RASTERIZAÇÃO CONCORRENTE POR FLUXO DE LINHAS: UM NOVO ALGORITMO POR COMPOSIÇÃO DE IMAGEM

João A. Madeiras Pereira[1,2]
Mário R. Gomes[1,2]

[1] INESC, R. Alves Redol 9, Apt. 13069, 1000 Lisboa, Portugal
[2]IST, Av. Rovisco Pais 4, 1000 Lisboa, Portugal

Sumário

Este artigo descreve a implementação de um algoritmo gráfico em multiprocessadores de uso-geral com comunicação por mensagens (designados vulgarmente por multicomputadores) por forma a explorar uma forma de paralelismo de granularidade alta em que o objectivo fundamental é calcular imagens completas em paralelo. Deste modo permite-se que os processadores individuais realizem os cálculos o mais assincronamente possível. Na estratégia clássica de implementação deste tipo de algoritmos, cada nó calcula uma imagem completa, com base nos polígonos recebidos e, em seguida, executa, em conjunto, com os outros nós um mecanismo de composição de imagens de modo a produzir a imagem final: cada nó transmite todos os pixels do ecrã (composição quadro-completo) aos outros nós que têm por missão resolver a questão da visibilidade. A estratégia proposta neste artigo distingue-se da anterior pelo facto de o mecanismo de composição se desenvolver durante o cálculo de imagem. A dispensa de memória local para armazenamento da informação de estado (côr e profundidade) dos pixels é o aspecto mais saliente deste novo algoritmo, o que significa que as operações de cálculo de profundidade apenas terão de ser realizadas uma única vez. O algoritmo baseia-se na paralelização do algoritmo de rasterização à linha de varrimento com *z-buffer* para remoção de superfícies ocultas e sombreamento *Gouraud*.

0. Introdução

As aproximações ao processo de síntese de imagem que recorrem principalmente à partição do Espaço Imagem (Paralelismo Imagem) padecem de duas importantes limitações:

- vulnerabilidade à duplicação de trabalho;
- pouca apetência para a extensibilidade (aumento do número de processadores).

Ambas as restrições estão relacionadas com o factor de sobreposição das primitivas já que um grande número de mosaicos de ecrã intersectados por uma primitiva geométrica provoca um aumento da redundância e agrava a necessidade de uma maior largura de banda para a comunicação.

Os algoritmos baseados na composição de imagem são, praticamente, insensíveis à redundância e revelam uma tendência natural para escalar com o aumento do número de processadores. Por outro lado, a exploração, quase exclusiva, do Paralelismo Objecto nesta classe de algoritmos implica a utilização de um modelo de programação simples: cada

processador calcula, independentemente dos outros, uma imagem com base nos polígonos recebidos, o que sugere que cada nó pode ser programado como um simples sistema gráfico completo. Claro que, a implementação destes algoritmos coloca uma questão óbvia e que é a responsável pela suas limitações: as imagens calculadas pelos processadores têm de ser combinadas por forma a produzir a imagem final. É, precisamente, esta operação de composição a geradora dos custos de comunicação que tanto penalizam esta forma de abordagem e que se agrava, por exemplo, quando se recorre à técnica de sobreamostragem (*oversampling*) para a redução do ruído.

1. Algoritmos por Composição de Imagem: uma breve panorâmica

O modelo conceptual proposto por Steven Molnar para a taxonomia dos algoritmos gráficos paralelos [Molnar94], contempla unicamente dois tipos de algoritmos por composição de imagem, **composição esparsa** e **composição quadro-completo**, os quais faziam uso do conceito de pixel activo. Diz-se que um pixel é activo, se for calculada, pelo menos, um valor de cor para a posição de ecrã correspondente àquele pixel. Os algoritmos baseados em composição esparsa apenas redistribuem pixels activos; a composição quadro-completo envolve a comunicação de todos os pixels (um ecrã completo).

Recentemente [Pereira96b], objectivando um maior refinamento, propôs um novo modelo conceptual para esta classe de algoritmos onde se contemplava a relação temporal entre os andares do bloco de rasterização - cálculo da côr dos pixels e cálculo de visibilidade (a composição). Esta subdivisão desempenha um papel pois as operações associadas a cada um desses andares são realizadas separadamente[1]. Os algoritmos pertencentes a esta classe são caracterizados pelo facto de o mecanismo de redistribuição de informação envolver pixels ou fragmentos de pixel. Por outro lado, é possível que a redistribuição dessa informação ocorra em instantes distintos:

- após cada nó ter calculado a sua imagem (parcial), o que siginifica que a **operação de composição apenas terá lugar após o cálculo de todos os pixels**;
- à medida que cada pixel é calculado, este é imediatamante enviado ao nó responsável pela sua posição no ecrã, o que significa **que a operação de composição se desenvolve durante a operação de cálculo da côr dos pixels**.

Esta distinção implica que, em termos de projecto, a primeira possibilidade recorra a memória local para retenção de informação (côr e profundidade) a ser posteriormente utilizada na composição.

Deste modo, a execução concorrente de ambos os andares respeita a sub-**classe**

[1] Geralmente na computação sequencial, o algoritmo de rasterização à medida que calcula o valor de cada pixel vai, igualmente, determinar se esse pixel irá ser visível ou não na posição do ecrã em questão.

Composição-no-Início e a execução consecutiva respeita a sub-classe **Composição-no-Fim**[2]. Qualquer das sub-classes pode empregar a redistribuição esparsa ou a redistribuição quadro-completo, tal como proposto por Steven Molnar. Assim, atingiu-se uma taxonomia em que o método de composição de imagem pode ser explorado de quatro formas:

- Composição-no-Início Esparsa (**CIE**);
- Composição-no-Início Quadro-Completo (**CIQC**);
- Composição-no-Fim Esparsa (**CFE**);
- Composição-no-Fim Quadro-Completo (**CFQC**).

A forma mais simples e directa de implementar a estratégia de Composição de Imagem num multiprocessador recorre à partição do Espaço Imagem no mecanismo de composição onde cada processador será responsável por uma determinada região do ecrã. O referido mecanismo poderá ser implementado de duas maneiras:

- à medida que os pixels são gerados localmente, são enviados aos nós que têm por tarefa resolver a questão da visibilidade naquelas posições. Trata-se de um algoritmo CIE e que foi explorado em diversas arquirecturas comerciais como [Evans92, Fuji93, Kubota93] e em sistemas de *software* [Cox95];
- cada processador calcula a sua imagem realizando cálculos de visibilidade locais e armazenando pixels numa memória de imagem local. Só então, os pixels activos de cada nó são combinados por forma a produzir a imagem final na memória de imagem distribuída. Encontramo-nos perante um esquema CFE já experimentado em [Cox93, Cox95].

Comparando as duas soluções, Michael Cox demonstrou em [Cox95] que, apesar de uma utilização pobre da memória de profundidade (útil apenas nas posições em que a profundidade é maior que 2), o tráfego de informação na rede de interligação exibia uma redução de cerca de 20% relativamente à solução CIE[3].

A arquitectura *PixelFlow* [Molnar92] é paradigmática da estratégia CFQC. O mecanismo de composição explora uma ligação entre processadores em formato *pipeline*. Após o cálculo das respectivas imagens (parciais), cada nó compõe a sua imagem com a imagem proveniente do nó anterior, enviando posteriormente o resultado para o nó seguinte. O último andar do

---

[2] A designação destas sub-classes derivou do facto de que a composição pode começar logo após o cálculo do primeiro pixel (início do processo de rasterização) - Composição-no-Início, ou começar apenas no fim do cálculo dos valores dos pixels - Composição-no-Fim.

[3] Refira-se, que em [Cox92] se formula um modelo analítico para a complexidade de profundidade o qual previa uma redução insignificante, cerca de 3%, do tráfego dos pixels activos, aquando da utilização de memória de *z-buffer*.

pipeline contém a imagem correcta. Este algoritmo CFQC, encontra-se, no findar de 1995, em fase terminal de desenvolvimento, na Universidade de North Carolina, Chapel Hill [Molnar92]. [Pereira95, Pereira96a] descrevem dois algoritmos CFQC que se distinguem pelo modo como é implementado o mecanismo de composição e propõe, ainda, um esquema para solucionar a questão do desiquilíbrio de carga nesta classe de algoritmos.

Tanto quanto se saiba, apenas o algoritmo de Rasterização Concorrente por Fluxo de Linhas, aqui descrito, se enquadra na lógica do esquema CIQC. A aproximação adoptada destina-se a sistemas de visualização sem suporte para tesselation (cenas descritas em termos de polígonos) nem sobreamostragem.

2. Rasterização Concorrente por Fluxo de Linhas de Varrimento (RCFL)

O algoritmo Rasterização Concorrente por Fluxo de Linhas de Varrimento (**RCFL**) é a versão concorrente do método sequencial de rasterização à linha de varrimento.

2.1 O algoritmo sequencial de rasterização à linha baseado em z-buffer

O tradicional algoritmo *z-buffer* funciona ao nível do polígono, ou seja, numa base em que os polígonos são rasterizados um a um. Consequentemente, o cálculo do valor final de um pixel tem de recorrer a uma memória contendo informação de estado (cor e profundidade) referente a todos os pontos do ecrã. Esta memória tem a vantagem de não depender da complexidade da cena; no entanto, tem o inconveniente da sua dimensão ser imposta pela resolução do ecrã.

Pelo contrário, um algoritmo de rasterização que efectue o seu processamento ao nível da linha de varrimento, ou seja, o cálculo de imagem é feito linha a linha, exige que a memória para armazenar a informação dos pixels tenha, apenas, a dimensão de uma linha de varrimento [Rogers85]. Em cada iteração, produz-se uma linha de pixels com a informação correcta pronta a ser enviada para o ecrã.

Os algoritmos de rasterização à linha de varrimento com coerência de aresta (**RLCA**) caracterizam-se por um funcionamento a dois passos. No primeiro passo, os polígonos são transformados geometricamente e submetidos ao processo de sombreamento nos vértices, as arestas são calculadas e colocadas numa estrutura de dados geralmente designada por Tabela de Arestas. Esta inserção é realizada pelo número de linha de varrimento na qual o polígono se torna pela primeira vez activo. No segundo passo, a Tabela de Arestas é percorrida linha a linha: para cada linha de varrimento gere-se uma Tabela de Arestas Activas (estrutura que contém as arestas intersectadas pela actual linha de varrimento) e executa-se o processo de "enchimento" dos pixels que se encontram no interior dos segmentos de recta (spans) delimitados pelas arestas activas. Na figura 1 apresenta-se o pseudo-programa que codifica o funcionamento básico deste tipo de algoritmos.

```
CalculaImagem ()
{   /* Primeiro passo: transformar, sombrear, e ordenação */
    foreach (Polígono)
    {
        Transformar (Polígono);
        if (! BackFaceCull (Polígono)) {
            SombrearVértices (Polígono);
            InserirTabelaArestas (Polígono); }
    }
    /* Segundo passo: rasterizar */
    foreach (LinhaVarrimento)
    {
        if (TabelaArestas[LinhaVarrimento] != NULL) {
            until done {
                Aresta = ObterArestas (TabelaArestas[LinhaVarrimento]);
                InserirTabelaArestasActivas (Aresta); }
        }
        if (TabelaArestasActivas != NULL)
            Rasterizar (TabelaArestasActivas, LinhaVarrimento);
        Actualizar (TabelaArestasActivas);
    }
}
```

Fig. 1 - Pseudo-código do algoritmo RLCA

Convém fazer algumas observações acerca do segundo passo do algoritmo RLCA. Atente-se, em primeiro lugar, na estrutura de dados designada por Tabela de Arestas (**TA**). O significado desta estrutura é diferente daquele que caracteriza a TA de um simples algoritmo de Rasterização de Polígonos com Coerência de Aresta (**RPCA**) [Gharachorloo89, Foley90]: enquanto que, neste último, a TA contém somente as arestas do polígono a ser processado, a TA de um algoritmo RLCA contém informação de todas as arestas de todos os polígonos. Deste modo, ao ser inserida uma aresta nesta TA de acordo com o valor da ordenada mínima, há que providenciar, para além dos tradicionais parâmetros (a abcissa mínima e a ordenada máxima da aresta, o inverso do declive e os incrementos na cor e profundidade), um outro campo que contenha um identificador do polígono a que a aresta pertence. Por outro lado, a estrutura de dados que contém as arestas activas numa determinada linha de varrimento, a Tabela de Arestas Activas (**TAA**), está organizada em grupos de arestas que correspondem aos polígonos activos, ou seja, cada grupo possui as arestas activas pertencentes a um mesmo polígono. Assim, o processo de preenchimento dos spans, com *z-buffer* e sombreamento *Gouraud* [Gouraud71], não é mais do que uma simples repetição da operação de preenchimento do algoritmo RPCA para cada um desses grupos.

2.2 Descrição do algoritmo paralelo

Considerando o facto de se utilizar uma arquitectura distribuída comercial baseada em comunicação de mensagens, o algoritmo implementado adoptou o modelo funcional em que um processador assume funções de **controlador central** e os outros processadores, denominados de **nós**, desempenham uma dupla função: a de **calculadores de imagem** e a de **compositores de imagem**.

O controlador central, após a leitura em disco da base de dados da cena, distribui os polígonos pelos nós do sistema recorrendo ao esquema de *scattering* (primeiro polígono ao primeiro nó, segundo polígono ao segundo nó, terceiro polígono ao terceiro nó e assim sucessivamente), a fim de providenciar estaticamente algum balanceamento de carga.

Os nós são responsáveis pela síntese da imagem final. Para isso, cada nó inicia o seu processamento através da execução do primeiro passo do algoritmo RLCA sobre o conjunto de polígonos recebidos. Trata-se de uma forma de trabalho Paralelo-Objecto em que os nós processam em paralelo primitivas geométricas. O segundo passo, ou seja a rasterização, começa após todos os nós terem inserido devidamente os polígonos nas respectivas Tabela de Arestas locais. Na implementação do segundo passo reside um esquema único de concorrência que permite sobrepor as actividades de "preeenchimento" de pixels e de composição, e que se baseia numa topologia *pipeline* de interligação dos nós: cada nó só rasteriza os múltiplos polígonos numa linha de varrimento após sincronização com o nó prévio com o intuito de receber a sua informação referente àquela linha de varrimento. Por outras palavras, um nó apenas calcula os valores dos pixels numa linha quando já estiver na posse dos valores desses mesmos pixels gerados pelo nó anterior, o que significa que o último nó na rede *pipeline* produz nessa linha os valores finais correctos, os quais são enviados para o controlador[4]. Cabe a este, a tarefa de receber as linhas produzidas pelo último nó e armazená-las na sua memória de imagem. Sublinhe-se que a informação de linha que um nó recebe do andar anterior contém todos os pixels, activos e inactivos, daí nos encontrarmos perante um esquema quadro-completo na redistribuição dos pixels que caracteriza um algoritmo Ordenação-no-Fim. A fim de manter todos os nós activos, utilizou-se um esquema *pipelined* para o fluxo das linhas de varrimento: enquanto um nó processa a linha de varrimento y, o nó prévio processa a linha y+1 e o nó seguinte processa a linha y-1.

Resumindo, o algoritmo Rasterização Concorrente por Fluxo de Linhas de Varrimento pauta-se pelo exercício das duas formas de concorrência no seu funcionamento: paralelismo durante a execução do primeiro passo, e *pipelining* durante a execução do segundo passo. O aspecto mais marcante desta abordagem centra-se no facto de que os cálculos de profundidade são efectuados uma única vez em cada linha de varrimento; no entanto esta vantagem é adquirida à custa de actividade de sincronização que penaliza o desempenho global do sistema.

O segundo passo do algoritmo, a rasterização, compõe-se de três processos concorrentes. O objectivo é explorar a capacidade de sobreposição das actividades de comunicação e computação, no sentido de que enquanto um nó está rasterizando uma linha, pode simultâneamente ler e enviar informação para os nós adjacentes. Assim, existe o processo **lê_linha()** que é responsável por ler e armazenar informação correspondente a linhas de varrimento provenientes do nó anterior. O processo principal **rasterizar_linha()** tem por

[4] No ideal, se o ritmo de saída das linhas no último nó fosse igual ao ritmo tempo real, as linhas poderiam ser imediatamente enviadas para o ecrã.

missão calcular os pixels individualmente em cada linha de varrimento, resolvendo a questão da visibilidade através da aplicação do algoritmo *z-buffer* entre os valores calculados e aqueles que já foram recebidos do nó anterior, sendo o resultado guardado num buffer para posterior envio ao nó seguinte. O processo **envia_linha()** acede a esse *buffer* local do nó e transmite, linha a linha, a informação para o nó seguinte. Os processos de recepção e envio de linhas são marcados por uma actividade intensa de comunicação, enquanto que o processo principal rasterizar_linha() tem uma acção predominantemente de computação.

Cada nó possui um *buffer* triplo de entrada, **buff**, onde é armazenada a informação correspondente a três linhas de varrimento. Assim, enquanto a posição i (i = 0,1,2) da variável buff, contendo a linha **n-1**, está sendo acedida pelo processo rasterizar_linha(), a posição [(i+1) mod 3] está disponível para o processo lê_linha() receber do nó anterior a próxima linha de varrimento **n** e guardá-la. Após o fim do cálculo dos valores dos pixels referentes à linha **n-1**, o processo rasterizar_linha() inicia imediatamente a sua acção com a linha de varrimento n através do acesso à posição [(i+1) **mod** 3] do buffer triplo. Entretanto, o processo envia_linha() acede à posição i para transmitir ao nó seguinte a linha n-1 e o processo lê_linha() recebe, na outra posição da variável buff, [(i+2) **mod** 3], a linha seguinte **n+1**. A figura 2 exibe o diagrama temporal associado aos fluxos de actividade dos três processos.

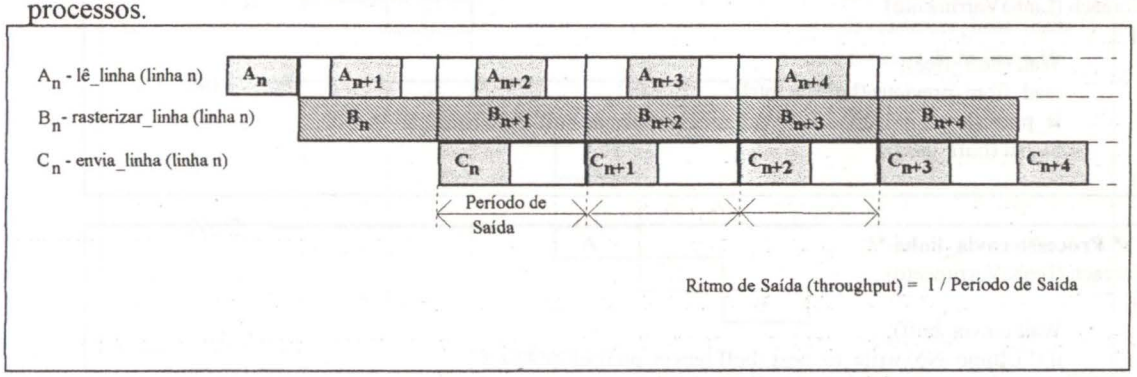

Fig. 2 - Diagrama temporal referente à execução do algoritmo RCFL num nó

O controlo e gestão da variável buff é feito por dois semáforos contadores **buff_free** e **buff_used** e um semáforo de acesso **envia_buff**. O semáforo buff_free controla a disponibilidade de posições livres e o buff_used controla a existência de posições ocupadas com informação ainda não processada. O semáforo envia_buff controla o acesso à variável buff por parte do processo envia_linha(). Esse semáforo é desbloqueado por uma instrução *Signal* no processo rasterizar_linha(). Consoante o processo que pretende aceder à variável buff, existem três ponteiros de acesso: **le_ptr**, **scan_ptr** e **envia_ptr**.

Na figura 3, apresenta-se os pseudo-programas que codificam os três processos concorrentes que determinam o funcionamento do algoritmo RCFL.

```
/* Processo rasterizar_linha */
foreach (LinhaVarrimento)
{
        if (TabelaArestas[LinhaVarrimento] != NULL)
        {
            until done
            {
                Aresta = ObterArestas (TabelaArestas[LinhaVarrimento]);
                InserirTabelaArestasActivas (Aresta);
            }
        }
        Wait (buff_used);
        if (TabelaArestasActivas != NULL)
            Rasterizar (TabelaArestasActivas, LinhaVarrimento, buff [scan_ptr]);
        inc_scan_ptr (scan_ptr); /* próxima posição de buff para ser rasterizada */
        Signal (envia_buff);
        if (TabelaArestasActivas != NULL) Actualizar (TabelaArestasActivas);
}
```

```
/* Processo lê_linha */
foreach (LinhaVarrimento)
{
        Wait (buff_free);
        read_from_previous (buff [le_ptr]);
        le_ptr = (le_ptr + 1) mod 3; /* próxima posição de buff para recepção */
        Signal (buff_used);
}
```

```
/* Processo envia_linha */
foreach (LinhaVarrimento)
{
        Wait (envia_buff);
        if (! Último_Nó) write_to_next (buff [envia_ptr]);
        else write_to_controller (buff [envia_ptr]);
        envia_ptr = (envia_ptr + 1) mod 3; /* próxima posição de buff para envio */
        Signal (buff_free);
}
```

Fig. 3 - Pseudo-código dos três processos concorrentes (2° passo do algoritmo RCFL)

É fácil concluir que o tempo de execução do algoritmo RCFL pode ser calculado por

$$t_{exec} = \frac{1}{Ritmo\ de\ Saída} * (n^{o}\ linhas - 1) + Latência \qquad (1)$$

em que o *Ritmo de Saída* vale

$$\text{Ritmo de Saída} = \frac{1}{\max(t1, t2)} \qquad (2)$$

onde

t1 = max (lê_linha (linha n), envia_linha (linha n))

t2 = max (rasterizar_linha (linha n))

Quanto à Latência, ou seja, o tempo que medeia entre o início da rasterização e o aparecimento da primeira linha de varrimento no controlador, depois de analisada a figura 4, conclui-se que num sistema com P nós:

$$Latência = P * [t1 + t2] \qquad (3)$$

onde :

t1 = max (lê_linha (linha 1), envia_linha (linha 1))

t2 = max (rasterizar_linha (linha 1))

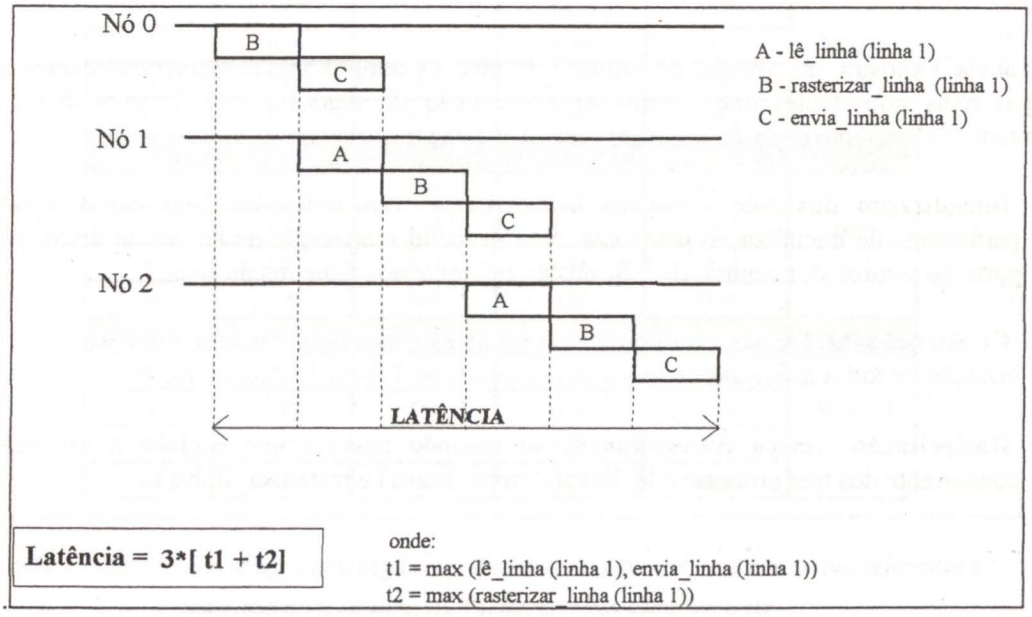

Fig. 4 - Latência do algoritmo RCFL (três nós)

2.3 Resultados experimentais

A implementação do primeiro passo do algoritmo RCFL contemplou apenas a construção da Tabela de Arestas. O objectivo era focar a actividade de rasterização já que esta é a maior consumidora de ciclos CPU. O bloco de rasterização respeitava o processamento de triângulos com *z-buffer* para remoção de superfícies ocultas (16 bits de profundidade) e sombreamento por interpolação linear de intensidades (24 bits para a cor).

Os testes decorreram na plataforma da Parsytec [Parsytec90], o MultiCluster-2, com 17 *transputers* T-800 a 20 MHz [Inmos87] e utilizando o sistema de exploração HELIOS. As imagens foram sintetizadas com uma resolução 512 x 512 a partir de simples ficheiros de entrada ASCII e guardadas em ficheiros com o formato ppm (*Portable Pixel Map*).

Recorreram-se a várias cenas de teste cujas imagens se encontram representadas nas ilustrações deste artigo, e cuja complexidade abaixo se discrimina:

Imagens	**Número de Triângulo** (Coordenadas de Ecrã)
Teapot	11666
Gears	16207
Misc_I	16727
Misc_II	25661
Tetra	34017
Misc_III	48168

A Tabela 1 mostra, em função do número de nós, os tempos parciais correspondentes às etapas mais importantes que constituem a execução do algoritmo RCFL para diversas imagens. Assim, podem-se distinguir as seguintes etapas:

- **Inicialização dos Nós** - respeita a distribuição dos polígonos bem como alguns parâmetros de inicialização pelos nós. Não se inclui a operação de leitura de disco, por parte do controlador central, da informação referente aos ficheiros de cena;
- **Construção da TA** - etapa referente ao primeiro passo do algoritmo e que lida com a inserção de todas as arestas de todos os polígonos na Tabela de Arestas local;
- **Rasterização** - etapa correspondente ao segundo passo e que engloba a execução concorrente dos três processos: lê_linha(), envia_linha() e rasterizar_linha().

Número de Nós	Inicialização dos Nós	Construção da TA	Rasterização	Throughput (tri/s)	Imagens
2	0.42	1.41	10.01	985	Teapot
4	0.46	0.58	7.22	1412	
8	0.39	0.18	6.2	1723	
16	0.34	0.18	5.59	1909	
2	0.57	1.94	29.87	500	Gears
4	0.63	0.94	16.76	884	
8	0.55	0.18	13.06	1175	
16	0.44	0.18	11.15	1376	
2	0.58	1.96	15.74	915	Misc_I
4	0.71	1.03	12.16	1203	
8	0.56	0.18	11.45	1372	
16	0.42	0.18	10.84	1462	
2	1.08	3.11	14.71	1357	Misc_II
4	0.98	1.39	8.58	2343	
8	0.85	0.2	6.32	3481	
16	0.65	0.17	4.93	4462	
2	1.16	3.88	17.04	1540	Tetra
4	1.28	1.93	8.33	2947	
8	1.02	0.34	6.16	4523	
16	0.83	0.18	4.53	6140	
2	ND	ND	ND	ND	Misc_III
4	1.82	2.59	12.94	2776	
8	1.48	0.46	8.51	4609	
16	1.12	0.17	5.79	6803	

Tabela 1 - Quadro dos tempos (em segundos) dispendidos em cada uma das etapas referentes à execução do algoritmo RCFL em diversas cenas de teste.

É possível antecipar, dada a quantidade de processamento envolvida, que a actividade desempenhada pelo processo rasterizar_linha() domina a etapa de rasterização sendo a principal responsável pelo ritmo de saída do *pipeline*. Isto significa que a execução do algoritmo é fortemente influenciada por dois tipos de actividade que caracterizam o funcionamento do processo rasterizar_linha(): computação e sincronização (resultante dos eventuais bloqueamentos no semáforo buff_used). Esta previsão é confirmada experimentalmente, como se poderá observar na figura 5. No gráfico desta figura, indica-se, cumulativamente, a percentagem de tempo dedicada à rasterização da cena *Teapot* e as respectivas percentagens relativas às actividades de computação e sincronização

desempenhadas pelo processo rasterizar_linha. Sublinhe-se que estas duas percentagens se baseiam nas médias dos respectivos tempos obtidos nos nós.

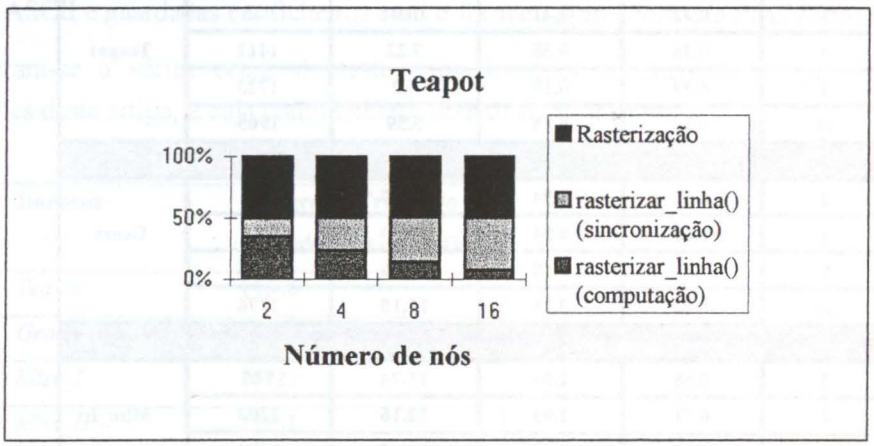

Fig. 5 - Tempo de rasterização versus tempo de computação e sincronização (execução do processo rasterizar_linha()). As percentagens são representadas cumulativamente.

Uma análise gráfica da evolução, em função do número de nós, das quantidades totais de computação e sincronização (**CT** e **ST**) referentes à soma dos respectivos tempos obtidos pela execução do processo rasterizar_linha() em todos os nós, - figura 6 - indicia que o tempo de execução do algoritmo RCFL é marcado, fundamentalmente, por uma componente paralela de computação, traduzida em termos práticos por curvas praticamente contantes, e por uma componente explícita não-paralela de sincronização cujas curvas têm um crescimento bastante acentuado. Esta conclusão, pode ser, igualmente, retirada da figura 5.

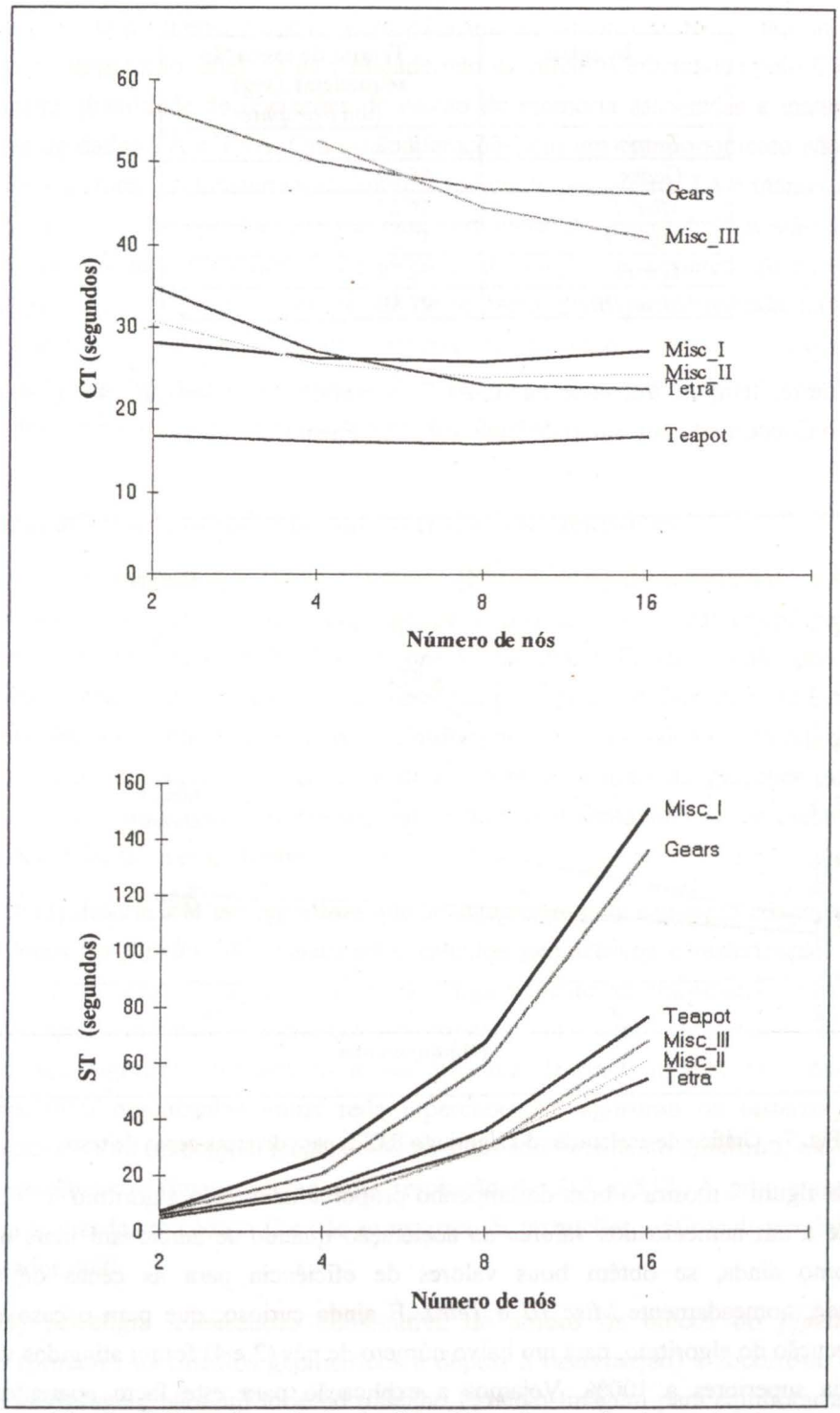

Fig. 6 - Gráficos da evolução de CT e ST em função do número de nós.
(ST - Soma dos tempos de sincronização obtidos nos nós;
CT - Soma dos tempos de sincronização obtidos nos nós)

Na avaliação do desempenho do algoritmo RCFL, os tempos de execução sequencial utilizados no cálculo da aceleração e da eficiência respeitam a execução do algoritmo RLCA num único *transputer*. Os tempos obtidos referentes às imagens usadas nos testes são os

seguintes:

Imagens	Tempo de execução sequencial (seg) (um *transputer*)
Teapot	18.62
Gears	65.62
Misc_I	28.87
Misc_II	37.2
Tetra	52.78
Misc_III	97.89

Com base nestes tempos, traçou-se na figura 7, o gráfico de aceleração para o algoritmo Rasterização Concorrente por Fluxo de Linhas de Varrimento.

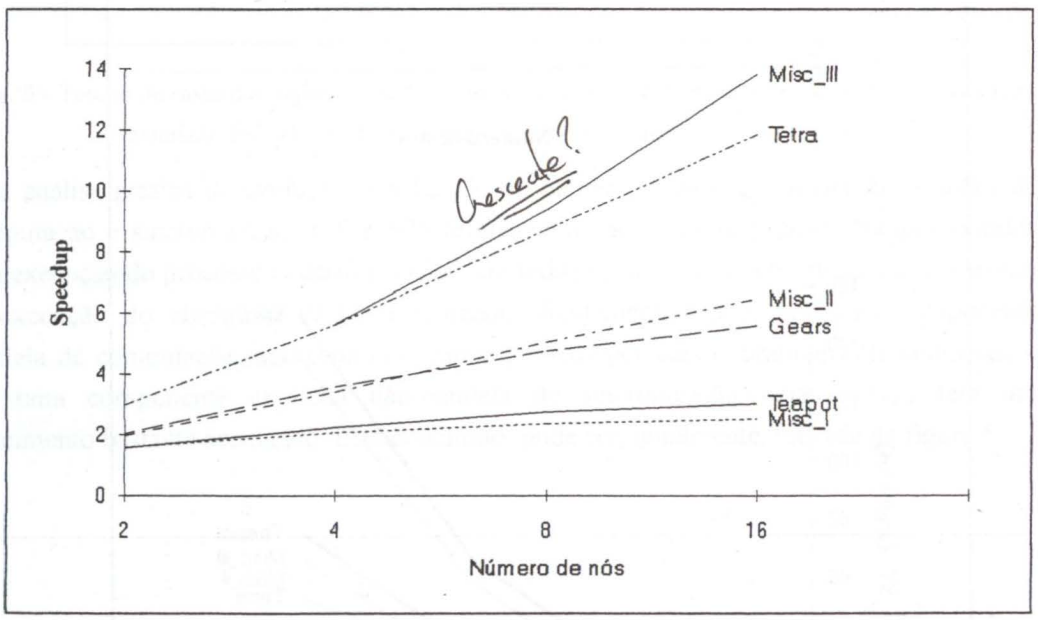

Fig. 7 - Gráfico de aceleração do algorimto RCFL para diversas cenas de teste

O gráfico da figura 7 mostra o bom desempenho proporcionado pelo algoritmo RCFL: não só se assiste a um aumento nos valores da aceleração quando se adicionam mais nós ao sistema, como ainda, se obtêm bons valores de eficiência para as cenas de maior complexidade, nomeadamente *Misc_III* e *Tetra*. É ainda curioso, que para o caso destas cenas, a execução do algoritmo, para um baixo número de nós (2 e 4) foram atingidas valores de eficiência superiores a 100%. Vejamos a explicação para este facto aparentemente insólito. O cálculo tradicional da eficiência baseia-se, antes de mais, na assunção de que, na ausência de comunicação, a computação (cálculos efectuados pelo CPU) tem um comportamento linear, ou seja, se um determinado problema envolvendo apenas cálculos demora t na execução sequencial, então demorará t/4 num sistema com quatro processadores. A existência de comunicação é a causa geralmente apontada para que as eficiências se situem

abaixo dos 100%. O problema é que no caso particular do algorimto RCFL, aquilo que foi designado por computação, engloba na realidade não só cálculos efectuados pelo CPU mas uma substancial quantidade de operações de gestão de memória associadas à manipulação das estruturas de dados TA e TAA. Ora estas operações têm um comportamento não-linear, em termos de aceleração. Na rasterização, em que se tem de percorrer a TA e manter a TAA, a questão desta não-linearidade torna-se bastante pertinente: por exemplo, a gestão de 1000 polígonos nessas estruturas implica mais que o dobro do tempo que se obtém quando se gere 500 polígonos. A confirmação experimental deste facto pode ser observada na rubrica "Construção da TA" na Tabela 1, em que as operações de acesso à memória são em menor quantidade relativamente às que ocorrem na rasterização mas em que já se vislumbra um comportamento não-linear quando o número de nós aumenta.

3. Comparação com resultados de outros trabalhos científicos

Thomas Crockett [Crockett93] e David Ellsworth [Ellsworth94] demonstraram, através do recurso ao método de partição no Espaço Imagem, que a síntese "em tempo real" era exequível em multicomputadores. No entanto, apesar das altas taxas de desenho produzidas pelos algoritmos, que se devem em grande parte à configuração de *hardware* utilizada, os valores de eficiência reportados eram baixos. Confrontar-se-á estes valores com aqueles que foram obtidos pela execução do algoritmo RCFL com o sentido de perceber qual das estratégias se perfila como a mais promissora em termos de desempenho e o de esclarecer as causas para determinados comportamentos.

Em [Crockett93] descreve-se um algoritmo que se caracteriza pela execução concorrente de ambos os blocos do *pipeline* de visualização, cálculos geométricos e rasterização, e pela utilização de um esquema de balanceamento de carga baseado na amostragem uniforme e coerente do Espaço Imagem (o ecrã era decomposto em "fatias" horizontais de igual tamanho). O seu algoritmo foi testado numa máquina da Intel, o *iPSC/860*, com 128 processadores i860 organizados numa rede hipercubo. O algoritmo de rasterização de triângulos baseou-se na tradicional técnica *z-buffer* com sombreamento *Gouraud*, e os testes respeitavam a síntese de imagens com uma resolução de 512 x 512. A programação do algoritmo foi realizada em código C e não se recorreu às instruções gráficas disponibilizadas pelo processador i860.

[Ellsworth94] priveligia a execução consecutiva de ambos os blocos do *pipeline* de visualização (primeiro os cálculos geométricos e depois a rasterização) e socorre-se de um esquema de amostragem não-uniforme no trabalho Paralelo-Imagem para equilibrar a carga de processamento: explorando a coerência de frame, as partições do ecrã (de igual tamanho) são mapeadas nos processadores com base em estimativas previamente realizadas na frame anterior. Este algoritmo foi experimentado na plataforma *Touchstone Delta* da *Caltech*, a qual comporta 512 processadores *i860* da Intel organizados numa malha 2D de dimensão 16 x 32. A rotina de rasterização de triângulos (*z-buffer* com sombreamento *Gouraud* e sem

anti-*aliasing*) foi escrita em linguagem *assembly i860* e utilizou as instruções gráficas deste processador. As imagens das cenas de teste foram sintetizadas com uma resolução de 640 x 512.

São por demais evidentes as diferenças (número de processadores, a potência de cálculo de cada processador, serviço de comunicações, etc.) entre as máquinas utilizadas por aqueles investigadores e o MultiCluster-2 onde foi testado o algoritmo descrito neste artigo pelo que os valores obtidos para as taxas de desenho (triângulos/segundo) estão completamente desfasados (superior a 100000 para as máquinas baseadas no *i860* da Intel) . A comparação deverá ser efectuada com base nos valores de aceleração publicados já que estas grandezas traduzem o verdadeiro desempenho de um algoritmo paralelo. Dado que as cenas utilizadas nos respectivos testes eram diferentes, optou-se por utilizar cenas em que o número de primitivas fosse muito semelhante, com o objectivo de uniformizar as condições de estabelecimento de um quadro comparativo. Assim, na Tabela 2 indica-se entre parêntesis o número de triângulos envolvidos nos respectivos testes.

Número de Nós	**Aceleração no alg. Crockett** (50000 triângulos)	**Aceleração no alg. Ellsworth** (59592 triângulos)	**Aceleração no alg. RCFL** (cena *Misc_III*: 48168 triângulos)
2	não disponível	1.9	não disponível
4	3.8	3.8	4.6
8	5.1	6	8.3
16	12.3	10	13.9

Tabela 2 - Quadro comparativo dos valores de aceleração reportados pelos algoritmos descritos em [Crockett93] e [Ellsworth94] e pelo algoritmo RCFL

Desta tabela ressalta os superiores valores para a aceleração que foram alcançados pela execução do algoritmo RCFL a que correspondem valores elevados de eficiência. É interessante mencionar que por outro lado, o comportamento exibido pelos algoritmos de Crockett e Ellsworth degrada-se a partir de um número de processadores superior a 32 como é referido nas conclusões dos respectivos artigos: no caso do algoritmo de Crockett a eficiência com 32 nós é de 60% e com 64 nós é de 40% enquanto que, no caso do algoritmo de Ellsworth, a situação piora com cerca de 50% de eficiência para 16 nós e apenas 31% para a configuração com 64 nós. A explicação destes baixos valores de eficiência prende-se precisamente com a pouca apetência para a extensibilidade que é uma das características negativas decorrente do Paralelismo. Esta restrição está relacionada com o factor de sobreposição das primitivas já que um grande número de mosaicos de ecrã intersectados por uma primitiva geométrica provoca um aumento da redundância e agrava a necessidade de uma maior largura de banda para a comunicação. Em [Ellsworth94] verifica-se ainda que, para um determinado número de nós e com algumas cenas mais complexas, a eficiência baixava igualmente devido ao maior custo de comunicação envolvido na redistribuição das primitivas de visualização.

Infelizmente, o MultiCluster-2 só possui 16 processadores pelo que uma análise para um

número superior de nós apenas se pode basear numa extrapolação dos resultados observados. Nestas condições, o algoritmo RCFL deixa antever propensão para a extensibilidade (cena *Misc_III*): com dois nós a eficiência é superior a 100% (razões explicitadas anteriormente), com dezasseis nós a eficiência é de cerca de 87% pelo que é expectável valores de eficiência superiores a 60% para 32 e 64 nós. Apenas, por alguma razão não detectada na explicação do funcionamento do algoritmo RCFL poder-se-ia considerar um valor de tal modo elevado no fenómeno da sincronização que conduzisse a valores de eficiência abaixo dos 50%.

4. Conclusões

Alguns investigadores mostraram que a síntese de imagem "em tempo real" era exequível em máquinas MIMD com memória distribuída [Crockett93, Ellsworth94] e memória partilhada [Whitman94]. Todas estas soluções baseiam a sua implementação na partição do Espaço Imagem. Estes algoritmos fazem recurso a um paralelismo com um nível de granularidade alto pois o seu projecto centra-se na partição do ecrã e nos esquemas de mapeamento dessas zonas aos processadores de modo a minimizar o desiquilíbrio de carga. Por isso, estas aproximações revelam vulnerabilidade à duplicação de trabalho e pouca apetência para a extensibilidade. Ambas as restrições estão relacionadas com o factor de sobreposição das primitivas já que um grande número de mosaicos de ecrã intersectados por uma primitiva geométrica provoca um aumento da redundância e agrava a necessidade de uma maior largura de banda para a comunicação. Nos multicomputadores, estes problemas reflectiram-se nos resultados experimentais que mostraram, apesar dos valores elevados para a taxa de desenho (as máquinas também eram rápidas), possuir eficiências baixas e serem penalizados quando da execução dos respectivos algoritmos sobre cenas mais complexas. O algoritmo Rasterização Concorrente por Fluxo de Linhas (RCFL), em que a etapa de composição era realizada concorrentemente com a rasterização, demonstrou ser mais eficaz que essas soluções na obtenção de melhores desempenhos. Na realidade, o algoritmo RCFL exibiu propensão, não só, para a extensibilidade (a aceleração aumenta face a um incremento no número de nós), mas também, para a obtenção de valores razoáveis para a eficiência, em particular muito bons no caso das imagens mais complexas. O sucesso deste algoritmo face aos algoritmos CFQC permitiu igualmente fornecer, no caso particular de uma máquina com um número de moderado de processadores (igual ou inferior a dezasseis), uma resposta clara à questão de saber se os andares de rasterização e composição devem ser executados consecutiva ou concorrentemente. A razão justificativa do bom desempenho do algoritmo RCFL localiza-se precisamente no andar de composição pois, apesar de linhas completas serem comunicadas entre processadores adjacentes, só as posições correspondentes aos pixels calculados por um nó vão estar envolvidas no cálculo da profundidade, ao contrário do que sucede com a estratégia CFQC cuja resolução desse problema implica testar todos os pixels do ecrã após o cálculo das imagens parciais.

5. Referências bibliográficas

[Cox93] Michael Cox, Pat Hanrahan, "Pixel Merging for Object-Parallel Rendering: A Distributed Snooping Algorithm", *Proc. Parallel Rendering Symp.*, ACM Press, New York, 1993, pp. 49-56.

[Cox95] Michael Cox, *Algorithms for Parallel Rendering,* Tese de Doutoramento, Princeton University, Maio 1995.

[Crockett93] T. W. Crockett, T. Orloff, "A Parallel Rendering Algorithm for MIMD Architectures", *Proc. Parallel Rendering Symp.*, ACM Press, New York, 1993, pp. 35-42.

[Ellsworth94] David Ellsworth, "A New Algorithm for Interactive Graphics on Multicomputers", *IEEE Computer Graphics and Applications*, Vol. 14, Nº 4, Julho 1994, pp. 33-40.

[Evans92] Evans and Sutherland Computer Corporation, *Freedom Series Tehnical Information*, Outubro 1992.

[Foley90] J. D. Foley, A. van Dam, S. Feiner, J. Hughes, *Computer Graphics Principles and Practice*, Second Edition, Addison-Wesley Publishing Company, 1990.

[Fuji93] Fujitsu Limited, "AG Series Graphics Technical Overview", Fujitsu Open Systems Solutions, 1993.

[Gharachorloo89] N. Gharachorloo, S. Gupta, R. F. Sproull, I. E. Sutherland, "A Characterization of Ten Rasterization Techniques", *Computer Graphics* (Proc. Siggraph), Vol. 23, Nº 3, Julho 1989, pp. 355-368.

[*Gouraud*71] H. *Gouraud*, "Continuous Shading of curved surfaces", *IEEE Transactions on Computers*, C-20(6), 623-628, 1971.

[Inmos87] INMOS Limited, "*Preliminary Data: IMS T800 Transputer*", Abril 1987.

[Kubota93] Kubota Pacific Computer, *Denali Technical Overview*, versão 1.0, Março 1993.

[Molnar92] S. Molnar, J. Eyles, and J. Poulton, "PixelFlow: High-Speed Rendering Using Image Composition", *Computer Graphics* (Proc. Siggraph), Vol. 26, Nº 2, Julho 1992, pp. 231-240.

[Molnar94] S. Molnar, M. Cox, D. Ellsworth, H. Fuchs, "A Sorting Classification of Parallel Rendering", *IEEE CG&A*, Vol. 14, Nº 2, Julho 1994, pp 23-32.

[Parsytec90] PARSYTEC GMbH, "MultiCluster-2: Technical Documentation",

Rev. 1.1, Maio 1990.

[Pereira95] João Pereira, C. Wuthrich, M. Gomes, "Implementing Sort-Last Algorithms for Polygon Rendering on a Multicomputer", Technical Report RT12/95, Inesc, Maio 95.

[Pereira96a] João Pereira, C.A. Wüthrich, M. R. Gomes, "Full-Frame Merging for Sort-Last Polygon Rendering on a Multicomputer", *The Fourth International Conference in Central Europe on Computer Graphics and Visualization '96* (WSCG '96), 12-16 Fevereiro 1996.

[Pereira96b] João Pereira, Cinerealismo em Arquitecturas Paralelas de Uso Geral, Tese de Doutoramento, Instituto Superior Técnico (Escola da Universidade Técnica de Lisboa), Julho de 1996.

[Rogers85] David F. Rogers, "Procedural Elements for Computer Graphics", *McGraw-Hill Int. Publications,* 1985.

[Whitman94] Scott Whitman, "Dynamic Load Balancing for Parallel Polygon Rendering", *IEEE Computer Graphics and Applications*, Vol. 14, N° 4, Julho 1994, pp. 41-48.