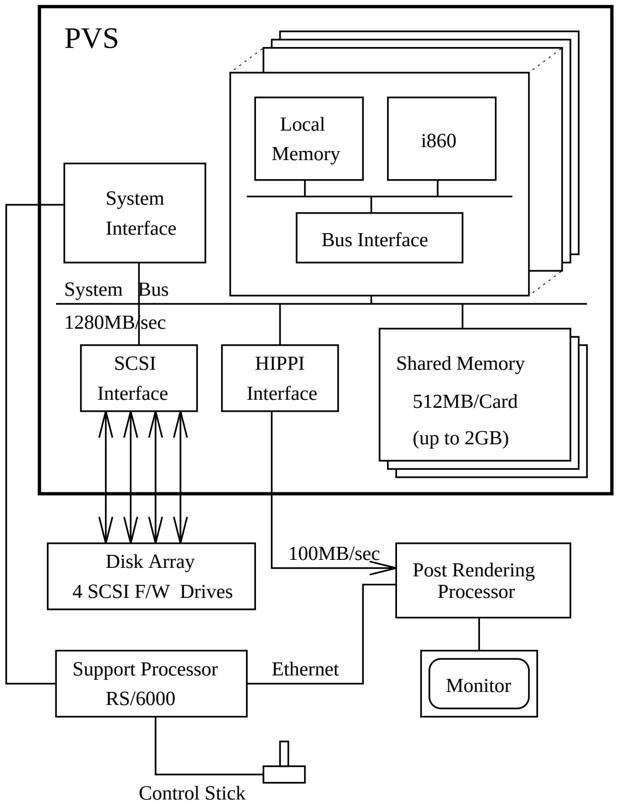

Figure 9: *A block diagram of the system*

provide a data bandwidth of 640MB/sec or 720MB/sec. This is accomplished by partitioning the memory into four interleaved memory banks, each of which can perform memory reads and writes, thus reducing the latency and improving throughput. In addition, there is interleaving between cards if there are multiple memory cards in the system.

An SCSI interface card with four Fast/Wide (peak of 20 MB/Sec) controllers is used to connect to the disk array. Using an SCSI disk reduces the system price and promises upgradability. The PVS strips the data into all the controllers, giving a throughput of more than 70MB/Sec. Thus, it can contain the database and can load the memory fast enough.

The PVS also provides means for producing and outputting the frames in real-time. A video controller which is attached via an HIPPI channel to the Server, includes two logically distinct frame buffers with a total capacity of up to 32MB. The first 8-bit buffer is used for workstation graphics and text from an X-Windows system. The other is a 24-bit/pixel double-buffered full color RGB image buffer at HDTV resolutions and above.

3.2 Implementation Details

The rendering task is partitioned among the processors. One of the them, selected arbitrarily, operates as the master and the rest are the slaves. The master processor, among its many tasks, sets the viewing parameters of the next frame, including the new positioning of the camera and its orientation, according to the trajectories of the flight. The generated image is treated as a pool of columns, and each slave processor renders one column of pixels as an atomic task. As soon as a slave terminates rendering one column, it picks a new column from the pool. Access to the pool is monitored by a semaphore operation provided by the PVS library. The semaphore forces exclusive access to the pool, so that only one processor at a time can pick a column. Moreover, as soon as the last columns of the frames have been picked and generated, the free processors start to generate the first columns of the next frame. Using this strategy the processors are kept busy with a perfect load balancing.

Although the PVS contains as much as two Gigabytes of RAM, the database cannot be loaded entirely into the main memory. The entire database is stored in the disk array while the relevant sections (i.e., the image footprint) are loaded dynamically into memory. The terrain database is partitioned into small square tiles. According to the viewing parameters, the master draws the rectangular frame of the image footprint on the terrain, makes sure that the tiles that fall in the frame footprint are already in memory, and loads the missing tiles from the disk-array. Since the footprint changes incrementally, only a few tiles need to be loaded at each frame. A large configuration of the main memory consists of two Gigabytes and can contain more than the size of one frame footprint; thus, we use an extended footprint. Some of the tiles that are in the larger footprint would otherwise have been loaded on the next frame. Thus, the extended footprint saves many critical loadings. The tiles that are loaded are actually prefetched and their presence is not critical for correct rendering of the current frame. This mechanism is found to be very efficient as it can treat fast changes of camera, as much as one entire field of view per second.

Pitch	32p	16p	8p	4p	2p
14.6	11.0	5.4	2.5	1.1	0.36
26.5	15.6	7.6	3.6	1.5	0.51
39.0	16.3	8.0	3.7	1.6	0.53

Table 1: *Frames per second (fps) generated by the PVS as a function of the number of processors. Each line of the table shows the fps sampled at different pitch angles.*

3.3 Results

Quantitative results are presented in Table 1. The frame generation rate of the PVS has been measured at three different angles for different numbers of processors. These rates are further accelerated by the PRP to achieve a steady frame rate of 30 frames per second. However, from these numbers we can learn about the performance of the algorithm. First, a linear speed up is achieved. The above numbers imply that by doubling the number of processors the frame generation rate is more than doubled. This is because one processor is dedicated as the master processor. A second observation is the dependency between the performance and the pitch angle. As the pitch angle gets smaller, the frame generation rate decreases. This is because at small pitch angles the frame footprint extends. However, since we use a hierarchy, the size of the footprint is bounded. It should be noted that there is a speed quality tradeoff. By scaling the pixel-to-voxel ratio it is possible to speed up the frame generation rate. As the voxels used are "scaled", the footprint sizes (voxelwise) decrease. Of course as the pixel-to-voxel ratio increases the voxels are oversampled and the image is blurred. However, this ratio is used as a tool to tune the quality of the image as the frame generation rate is guaranteed by the PRP.

A typical database consists of a large terrain with tens of target areas. The global terrain is a 1 meter resolution playground of 55x80 square kilometers, which is 4.5Giga voxels. Each voxel is four bytes, thus the size of the global terrain is 17.6 Gigabytes. Adding the hierarchy requires a third more (5.9G), thus 23.5G bytes in total. Each target area consists of three levels of detail: 2.5x2.5 square kilometers of 0.5 meter resolution, 1.25 by 1.25 square kilometers of 0.25 meter resolution, and 625 by 625 square meters of 12.5 centimeters for the highest resolution. A single target area database size is

0.3G bytes. No hierarchy is needed because the coarser levels are given in the global terrain. Assuming, for example, 40 target areas require over 12G bytes. In total 35G bytes are needed for the terrain data. The 3D objects consume more space. A typical object requires about 1.5M bytes. Here we should mention that if true colors were needed, and not only grey levels, the database would have been almost double the size.

4 The Post Rendering Processor

The images generated by the PVS are asynchronous since their rate is dependent on the viewing direction. The frames are created at a rate of 10-15Hz. From these images an NTSC video signal should be produced. The image fields, that is the even/odd NTSC rows, have to be transmitted at a rate of 60 Hz (interlaced). If the fields contain only the last frame generated by the PVS, the human eye would detect jumps every time the frame is changed. To achieve a smooth sequence of images that do not irritate the eye, it is necessary to generate the frames at a synchronous rate. The idea is to simulate small changes in the camera position as 2D transformations applied to the last frame available. However, unlike the interpolation method [2], here the image needs to be extrapolated. The image is digitally warped on the fly with respect to the flying trajectories. The warping is done using the *Datacube* MaxVideo machine. The MaxVideo serves as the Post Rendering Processor and is also used for some other 2D functions, such as automatic gain control (AGC), filtering, scaling and rolling the image. It should be emphasized that interpolating between available frames is not possible since it would cause a small but critical latency which is not acceptable in real-time systems, where real-time feedback is vital. The extrapolated images may have minor differences from the next real frame. However, since the flying trajectories are known and are relatively smooth, the transition between the extrapolated frame to the real frame is smooth. Since the warping function might mapped back a point outside the source frame, the real frames are slightly larger and include margins. These margins are relatively small since flying trajectories are smooth, recalling that the real images are created at a rate of 10-15Hz.

Given an image A generated by some camera position, the goal is to warp the image so that it approximates the image B that would have been generated by a new camera position. Let us define f as the function that maps B back to A, such that if p is a point in the 3D space that is seen from pixel $\bar{x}$ in B and in pixel $\bar{x}'$ in A, then $f(\bar{x}) = \bar{x}'$. Once f is known, the pixel color at $\bar{x}$ is determined by bilinear interpolation at $\bar{x}'$.

A perspective warp function would be best; however, the MaxVideo supports a second degree polynomial warp. Thus, f is composed of two functions $f = (f_x, f_y)$, where f_x and f_y are two second degree polynomials:

$$f_x(x, y) = a_1 + a_2x + a_3y + a_4xy + a_5x^2 + a_6y^2$$

and

$$f_y(x, y) = b_1 + b_2x + b_3y + b_4xy + b_5x^2 + b_6y^2$$

.

To determine the above 12 coefficients, a set of $2n > 12$ equations is explicitly defined by n control points. The system of $2n$ equations is‘ solved using a least squares method. The $2n$ equations are defined by calculating the position of n points in the 3D world coordinate for camera position A and B, and projecting them back to the image space. We used nine points evenly distributed in the image plane. During rendering the 3D coordinates of the terrain point seen from those nine fixed locations are registered.

Denote the vector of unknown coefficients by $C_j = (a_j, b_j)$, $1 \leq j \leq 6$. The system that we need to solve is $FC = X$, where $F_i = (1, x_i, y_i, x_iy_i, x_i^2y_i^2)$ and $X_i = ({x'}_i, {y'}_i)$, $1 \leq i \leq n$. These are two sets of n equations for six variables. Assuming n is larger than six, the least squares solution gives us $C = (F^tF)^{-1}F^tX$. Note that for $n = 6$, $C = F^{-1}X$.

Note also that since the roll rotation is a simple 2D transformation, it can be implemented directly using the MaxVideo warper.

5 Modeling Voxel-Based Objects

The process known as *voxelization* converts a continuous geometry into a discrete representation [4]. Many existing models are represented by a polygon mesh that approximates the real object. However, for a photo-realistic application *photo mapping* [9] is essential (see Figure 10). This requires warping the photograph of the object so that it matches the 3D model, and then applying it as a texture map to the voxels. Alternatively, a sculpting technique

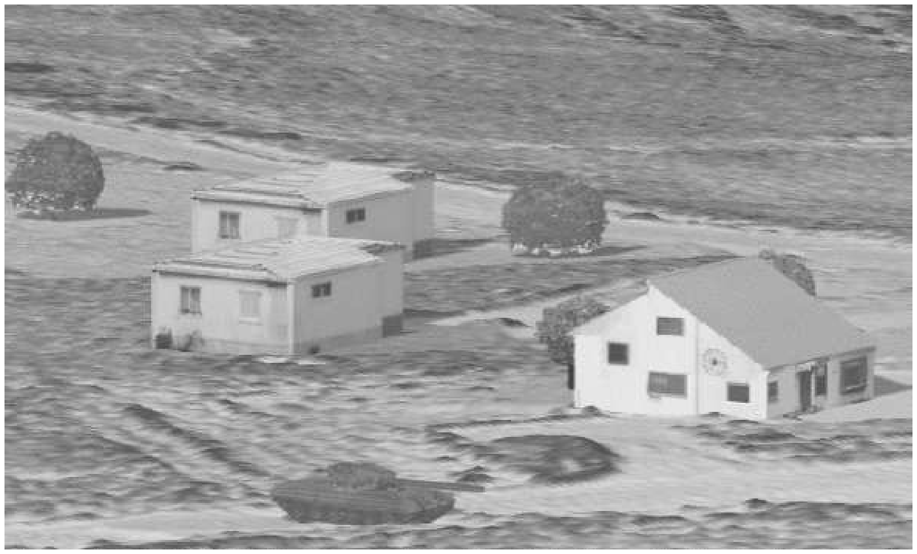

Figure 10: *Voxel-based objects: houses, trees and a tank.*

can be employed. Given a set of images of an object from known directions, one can craft the shape of the model by peeling away the background voxels around the projected images. We start from a solid box of "black" voxels. Then, given an image, rays are cast from the background pixels back into the voxels, "clearing" the voxels encountered into background color. Repeating this process from many images which view the model from different directions, leaves the non-background voxels with the shape of the model. This process of reconstruction from projection yields the texture mapping inherently by projecting the non-background pixels back towards the voxels by means of ray casting.

A simplified implementation of the above sculpting technique has been employed. We use only three photographs of a toy object. For example, the three photographs of a toy Scud are shown in Figure 11(a). These three photographs are scaled down to the voxel space resolution as can be seen in Figure 11(b). At this stage the object is separated from the background pixel. If this is not achieved by color thresholding, a contour is drawn manually around the object. The result of the sculpting process and the photomapping from these images is a 3D voxel-based textured object which can be rendered from arbitrary viewing direction. The images shown in Figures 12 and 14 are rendered very close to the object in order to observe the fine details. Note

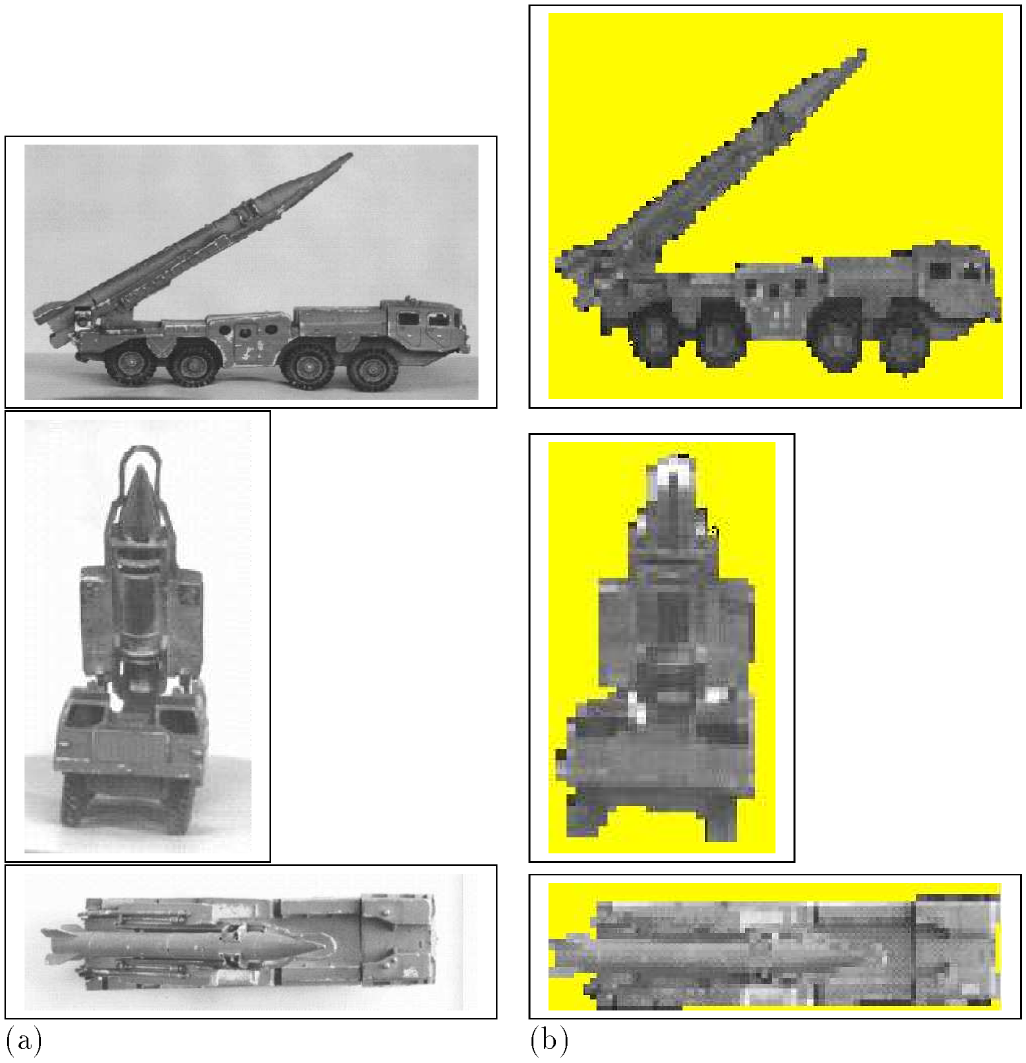
(a) (b)

Figure 11: *A toy Scud. (a) Three photographs (side,front,top). (b) the images after scaling to the voxel space resolution.*

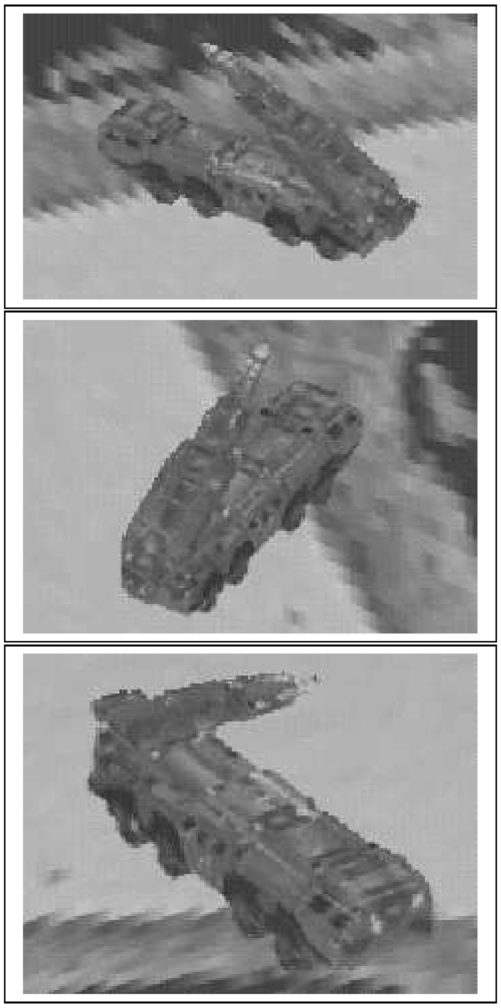

Figure 12: *The voxelized Scud from three different viewing directions.*

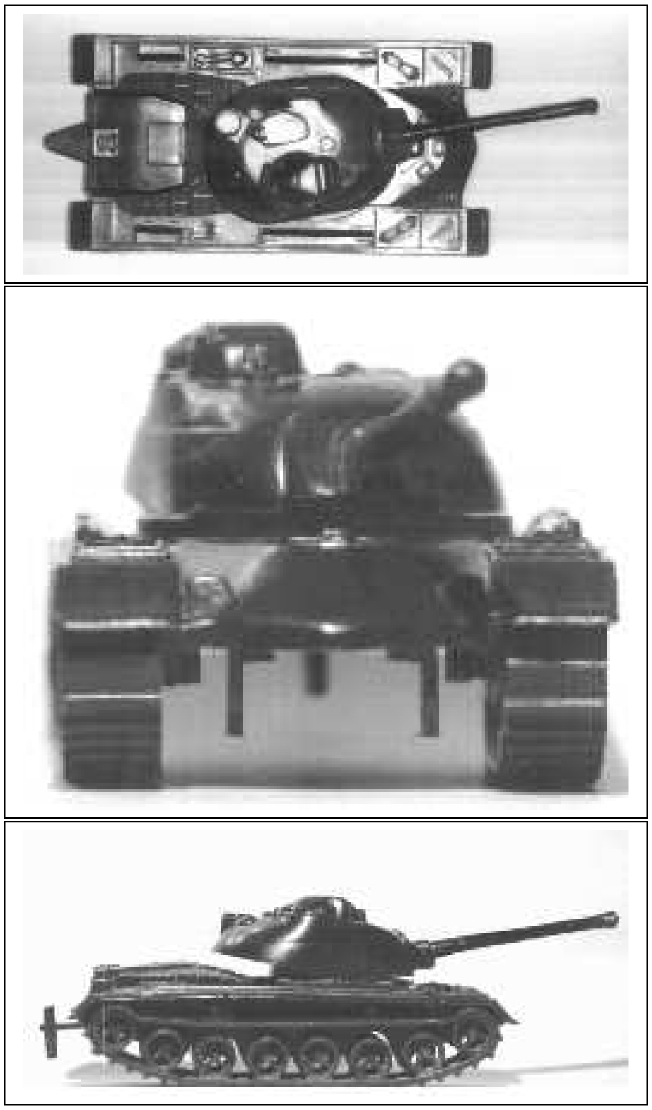

Figure 13: *Three photographs of a T62 tank.*

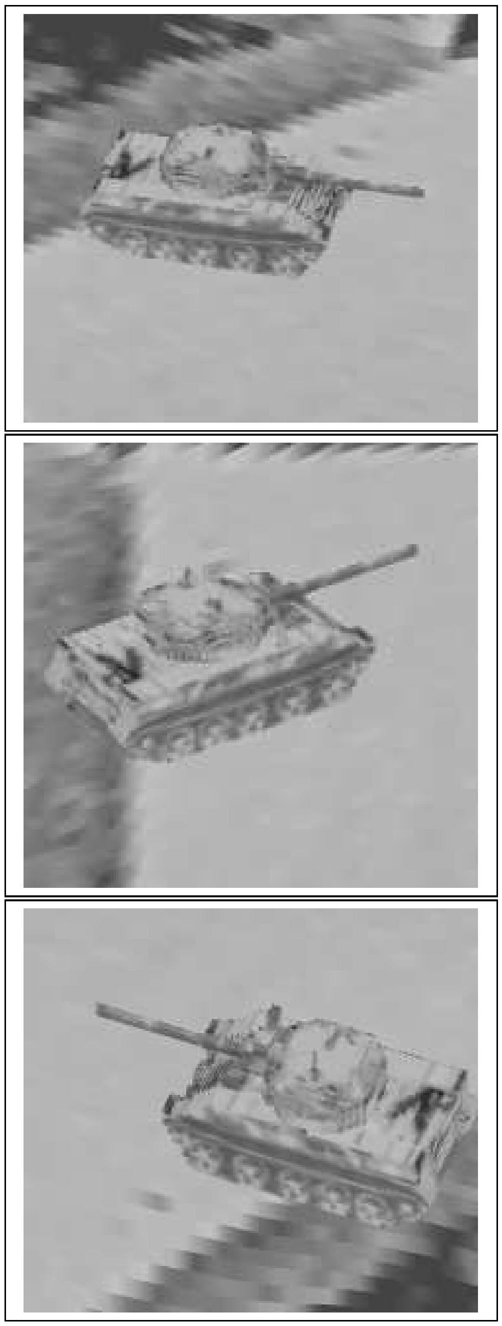

Figure 14: *The voxelized T62 from three different viewing directions.*

that the resolution of the object is higher than of the terrain. However, these objects are to be seen from a distance as shown in the previous images.

6 Current Porting Activity

The development project was started in 1992 while the PVS was state-of-the-art, but since then the processing power of a single processor has grown by a factor of 10 compared to the i860.

Although the performance achieved on the PVS is satisfactory, it is clear that a faster platform will allow us to deal better with higher resolutions, and more and more objects of richer detail. The portability of the application permits the adoption of a new parallel shared memory architecture according to the behavior of the commercial market.

Using a distributed memory machine was ruled out since the application was designed for shared memory architecture. The only company that manufactures shared memory architecture in the same price range as the PVS is Silicon Graphics Inc. (SGI). SGI's machines have a similar architecture to the PVS with the exception that SGI uses a large cache (4 MBytes) in contrast to the 16MBytes of the PVS's local memory.

SGI offers the Challenge with a maximum of 36 R4400/250Mhz CPUs and the Power Challenge with a maximum of 18 R8000/90Mhz CPUs. The Power Challenge was designed for floating point applications and each CPU is more than twice as fast in such applications. In integer applications the R4400 and R8000 have the same performance, giving the Challenge double the performance of the Power Challenge. Both machines can store up to 68.8 GBytes internally and up to 6.3 TBytes externally.

The primary results on the SGI Challenge indicate a speed up of about 4.5 times faster than the PVS, while the scalability remains linear. This is achieved with only minor changes in the code used for the PVS, mainly to compensate for the absence of local memory.

7 Final Remarks

We have presented a discrete ray casting algorithm accelerated by ray coherence and multiresolution traversal. The time complexity of the algorithm is proportional to the number of image pixels, which can be regarded as constant. The combination of the efficient rendering algorithm and the powerful parallel machine results in a real-time photo-realistic visual flythrough. The parallel rendering task partitions the image space among the PVS processor elements, putting the load at the scene space stored in the PVS shared memory. Due to data prefetching, the wide bandwidth of the busses, linear speed-up has been observed as well as hardly any read or write contentions in the shared memory. We have achieved perfect load balancing by overlapping between frames.

It should be noted that the sequential version of the rendering algorithm runs well under two seconds on an SGI workstation for a terrain size that can fit into main memory. It is expected that in the future, with the progress of memory bandwidth and CPU speed, visual flythrough will be able to run in real-time on advanced sequential workstations.

8 Acknowledgments

This work was developed at Tiltan System Engineering in collaboration with IBM Israel. Our thanks to Meir Nissim-Nir who built the terrain database, and to Sylvia Kohn who built the objects and developed some new ideas during the process. We thank all the stuff at IBM who helped us along in many different ways.

References

[1] K. Akeley. Reality Engine graphics. In *Proceedings of SIGGRAPH '93*, pages 109–116. ACM, 1993.

[2] E.S. Chen and L. Williams. View interpolation for image synthesis. In *Proceedings of SIGGRAPH '93*, pages 279–288, 1993.

[3] D. Cohen and C. Gotsman. Photorealistic terrain imaging and flight simulation. *IEEE Computer Graphics and Applications*, 14(2):10–12, March 1994.

[4] D. Cohen and A. Kaufman. 3D scan-conversion algorithms for linear and quadratic objects. In A. Kaufman, editor, *Volume Visualization*, pages 280–301. IEEE Computer Society Press, Los Alamitos, CA, 1991.

[5] D. Cohen and A. Shaked. Photo-realistic imaging of digital terrains. *Computer Graphics Forum*, 12(3):363–373, 1993.

[6] S. Coquillart and M. Gangnet. Shaded display of digital maps. *IEEE Computer Graphics and Applications*, 4(7):35–52, July 1984.

[7] Evans and Sutherland Computer Corporation. Esig4000 technical overview. Technical report, 600 Komas Drive, Salt Lake City,UT 84108.

[8] J. Folby, M. Zyda, D. Pratt, and R. Mackey. Npsnet: Hierarchical data structures for real-time three dimensional visual simulation. *Computers and Graphics*, 17(1):437–446, 1991.

[9] A. Kaufman, D. Cohen, and R. Yagel. Volume graphics. *IEEE Computer*, 26(7):51–64, July 1993.

[10] Y.G. Leclerc and S.Q Lau. Terravision: A terrain visualization system. Technical report, Technical Report 540, SRI international, April 1994.

[11] C. Lee and Y.G. Shin. An efficient ray tracing method for terrain rendering. *Pacific Graphics '95*, pages 180–193, 1995.

[12] F.K. Musgrave. Grid tracing: Fast ray tracing for height fields. Technical report, Department of Mathematics, Yale University, December 1991.

[13] D.W. Paglieroni and S.M. Petersen. Height distributional distance transform methods for height field ray tracing. *ACM Transactions on Graphics*, 13(4):376–399, October 1994.

[14] L. Williams. Pyramidal parametrics. In *Computer Graphics*, volume 17(3), pages 1–11, 1983.

[15] J. Wright and J. Hsieh. A voxel-based, forward projection algorithm for rendering surface and volumetric data. In A.E Kaufman and G.M Nielson, editors, *Proceedings of Visualization '92*, pages 340–348. IEEE Computer Society Press, 1992.

9 Bio

Daniel Cohen-Or is senior lecturer at the Department of Computer Science in Tel-Aviv University. His research interests include rendering techniques, volume visualization, architectures and algorithms for voxel-based graphics. He received a BSc Cum Laude in both Mathematics and Computer Science (1985), an MSc Cum Laude in Computer Science (1986) from Ben-Gurion University, and a Phd from the Department of Computer Science (1991) at State University of New York at Stony Brook.

Dr. Cohen-Or has extensive industrial experience, in 1992-3 he designed a real-time flythrough at Tiltan System Engineering, during 1994-5 he worked on the development of a new parallel architecture at Terra Computer, and recently he has been working with MedSim Ltd. on the development of an ultrasound simulator. He can be reached at the Department of Computer Science, Tel-Aviv University, Israel. His e-mail address is daniel@math.tau.ac.il

Eran Rich received the B.Sc. degree in Electrical Engineering from the Tel-Aviv University in 1992. Since then he has been working at Tiltan System Engineering, leading the development group of the flythrough and other applications. His main areas of interest are parallel computing, computer graphics and image processing. He is a member of IEEE Computer Society and IEEE.

Uri N. Lerner is a doctoral candidate in Computer Science in Stanford University starting September, 1996. He received his B.Sc. summa cum laude in both Mathematics and Computer Science from Tel-Aviv University in 1995. He joined the Tiltan System Engineering in 1994, and has led the flythrough team for the last year. His current interests include computer graphics and parallel computing with a special emphasis on voxelized terrain rendering and anti-aliasing methods.

Victor Shenkar is the General Manager of Tiltan System Engineering Ltd. in Bnei-Brak, Israel. He received a B.Sc and a M.Sc in Electrical Engineering from the Technion, Israel Institute of Technology and a Ph.d from Stanford University. His current interest include accelerated image rendering and digital photogrammetry.