


6 Conclusions

We have described a parallel terrain image rendering system producing high quality video-resolution imagery at high rates. Our long term goal remains to achieve real-time visual flight simulation. We believe that this cannot be achieved on state-of-the-art graphics workstations without significant improvement of their ability to handle very large datasets. Parallel processing, however, does hold promise for this application, as we have demonstrated here. We continue to optimize the speed/quality tradeoff by optimizing our rendering algorithms and identifying, and alleviating, bottlenecks in system throughput.

Acknowledgements

We wish to thank Richard Cleave of Rohr Productions Ltd. (Nicosia) for providing the satellite image data used in this work, and John Hall of the Geological Survey of Israel (Jerusalem) for providing the DTM data. Partial support for the work of the second author was provided by the Fund for Promotion of Research of the Technion.

References


Figure 8: The number of texture pixels actually used by each of the nine R processors when rendering a subimage.
Figure 7: The total number of texture pixels, communicated in tiles of $64 \times 64$ pixels, shipped from each of the four T processors to the nine R rendering processors. The top four curves are in the case of an image raster scan, the bottom in the case of a Hilbert scan. It is evident that the load is evenly distributed among the T processors in both cases. The speed of the flight is such that new tiles are required all the time.
5.2 Performance

We have produced PAL video-resolution (576×768 pixels) image sequences with our parallel system, using an experimental configuration consisting of nine R, four T and one E processors. The size of the texture tiles in the database are 64×64 pixels. Inter-processor communication overhead incurred by the system is measured by the amount of data shipped to the R processors per rendered image. The efficiency of the parallel architecture is measured by the load balance between the R processors, and the load balance between the data cache T and E processors.

Fig. 7 shows the total number of texture pixels, communicated in tiles of 64×64 pixels, from the four T to the nine R processors. To check the gain from using the Hilbert scan to render pixels, we compared this to the identical statistic obtained when using a regular raster scan, also plotted on the graph. The graph shows that the spatial image coherence utilized by the Hilbert curve reduces to approximately 55% the amount of communication required. It may also be seen that in both (the Hilbert and raster scan) cases, the load is evenly distributed between the four T processors, confirming the efficiency of the tile addressing scheme described in the previous section.

Fig 8 depicts the number of texture pixels actually used by each of the R processors during the rendering process. The sum of these numbers (on the average 5.8MB per frame) is more than the number of communicated pixels (2.4MB per frame), as the internal caches of the R processors supply more than half the required pixels.

Note that not all communicated pixels may be actually used, as they are supplied in integral tile chunks, and some of the pixels in the tiles contributing to the image pixels along a subimage border may be redundant. For this configuration, the amount of overhead is typically 60% of the amount of non-redundant data. It could theoretically be reduced by using smaller tiles, but this would result in an increase of memory required to store and maintain the texture pyramid table in the R and T processors. We found experimentally the tile size of 64×64 pixels to be optimal. The fact that all nine curves of Fig. 8 are close indicates that the rendering load is more or less evenly distributed among the R processors. A typical image pixel is a filtered value of approximately 13 texture pixels. The exact number per image pixel depends on the local scale of the image in the vicinity of that pixel relative to the hierarchy of scales in the texture pyramid.

When all 26 processors of the Meiko are used (a typical configuration would be 16 R processors, six T processors, two E processors and one V and W processors), we have been able to achieve an almost optimal speedup in rendering time. This results in the production of a PAL-resolution 24-bit color image every 4 seconds, including I/O and communication overheads.
- **R (render)** - the rendering processors. These processors receive the view parameters from the V processor, perform the rendering of the subimage, as described in Sections 3 and 4, and send the results to the W processor. The R processor has a small internal cache for texture and elevation tiles. If a required texture (elevation) tile is not in its cache, it is shipped in from the T (E) or E processor. Each R processor renders the same subimage for all frames in the sequence. This way temporal coherence of the image sequence is exploited to maximize the hit ratio of its cache.

- **W (write)** - the output processor. Gets rendered subimages from R processors, combines them to one final image, and stores it.

- **T (texture)** - a texture cache. Maintains a large cache of texture tiles, supplying them on demand to the R processors.

- **E (elevation)** - a elevation cache. Maintains a large cache of elevation data tiles, supplying them on demand to the R processors.

Apart from the V and W processors, which are unique, there may be a variable number of the other types of processors. Obviously, a large number of R processors, where the actual rendering is done, is recommended. If more than one T (E) processor is used, the load is distributed evenly between these n processors. We ensure this by using the following addressing scheme: Processor $T_k$ ($E_k$) ($0 \leq k < n$) deals only with the $(i, j)’$th tiles such that $(i + j) = k$ (mod $n$). Any R processor requiring a specific tile communicates with the appropriate T (E) processor, according to this rule. Fig. 6 shows the assignment of the tiles of a database to three cache processors.

![Figure 6: Assignment of data tiles to three cache processors, according to the mapping $M(i, j) = (i + j)$ mod 3.](image)

When a R processor encounters a situation where a required elevation tile is not present in its own RAM, it sends a request to the E processor, and continues rendering other triangles. The tile is processed on arrival. Similarly, when a T or E processor encounters a situation where a required tile is not in its cache, it initiates an I/O procedure, continuing the processing of other requests.
the interprocessor connection topology configurable by the user. The rendering is performed in a
three-stage pipeline, the second stage consisting of a variable number of processors (see Fig. 5),
each rendering a small subimage of the final image. Another two types of processors maintain
texture and elevation data caches. The processors are divided into the following five categories,
and connected through the VCS (Virtual Computing Surface) of the computer.

- V (view) - the main control unit. This processor generates the view parameters for each
image of the sequence. As the image is rendered in parts, the processor performs hierarchical
ray-casting of the pixels along the borders of the subimages, and supplies a list of elevation and
texture data tiles required for the rendering of each subimage to the appropriate R processor,
as described in Section 2.
Figure 3: Hierarchical data structures used for rendering. (a) Elevation pyramid. (b) Texture pyramid.

(i.e. filtering of the appropriate pixel polygon in texture space). Even though the tiles may be compressed (e.g. with JPEG [15]), a cache must be maintained explicitly to minimize unnecessary I/O. The image spatial coherence guarantees that the set of data tiles required for a pixel will have a significant intersection with the set of tiles required for an adjacent pixel. To increase the hit-ratio of the cache, we render the screen pixels in an order reflecting their proximity. Figure 4 shows two possible rendering orders: a suboptimal “raster” order, and a better “space-filling” curve with the attractive feature that points geometrically close to each other are usually not far apart along the curve. Consequently, most tiles do not have to be loaded again, when going from pixel to pixel. For image sizes which are powers of two, a recursive construction exists for such optimal Hilbert curves (see e.g. [17] for details on this and other curves), and for sizes which are not powers of two, the construction of [12] may be used.

5 The Parallel System

5.1 Architecture and Design

Our parallel system for terrain rendering is implemented on a Meiko computer containing 26 Intel i860 RISC processors. Each processor has 8MB RAM, which is relatively small for our application. Each processor also has 8 unidirection communication channels (with 1 Mbyte/sec bandwidth),
Fig. 1 (c) shows the pixel polygons associated with a 34x30 pixel image of the terrain of Fig. 1 (d). In effect, the vertices of these pixel polygons define a warp of texture space to image space, as these vertices have to be mapped to a regular grid representing the corners of the screen pixels. Note that our description of rendering as a texture warp implies that, ultimately, all texture pixels in the pixel polygon contribute to the final image, even those who are occluded in reality. We found this approximation to be quite reasonable, the occluded texture pixels affecting only a few image pixels, and, in fact, “emphasizing” terrain silhouettes. There are two ways to obtain the average of the texture pixels within a pixel polygon:

- **Direct Filtering:** Exhaustive averaging of the texture pixels in the pixel polygon. This gives an accurate result, but when the pitch angle is small or viewpoint very high, the pixel polygon may be arbitrarily large. In this case, a large number of texture pixels have to be averaged, which is time-consuming. This is compounded by the fact that not all the texture may reside in RAM, so apart from CPU time, costly I/O may be required.

- **Filtering on a pyramid:** This is a simple way to efficiently approximate the result of direct filtering. A texture pyramid [16] is constructed, such that a texture pixel at any level in this pyramid is the average of four values from the previous level (and covers four times the area) (see Fig. 3(b)). The base of the pyramid is the original texture database. All levels of this pyramid are also stored as tiles. All texture tiles are the same size in pixels, but cover different size areas on the terrain (depending on the level). Filtering is performed by sampling the pyramid. Various techniques are employed, depending on the quality of approximation required (see [5] for a survey). The simpler techniques use only one level of the pyramid, determined by the area of the pixel polygon, and the more sophisticated techniques interpolate between two adjacent levels. The latter is obviously more accurate, but also slower.

The direct filtering method is too costly and the accuracy of the crude pyramid approximation is not sufficient for rendering high quality animation sequences. We employ a hybrid approach, sampling only one pyramid level, determined by the area of the pixel polygon. However, we sample at a level lower than what the standard pyramid filtering algorithms prescribe, averaging more pixels, better approximating the shape of the pixel polygon. This results in a significantly more accurate result, at the expense of some more time.

### 4 Dataflow Management

The most time consuming component of image sequence rendering in large terrain databases is the I/O incurred by loading the texture tiles containing data needed for the rendering of a screen pixel
Fig. 2). In cases where the horizon is present in the image, for which the bounding polygon is potentially infinite, it is truncated to the borders of the texture database. For the ray-casting operation, we use the hierarchical ray casting technique. It is an accelerated version of traditional incremental ray casting, originally designed to run on voxel terrains. The data structure supporting this algorithm is the elevation pyramid (Fig. 3(a)). At its base lies the original elevation database. A value at any higher level is the maximum of four adjacent elevation in the level below it, forming a quadtree structure. Using an elevation pyramid, we have adapted the hierarchical ray-casting method to run on triangle meshes in a straightforward manner.

![Figure 2: Hybrid ray-casting and projection technique.](image)

After ray casting the border pixels, the bounded triangles are projected onto the image plane. Using a modified Z-buffer scheme, and the rational interpolation technique [6], we scan-convert the projections of all these triangles to obtain the \((x, y)\) coordinates on the terrain, and in texture space, corresponding to each screen pixel corner (a similar procedure is used in the texture-mapping hardware of the SGI Reality Engine [1]). This is needed for accurate texture filtering, as we elaborate on in the next section.

3 Filtering

For high quality anti-aliased rendering, every screen pixel should be considered not a point, but a small square of finite area. The color value assigned to the screen pixel during rendering should be a weighted average of all texture pixels projected onto this square.

To locate these pixels, map the four corners of the screen pixel to the terrain. This defines a \textit{pixel polygon} in texture space. All texture pixels within this polygon participate in the average.
Figure 1: (a) A sample texture (merged Spot+Landsat satellite image). The polygon roughly indicates the part of the texture contributing to the perspective view in (d). (b) 3D oblique view of the terrain corresponding to the area and view marked in (a) reconstructed from a subsampled DTM as a triangle mesh. (c) Pixel polygons of a 34x30 pixel screen corresponding to the view (b). Note how large some of the polygons are, implying that the filtering operation for that screen pixel might be costly. (d) Oblique terrain image created by mapping the texture of (a) onto the terrain of (b), or, alternatively, warping (a) according to (c) onto the screen pixels.
In this paper we present efficient algorithms to alleviate these three bottlenecks, namely, hidden surface elimination, dataflow and filtering. These algorithms were developed as part of an ongoing effort to develop real-time visual flight simulators at a modest budget. At the current state of technology, it seems that only a hardware platform incorporating parallel processing has any chance of achieving this goal. Towards this end, we are experimenting with a Meiko parallel computer, containing 28 i860 processors. We present a parallel architecture and implementation of our algorithms on a Meiko computer, designed specifically to cope with the three bottlenecks mentioned above. The paper is organized as follows: Sections 2,3 and 4 describe our methods of alleviating the bottlenecks. Section 5 describes our parallel implementation. We conclude in Section 6.

2 Hidden Surface Elimination

There are two main approaches in the literature to hidden surface elimination in terrain triangle meshes:

- **Ray-casting:** A ray is cast from the viewpoint through each screen pixel center and its first intersection with the terrain calculated. This may be thought of as bringing the screen to the terrain. The main advantage of ray-casting is that there are no computational expenses for the hidden or clipped parts of terrain, but even so, existing ray-casting algorithms [10] are extremely slow. Attempts to increase speed by different types of acceleration methods (hierarchical [7, 3], parametric [11]) improve the situation significantly, but still do not achieve real-time speeds on high-end workstations. An advantage, however, of the ray-casting method is that parallelization is almost trivial. Even so, to achieve reasonable rendering time (e.g. less than one second) for an image of video resolution - a massively parallel computer (e.g. [14]) or special parallel ray-casting hardware (e.g. [13]) is required.

- **Projection:** The 3D triangles are projected onto the 2D screen, and hidden surfaces eliminated with a Z-buffer algorithm. This can be thought of as bringing the terrain to the screen. The main advantage of this approach is its speed, but there are also expenses for the hidden/clipped parts of the terrain, which do not contribute to the final image. In large terrains, it is hard to tell apriori which triangles will contribute to the final image. The traditional “brute-force” solution of projecting all scene geometry and letting the software/hardware eliminate the redundant ones, is obviously impractical in this scenario.

We use a hybrid ray-casting and projection technique. Ray-casting is performed only for the pixels along the screen border. The intersection points of these rays with the terrain form the vertices of a polygon, bounding the terrain triangles whom suffice to render the scene. Only triangles in this region are projected. The rest of the triangles may be eliminated (“culled”) (see
appropriate viewpoint, looking at the appropriate viewangles at the terrain. For a description of
the state of the art in these, and related, applications, see the survey of Cohen and Gotsman [2].

Terrain rendering is achieved through the combination of two databases - one containing in-
formation about terrain color (texture) from a vertical viewangle (such as a satellite image) and
the other containing information about terrain topography (elevation), supplying the third dimen-
sion. For the elevation database, the term DTM (Digital Terrain Model) is widely used. In most
applications the resolution of the DTM is considerably less than that of the texture. In polygon
systems, the terrain surface is reconstructed from the DTM as a $C^1$ triangle mesh, each triangle
corresponding to many texture pixels. This contrasts with voxel systems [3], in which the DTM
is interpolated to produce values at a resolution identical to that of the texture. An elevation and
RGB color are then associated with each such voxel (column). The collection of voxels is a $C^0$
reconstruction of the terrain surface.

The oblique perspective terrain image is created by mapping the texture onto the reconstructed
terrain surface and projecting the surface onto a viewing plane, incorporating hidden surface elim-
ination [8, 4]. Fig. 1 (a) shows a sample texture, (b) the corresponding triangulated DTM, and
(d) the result of mapping this texture to the triangle mesh surface. The geometric hidden surface
elimination operation, essential for producing correct images, is the first potential bottleneck in the
rendering process.

In typical flight simulation scenarios, image sequences, over large areas, are required. The
texture database, corresponding to this area, is usually very large and cannot be held in the main
memory (RAM) of the computer. Sometimes, for views with very low pitch angles, even the
texture required for just a single image cannot be held in RAM in full resolution. The performance
of high-end graphics workstations, such as the SGI Reality Engine [1], relies on all the texture
being present in RAM, and, as such, typically cannot handle such large terrain scenarios. In our
system, the texture is stored on disk as a sequence of fixed-size squares, called tiles. At any given
moment only a small subset of these tiles are present in RAM. An important factor in any system
design is the efficient management of such offline data, minimizing the amount of I/O incurred
during rendering of the sequence. This dataflow problem is a second potential bottleneck. It may
be alleviated by special-purpose image retrieval hardware over high-speed networks [9]. We address
the problem through efficient caching.

Rendering realistic terrain image sequences requires special care, because even though each
individual image of the sequence may appear reasonable, the display of the sequence may result in
acute "flickering" and "shimmering", a manifestation of temporal aliasing. This may be alleviated,
even eliminated, by correct filtering, in both spatial and temporal domains. The filtering, essentially
weighted averaging of a large number of texture pixels, tends to be costly in time, becoming a third
potential bottleneck.
A Parallel System for Rendering Realistic Terrain Image Sequences

Gennady Agranov * Craig Gotsman †

Computer Science Department
Technion - Israel Institute of Technology
Haifa 32000, Israel

Abstract

We present algorithms for rendering realistic images of large terrains and their implementation on a parallel computer for rapid production of terrain animation sequences. By “large terrains”, we mean the use of datasets too large to be contained in RAM, a fact which significantly influences the design of rendering algorithms, and has not been addressed in other works on this subject. To achieve a good speed/quality tradeoff, we use a hybrid ray-casting and projection technique, and Hilbert space-filling curves to determine the image pixel rendering order, fully exploiting spatial coherence. A parallel version of the algorithm is presented, based on an architecture implemented on a Meiko parallel computer. The architecture is designed to relieve dataflow bottlenecks caused by slow interprocessor communications, and exploit temporal image coherence. Using our parallel system, we are able to generate a full color terrain image at video resolution, without noticeable aliasing artifacts, every 4 seconds, including I/O and communication overheads.

Keywords: Parallel processing, Rendering, Terrain Images.

1 Introduction

The speed/quality tradeoff for 3D terrain rendering is a major consideration for developers of visual flight simulation systems. To achieve true real-time flight simulation, such a system must produce at least 25 video-resolution images per second. At the same time these images should be of realistic quality, free of spatial and temporal aliasing and blurring effects. By “realistic” we mean that the images created are as similar as possible to what a pilot would see if he were positioned at the

*Partially supported by a grant from the National Council for Research and Development - Israel Ministry of Science.
†E-mail: gotsman@cs.technion.ac.il