ScanlineFlow Rasterization – A Sort-Last Algorithm for Polygon Rendering on a Multicomputer

João Madeiras Pereira and Mário R. Gomes
Instituto de Engenharia de Sistemas e Computadores/Instituto Superior Técnico, R. Alves Redol 9, 1000 Lisbon, Portugal.
{jap, mrg}@inesc.pt

Abstract
A sort-last strategy for polygon parallel rendering, namely the ScanlineFlow Rasterization algorithm, is described. Its main characteristic resides on the fact that both steps, rendering and merging, run concurrently. By taking advantage of a pipeline interconnection network and since this algorithm renders an image one scanline at a time, each node rasterizes multiple polygons active on a given scanline after synchronisation with the previous node in order to read the rasterization data concerning that scanline. The execution is pipelined, in the sense that while a node is rasterizing a scanline, it is also receiving the next scanline from the previous node and it is sending the previous scanline (already rasterized) to the next node. The algorithm makes use of the full-frame merging technique because merging a full frame from each node is very regular and easy to implement.

This solution has provided good results and is a viable alternative to implement sort-last algorithms on a multicomputer. Our developing platform consisted of a Parsytec MultiCluster machine with seventeen processors running the Helios Operating System and using the CDL (Component Distribution Language) parallel programming language.

Key words: rendering, parallelism, pipelining, pixel-merging, multicomputer.

1. Introduction

Steven Molnar [10] proposed a conceptual model for multiprocessor architectures that looks at parallel rendering as a sorting problem. For fully parallel renderers (systems where both geometry processing and rasterization are performed in parallel), this sorting involves a redistribution of data between processors, because responsibility for primitives and pixels are distributed. The sort can occur anywhere in the rendering pipeline: during geometry processing (sort-first), between geometry processing and rasterization (sort-middle), or during rasterization (sort-last). Sort-first algorithms redistribute "raw" primitives (before their screen-space parameters are known), sort-middle redistribute screen-space primitives and sort-last redistributes pixels (pixel fragments or samples).

Sort-last (SL) systems assign to each processor (a complete renderer) arbitrary subsets of primitives. Each renderer computes pixel values for its subset disregarding the screen location and then, transmits these pixels over an interconnection network to the compositing processors. These processors are responsible for solving the visibility of pixels from each renderer. This composition operation is also known as pixel merge problem [1].

SL can be done in two ways. One approach, called SL-sparse, minimises communication by distributing only those pixels actually produced by scan-conversion. The second approach, called SL-full, stores and transfers a full image from each renderer.

Sort-first and sort-middle class algorithms make use of both object-parallelism (processors are assigned subsets of primitives) and image-parallelism (processors are assigned a portion of the screen). SL algorithms instead use object-parallelism both in the geometry processing stage and in the rasterization stage.

The architectural approaches that use image parallelism suffer from two important problems:

• as machine scale, each processor has fewer and fewer pixels; thus, as the number of processors grows, the ratio of communication overhead for the assignment of objects to the processors increases;
• the per-primitive setup time also increases, since for each primitive that covers the screen-space of k processors, k processors must perform the setup calculations for rendering, which may include expensive operations such as vertex shading calculations.

On the other-hand, SL systems offer two important advantages:
• arbitrary scalability: system performance can be scaled arbitrarily by increasing the number of renderers;
• a simpler programming model: renderers compute images of their portions of the database independently and, therefore can be programmed as simple graphics systems.

However, there is one obvious problem with these systems: the images rendered by all processors must be merged, or composited, to produce the final image. This issue is the rate-limiting factor of sort-last architectures.

2. Sort-Last Algorithms Overview

As said before, the convention described in [10] proposed two types of sort-last algorithms, SL-full and SL-sparse, which make use of the concept of active and inactive pixels at each processor. We say that a pixel location is active if at least one pixel has been rendered to that position, and that it is inactive otherwise. SL-sparse algorithms merge only active pixels. SL-full algorithms merge a full frame (all active and inactive pixel locations) from every rendering processor. Full-frame merging takes advantage of the fact that merging a full-frame from each processor is very regular and, thus, can be implemented easily.

Pereira in [13] proposed a more subtle classification for sort-last algorithms based on the scheduling of the rasterization stages. In fact, the rasterization step of the graphics pipeline can be broken into two stages called pixel rendering (computing pixel values) and pixel merging (determining which pixels are visible). We classify an algorithm as sort-last since it redistributes over the interconnection network. This redistribution can occur after each node has finished to generate pixels for its subset of the graphics database, which means that merging will take place after the pixel rendering, or it can occur as each node generates pixels, which means that merging takes place during the pixel rendering. So, another variable is the specification of when the pixel rendering/composition stages are to run: are they run concurrently or consecutively? The former case is designated by Sort-Last Merging-First (SLMF) where computation and communication are overlapped. The later one is called Sort-Last Merging-Last (SLML). Again, these two sorts of algorithms make use of either the sparse merging or the full-frame merging techniques.

Let's see how known SL systems are placed in the context described above.

In the most straightforward solution to SL rendering on a multiprocessor, each node is assigned responsibility for merging pixels from some subset of the screen resolution. There are two obvious ways of implementing the pixel merging. In the first one, as each node generates pixels for its subset of the graphics database, it sends each pixel to the destination node that is responsible for merging for that pixel location. This is typically an SLMF-sparse algorithm and was explored in commercial systems [6] and in software systems [3]. In the other way, each processor renders its subset performing local z-buffer (that is, with respect to the other pixels it generates) and producing pixels into a local framebuffer. Then, the active pixels from all nodes local framebuffers are merged into the global framebuffer. We are facing an SLML-sparse algorithm. This solution implies the need of local frame- and z-buffers between the processors and pixel merging network. It has been shown experimentally that, despite of underutilization of the local z-buffering hardware, traffic savings of about 20%, when compared with the above method, should be expected [3]. An alternative SLML-sparse algorithm, when network broadcast is available, is the Distributed-Snooping merge algorithm [2]. PROOF system described in [15] is an example of SLML-sparse hardware architecture. This architecture included a pipeline of object processors (each node was responsible by an object). A list of objects that were potential contributors to the pixel colour was associated with each pixel. Then, the shading stage used those lists to implement Phong shading algorithm.

In the PixelFlow algorithm developed at the University of North Carolina, Chapel Hill [9], processors are connected by a pipeline network. Upon rendering completion each node streams its full frame to the next node. In the end, the last node contains a correctly z-buffered image. This is an SLML-full algorithm. Pereira in [12, 13] describes also two SLML-full algorithms that use different strategies in the merging step implementation: the Distributed Framebuffer approach and the Pipelined Composition approach. In the Distributed Framebuffer approach each node is made responsible for a specific area of the screen (image space partition); when the merging phase starts it asks to each other of (N-1)\(^1\) nodes for the portion of the framebuffer concerning that screen region and performs the depth-comparison on it. This procedure is performed simultaneously by all nodes. The Pipelined Composition approach was based on the PixelFlow algorithm [8]. In the straightforward implementation of this approach, each node receives the framebuffer of the previous neighbour, Z-buffers its contents with its

\(^{1}\) N is the total number of nodes.
own framebuffer and then sends the resulting framebuffer to next node. In order to exploit as much as possible the available parallelism provided by the developing platform, instead of considering a framebuffer as whole task, this was divided into small chunks, scan-lines, and pipelined these; which means that each node performs simultaneously (multi-threading) three tasks: reading a scanline from previous node, z-buffering a received scanline with its own corresponding scanline and streaming the resulting scanline to the next node.

The ScanlineFlow Rasterization algorithm that should be included in the SLMF-full category was first described in a PhD thesis [13]. This paper shows now results obtained with this algorithm.

3. The MultiCluster System

MultiCluster2 from Parsytec is a MIMD message-passing system (also known as a multicompiler) with seventeen Transputers T800, each processor with 4 Mbytes of private memory, and a reconfigurable network. It runs the Helios Operating System. With a user interface similar to that provided by Unix, Helios is designed to work on Transputers that have no memory management hardware. Helios provides a facility called multi-tasking, which enables multiple tasks to be run on one or more processors. The objects that can be multi tasked under Helios are divided into two types: tasks and threads. A task is similar to a process under Unix. It is a self-contained unit of execution. Tasks can only run on one processor at a time but more than one task can be run on the same processor. Helios tasks can communicate with each other using the standard Helios inter-task communication mechanism. A parallel program (or a task-force) is a set of tasks. A thread is an occean process, which is implemented in a particularly efficient manner directly by the Transputer in hardware. A number of threads may be running within a single task. Threads are allowed to share memory, and they normally synchronise by using semaphores. The Helios system is based on the client-server model. Server tasks control access to the system resources (such as discs, screens, and keyboards). Client tasks are application programs that access the system resources by sending requests to the appropriate servers. The Helios implementation of the client-server model allows the clients and servers to be situated in different processors.

The Component Distribution Language, or CDL, enables a programmer to carry out parallel programming under Helios. The purpose of CDL is to provide a high-level approach to parallel programming, where the programmer defines the program components and their relative interconnections (the application topology) independently of the size and topology of the Transputer network. It is, then, Helios who is responsible for mapping the task force onto the available physical resources.

4. Algorithm Description

The implementation of the ScanlineFlow strategy was based on the Scanline z-Buffer sequential algorithm [14] along with a smooth-interpolation shading scheme. In comparison with z-buffer algorithm, where the state information necessary for rendering a pixel is stored for every pixel on the screen, a scanline rendering algorithm pre-sorts the object database in screen space, and renders each scanline individually - only one scanline of pixel state information is kept. This is a two-pass algorithm. In the first pass, the polygons are transformed, shaded and bucket sorted to an Edge Table (ET) by the number of the first scanline on which they first become active. Then, in the second pass, the bucket-sorted list is traversed in screen-y order, maintaining an active polygon list which are, then rasterized.

The SL algorithms explore object-space parallelism: graphics primitives are assigned to processors disregarding their screen location, and each processor completely renders the primitives it is assigned. Since we are using a message-passing commercial architecture the basic software model for the implementation of the algorithms consists of two major components: the central controller and the nodes. We have implemented a symmetrical algorithm in which each node serves both as a renderer and as a compositor. Thus, the highest parallel configuration will use one processor for the central controller and 16 processors for the nodes. We will describe now the ScanlineFlow Rasterization algorithm that assumes a pipeline interconnection network for its execution.

The central controller reads in the polygonal object descriptions from disk files and then distributes them to the nodes by using the scattering method [8]. If there are P primitives and N nodes we simply assign P/N primitives to each node. This assignment is done by shuffling primitives in a round-robin fashion, which assigns the first primitive to the first node, the second to the second node, and so forth. The primitives in most databases contain some amount of geometric coherence. That is, primitives near each other in the database file, generally lie near to each other in the image as well. This means that scattering distributes coherence among the nodes; in other words, scattering distributes nearby primitives over each of the nodes.
So, the scattering method tries to minimise two sources of static load imbalances: unequal numbers of primitives on the nodes and unequal rasterization times due to the size and shape of the primitives.

After the polygon distribution by the scattering method, all nodes perform, in parallel, the first pass of the algorithm by bucket sorting their polygons into the local ET. Then, each node, in the second pass, rasterizes multiple polygons active on a given scanline after synchronisation with the previous node in order to receive the rasterization data concerning that scanline. In other words, each node only calculates the pixels in a scanline after it has stored, in the local buffers, the pixel data information from the previous node corresponding to that scanline (active and inactive pixels). This means that last node generates scanlines with the correct (final) information. The strongest point of this solution resides on the fact that, for each node, depth-comparison is performed once for each scanline.

The execution of the second pass of the algorithm through the system is pipelined in order to keep all nodes working: while a node is processing scanline y, the previous node is processing scanline y+1 and the next node is processing scanline y-1. Besides that, communication and computation are overlapped in the sense that while a node is rasterizing a scanline, it is also reading the next scanline from the previous node and forwarding the previous scanline to the next node. This scheme makes possible that the pixel rendering and the merging stages run concurrently in each node. In implementation terms, the second pass of the algorithm consists of three concurrent threads: read_scanline(), send_scanline() and render_scanline(). The read_scanline() thread scans lines from the previous node and stores them into the local buffers. The main thread, render_scanline(), calculates locally the pixel values for a scanline and performs the z-buffer algorithm by making depth comparisons between the calculated values and those that were received from the previous node. The send_scanline() thread accesses the local buffers and send information to the next node on a scanline basis. Each node has a triple-buffer, which holds three scanlines information. This way, while i position (i = 0, 1, 2) of the triple-buffer, holding information about n-1 scanline, is being accessed by render_scanline() thread, the [(i+1) mod 3] position is available to the read_scanline() thread in order to read the n scanline from the previous node and to store it. After pixel calculations for the n-1 scanline, the render_scanline() thread accesses the [(i+1) mod 3] position of the triple-buffer in order to start the pixel rendering for the n scanline. Meanwhile, the send_scanline() thread sends n-1 scanlines to the next node by accessing the i position of the buffer and read_scanline() thread is receiving n+1 scanline from the previous node and storing it into [(i+2) mod 3] position of the triple-buffer. Figure 1 illustrates the temporal diagram of the three concurrent threads activity.

![Temporal diagram of the concurrent threads activity in a node.](image)

The control and management of the triple-buffer are executed through the use of two counting semaphores and one access semaphore. The two counting semaphores are associated with the free and occupied (scanlines not processed yet) positions in the triple-buffer.

Considering an image with L scanlines, the second pass execution time of the ScanlineFlow Rasterization algorithm can be calculated by the following formula:

$$t_{exec} = \frac{1}{\text{Throughput}} \ast (L - 1) + \text{Latency} \quad (1)$$

The Throughput, number of scanlines per second, can be derived by expression 2.

$$\text{Throughput} = \frac{1}{\max (t_1, t_2)} \quad (2)$$

where

$$t_1 = \max \text{(read\_scanline\(n\), send\_scanline\(n\))}$$

$$t_2 = \max \text{(render\_scanline\(n\))}$$

The latency in a pipeline with N nodes, after figure 2 analysis, is given by the formula 3. The latency represents the elapsed time between scanline 1 rasterization start time in node 1 and rendered scanline 1 end time in last node.

$$\text{Latency} = N \ast [t_1 + t_2] \quad (3)$$

where

$$t_1 = \max \text{(read\_scanline\(1\), send\_scanline\(1\))}$$

$$t_2 = \max \text{(render\_scanline\(1\))}$$
t2 = max (render_scanline(1))

\[ t1 = \max (\text{read\_scanline}(1), \text{send\_scanline}(1)) \]

Latency = 3 \cdot [t1 + t2]

**Figure 2.** Latency of ScanlineFlow Rasterization algorithm second pass in a pipeline with 3 nodes.

In conclusion, the strongest point of this solution resides on the fact that depth-comparison is performed once for each scanline but this comes at the expense of synchronisation, which affects the throughput and the latency of the algorithm. It is very important to highlight that this algorithm implements two methods of concurrency: parallelism in the first pass and pipelining in the second pass.

### 5. Performance results

Since the rasterization phase of the graphics pipeline is the most time consuming section, our attention had focussed on it. The performance tests don’t consider the time spent on geometric transformations. We are finishing a version where a complete renderer is implemented in each node.

We assume that we are given ASCII files containing polygonal representations of 3D objects in screen space coordinates with backfaces culling performed. Another simplification made by us imposes that the objects are described as collections of triangular facets. The input scenes are rendered with a resolution of 512 x 512, producing graphics images in Portable Pixel Map (PPM) format.

Several test scenes, illustrated in Appendix A, were used:

<table>
<thead>
<tr>
<th>Scenes</th>
<th>Number of Triangles (screen coordinates)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Teapot</td>
<td>11666</td>
</tr>
<tr>
<td>Gears</td>
<td>16207</td>
</tr>
<tr>
<td>Misc I</td>
<td>16727</td>
</tr>
<tr>
<td>Misc II</td>
<td>25661</td>
</tr>
<tr>
<td>Tetra</td>
<td>34017</td>
</tr>
<tr>
<td>Misc III</td>
<td>48168</td>
</tr>
</tbody>
</table>

The Teapot, Gears and Tetra scenes were created by using the well-known Standard Procedural Databases (SPD) from Eric Haines [7].

In the algorithm execution there are three main phases:

- Node Initialisation – concerns the distribution time through the scattering method. The time to read databases from the disc was not considered.
- Edge Table Building – concerns to the first pass of the algorithm that deals with the bucket sorting of the polygons into the local Edge Table.
- Rasterization – this corresponds to the second pass of the algorithm.

The timing facilities provided by Helios are not very accurate since the clock resolution is only 10 ms. Table 1 shows the measurements of the time taken for the main stages of the algorithm in function of the number of nodes for the Misc. III scene. The Throughput item provides the number of polygons rasterized per second.

<table>
<thead>
<tr>
<th>Number of nodes</th>
<th>Node Initialisation</th>
<th>Edge Table Building</th>
<th>Rasterization</th>
<th>Throughput (tri/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>4</td>
<td>1.82</td>
<td>2.59</td>
<td>12.94</td>
<td>2576</td>
</tr>
<tr>
<td>8</td>
<td>1.48</td>
<td>0.46</td>
<td>8.51</td>
<td>4609</td>
</tr>
<tr>
<td>16</td>
<td>1.12</td>
<td>0.17</td>
<td>5.79</td>
<td>6803</td>
</tr>
</tbody>
</table>

**Table 1.** ScanlineFlow Rasterization timing chart for Misc. III scene. All times are in seconds.

The usual measure of effectiveness of a parallel algorithm is speedup, defined as the time to execute a problem on a single processor divided by the time to execute it on N processors. Efficiency is also a useful measure that gives us an indication of the utilisation of the processors in the system, and can be obtained by dividing the speedup by the number of used processors. To do that, we measured the execution times of the sequential version of the scanline z-buffer algorithm at one processor, which are shown in Table 2.
### Table 2. Uniprocessor execution times in seconds

<table>
<thead>
<tr>
<th>Scenes</th>
<th>Uniprocessor Execution time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Teapot</td>
<td>18.62</td>
</tr>
<tr>
<td>Gears</td>
<td>65.62</td>
</tr>
<tr>
<td>Misc_ I</td>
<td>28.87</td>
</tr>
<tr>
<td>Misc_ II</td>
<td>37.2</td>
</tr>
<tr>
<td>Tetra</td>
<td>52.78</td>
</tr>
<tr>
<td>Misc_ III</td>
<td>97.89</td>
</tr>
</tbody>
</table>

Figure 3 exhibits the speedup values for this algorithm.

#### ScanLine Flow

![Speedup Graph](image)

Table 3. Speedup values comparative table provided by Crockett, Ellsworth and ScanlineFlow Rasterization algorithms.

<table>
<thead>
<tr>
<th>Number of nodes</th>
<th>Crockett alg.</th>
<th>Ellsworth alg.</th>
<th>ScanlineFlow Rasterization</th>
</tr>
</thead>
<tbody>
<tr>
<td>(50000 triangles)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>NA</td>
<td>1.9</td>
<td>NA</td>
</tr>
<tr>
<td>4</td>
<td>3.8</td>
<td>3.8</td>
<td>4.6</td>
</tr>
<tr>
<td>8</td>
<td>5.1</td>
<td>6</td>
<td>8.3</td>
</tr>
<tr>
<td>16</td>
<td>12.3</td>
<td>10</td>
<td>13.9</td>
</tr>
</tbody>
</table>

#### 6. Comparison with other strategies

Now we will compare these results with those provided by Thomas Crockett [4] and David Ellsworth [5] – see Table 3. These two researchers developed two sort-middle algorithms, where the scan-conversion stage exploited the image-space parallelism. We will only analyse the speedup figures (and not the number of polygons per second) achieved by these two approaches since the used developing platforms were very powerful when compared with our Transputer-based MultiCluster². Both algorithms were based on the traditional z-buffer technique with smooth shading and synthesised 512 x 512 image resolution.

Table 3 shows that, for a number of nodes greater than 8, the ScanlineFlow Rasterization technique provides a better efficiency (87%) than the others algorithms. One explanation for this fact is due to the nature of image parallelism exploited by Crockett and Ellsworth that are very vulnerable to the scalability issue.

#### 7. Conclusions

For a general purpose distributed memory computer system, the ScanlineFlow Rasterization algorithm gave us a very important indicator about research directions for the implementation of sort-last algorithms on multicomputers: it concerns to the merging phase. The answer to the question if the rendering/merging phases are to run concurrently or consecutively was given in this paper. In fact, the SLMF-full strategy, through the ScanlineFlow Rasterization algorithm, has demonstrated to be capable to provide good results: not only the speedup values increase as more nodes are added but good efficiencies are also achieved.

---

²Crockett used an Intel iPSC/860 with 128 i860 processors organised into a hypercube network and Ellsworth used the Touchstone Delta from Caltech with 512 i860 processors organised into a 16 x 32 2D-mesh.
References


Appendix A – Colour Plates

Teapot

Misc_I

Gears

Misc_II

Tetra

Misc_III