2
Trabalhos Relacionados

Neste capítulo é feita uma recapitulação dos trabalhos relacionados às áreas de renderização em tempo real e de renderização distribuída. São apresentados uma classificação de arquiteturas de renderização distribuída e os modelos propostos para a distribuição dos dados entre os nós do agrupamento.

2.1
Classificação de Renderização Distribuída

Molnar et al. [7] classificaram as arquiteturas de renderização distribuída com base no estágio da linha de produção da renderização em que ocorre a ordenação de visibilidade. Três classes de arquiteturas foram concebidas com base neste critério: arquiteturas com ordenação no início (*Sort-First*), no meio (*Sort-Middle*) e no fim (*Sort-Last*).

Na arquitetura com ordenação no início, a tela é dividida em um conjunto de ladrilhos disjuntos (normalmente com forma retangular), que são distribuídos aos processadores gráficos. A distribuição de trabalho é realizada num estágio inicial da linha de produção gráfica: a distribuição de primitivas geométricas entre os processadores, a qual é feita tendo em vista qual pedaço da tela a projeção de cada primitiva intersecta. Cada processador gráfico é responsável por toda a computação que afeta o pedaço da tela que ele está renderizando. No caso de sistemas com um único dispositivo de saída, os resultados da renderização de cada ladrilho devem ser enviados para o computador responsável por mostrá-los na tela. Como os ladrilhos não se intersectam e já contêm o resultado final da renderização, a composição dos resultados parciais é direta: basta mostrá-los na tela.

As vantagens dessa arquitetura são os baixos requerimentos de comunicação entre os processadores e o fato de que cada processador implementa a linha de produção de renderização inteira para uma porção da tela, o que possibilita o uso de transparência. No entanto, essa arquitetura é sus-

cetível a desbalanceamento de carga devido à possível distribuição desigual de primitivas sobre a tela. Além disso, uma primitiva deve ser desenhada por todos os ladrilhos que a intersectam, o que limita a escalabilidade desta técnica quando passa-se a dividir a tela entre muitos processadores. Um diagrama dessa arquitetura encontra-se na Figura 2.1.

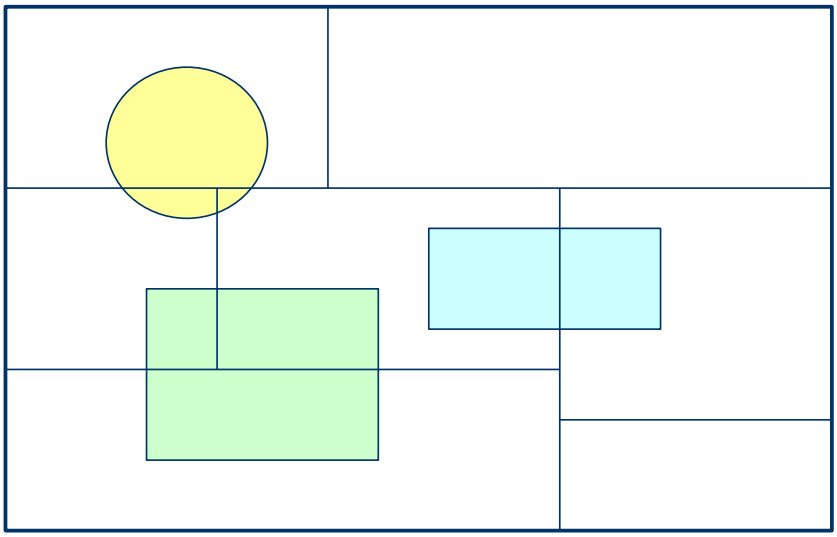

Figura 2.1: Diagrama das arquiteturas com ordenação no início. Cada processador fica responsável pelo desenho em um pedaço da tela.

Na arquitetura com ordenação no meio, a geometria a ser desenhada é particionada entre um conjunto de processadores de geometria. Após as primitivas serem transformadas para o espaço de tela (entre o estágio da geometria e o estágio da rasterização), elas são redistribuídas entre processadores de rasterização. O critério para a distribuição das primitivas no espaço de tela é igual ao das arquiteturas com ordenação no início: a tela é dividida em ladrilhos disjuntos, sendo cada um destes atribuído a um processador de rasterização. Como nas arquiteturas com ordenação no início, após cada processador de rasterização desenhar todas as primitivas do seu pedaço da tela, este está pronto para ser exibido.

Essa arquitetura tem a vantagem de fazer a redistribuição de primitivas em um ponto natural da linha de produção gráfica. Entretanto, devido à necessidade de uma grande largura de banda de comunicação entre os processadores de geometria e de rasterização, esta arquitetura atualmente só é possível em estações de trabalho especializadas. O desbalanceamento de carga proveniente da distribuição desigual de primitivas sobre a tela também é uma desvantagem, assim como nas arquiteturas com ordenação no início. Essa arquitetura está ilustrada na Figura 2.2.

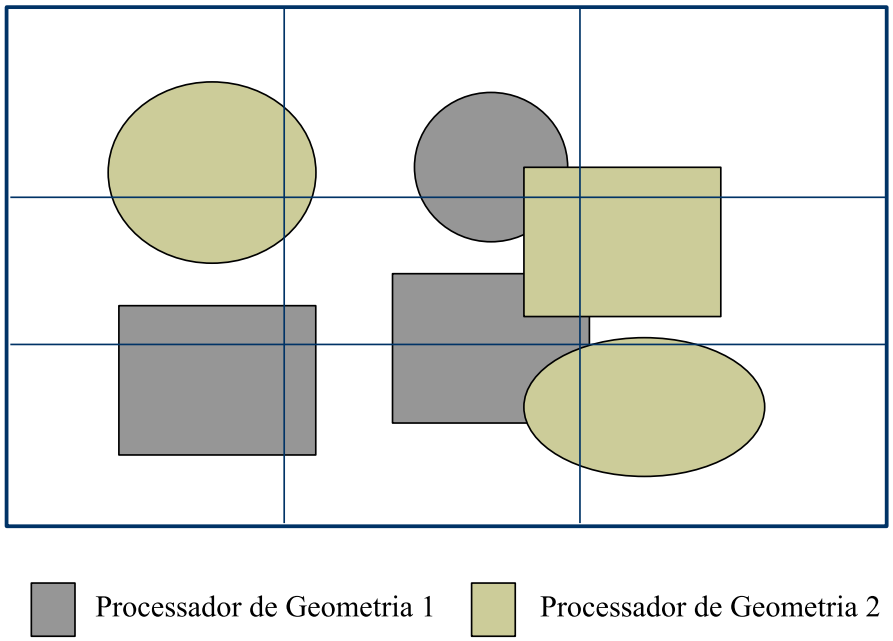

Figura 2.2: Diagrama das arquiteturas com ordenação no meio. Cada processador de geometria fica responsável por um pedaço do modelo e cada processador de rasterização fica responsável por um pedaço da tela.

Já na arquitetura com ordenação no fim, a resolução de visibilidade é transferida para o fim da linha de produção gráfica. A parte do modelo a ser desenhada é dividida igualmente entre os processadores. Cada processador renderiza suas primitivas completamente (geometria e rasterização). Após esse estágio, a parte da tela em que o processador desenhou primitivas é enviada a um processador de composição, que resolve a visibilidade de cada fragmento com base nas sub-imagens de todos os processadores. Para que a resolução de visibilidade seja possível em outro processador, devem ser enviadas as informações de cor e de profundidade, no caso de modelos baseados em primitivas geométricas, ou de componente *alpha*, no caso de composição com *alpha-blending*.

Como cada primitiva do modelo é desenhada por apenas um processador, essa arquitetura possibilita ótima escalabilidade em termos do número de primitivas desenhadas. Porém, como é necessária a transmissão de um número de buffers de cor e visibilidade proporcional ao número de processadores, a escalabilidade dessa arquitetura fica fortemente limitada pela largura de banda da rede que conecta os processadores do agrupamento. Além disso, se o estágio da composição for feito por apenas um processador, é possível que este seja sobrecarregado, diminuindo o desempenho da aplicação. Essa arquitetura está representada na Figura 2.3

Na tentativa de combinar as vantagens das arquiteturas com or-

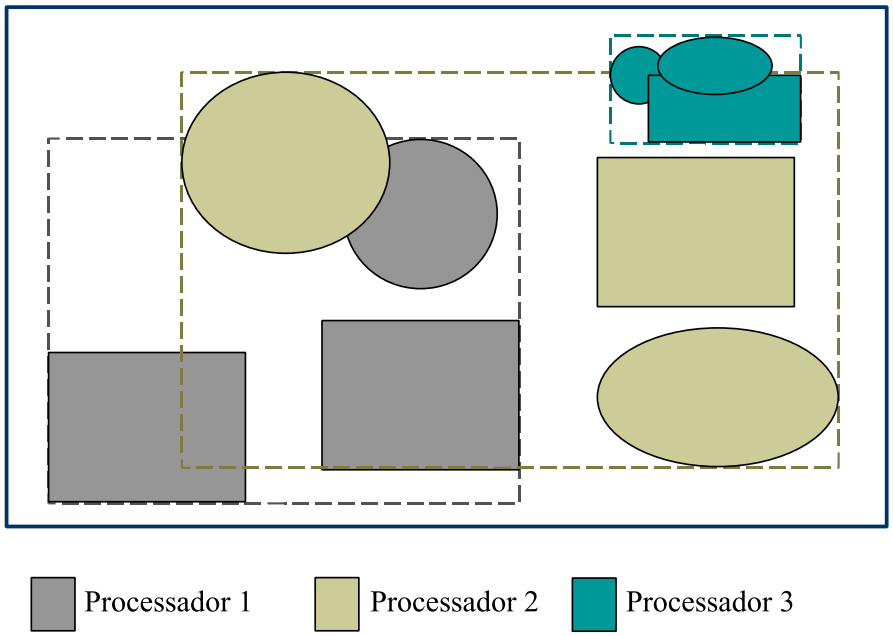

Figura 2.3: Diagrama das arquiteturas com ordenação no fim. Cada processador desenha um pedaço do modelo, havendo necessidade de se enviar informações de visibilidade para o compositor dos resultados parciais.

denação no início e com ordenação no fim, Samanta et al. [15] apresentaram uma arquitetura híbrida, na qual a partição do modelo 3D entre os processadores é feita considerando-se a projeção dos objetos sobre a tela. O uso de uma partição que agrupa objetos próximos no espaço de tela diminui a interseção entre as áreas que devem ser lidas em cada processador, diminuindo portanto a quantidade de informações de visibilidade que devem ser lidas, enviadas e compostas a cada quadro. Isto está ilustrado na Figura 2.4. Esta técnica apresentou bons resultados em termos de escalabilidade, porém o algoritmo apresenta um possível gargalo no estágio da distribuição de trabalho, que é feita inteiramente no cliente e é dependente da complexidade do modelo.

Ma et al. [8] apresentaram uma técnica chamada "Binary-Swap Compositing" com o objetivo de paralelizar o estágio de composição nas arquiteturas com ordenação no fim. Nessa técnica, a composição é realizada em múltiplos passos. Em cada passo, os processadores são agrupados em pares, dividem suas imagens no meio e as trocam, ficando cada processador responsável por uma metade da imagem e trocando a outra metade. Este processo é feito de forma a não deixar nenhum processador ocioso durante a composição. Isto leva a um número de passos de composição proporcional ao logaritmo do número de processadores e à divisão do esforço total de composição. A latência de resposta é aumentada com essa técnica pela

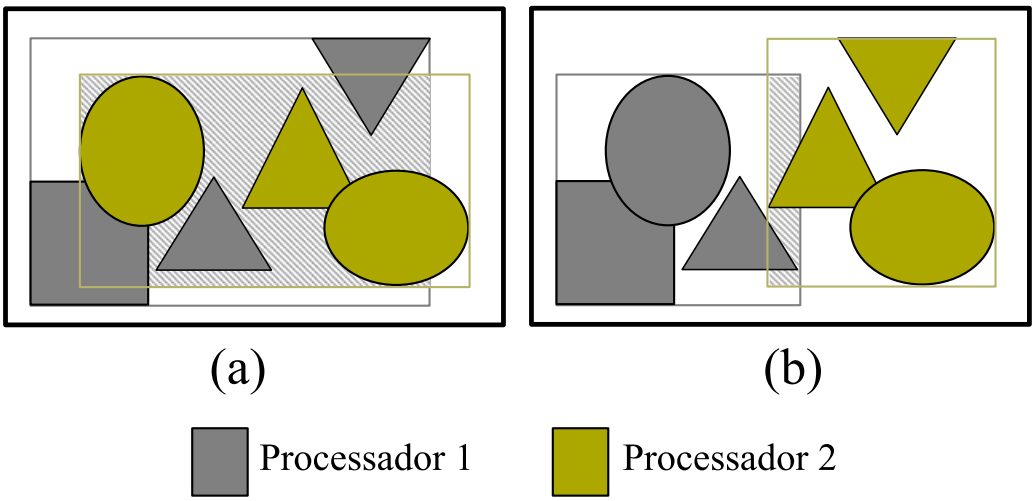

Figura 2.4: A influência da distribuição espacial dos objetos na quantidade de dados a serem transmitidos: (a) agrupamento aleatório; (b) agrupamento espacial.

existência de múltiplos passos de composição. Yang et al. [18] e Takeuchi et al. [26] apresentaram melhorias a esse algoritmo, principalmente visando o balanceamento de carga na composição.

2.2 Modelos de Distribuição de Dados

Com relação à maneira como os dados são distribuídos em um sistema de renderização distribuída, existem dois modelos gerais: Cliente-Servidor e Mestre-Escravo.

No modelo Cliente-Servidor, o usuário interage com uma única instância da aplicação, a qual é executada em um nó cliente. Este cliente é responsável por gerar a geometria a ser renderizada e distribuí-la entre os servidores de renderização. Dois modelos de distribuição de primitivas emergem desta arquitetura: o Modo Imediato e o Modo Retido. Nas arquiteturas em Modo Imediato, a cada quadro todas as primitivas visíveis são enviadas pelo nó cliente aos nós servidores. No Modo Retido, cada servidor de renderização armazena as primitivas que já lhe foram enviadas. Assim, o cliente apenas precisa enviar as mudanças na geometria e tratar da transição de geometria entre os nós. Este modelo utiliza menos largura de banda da rede que o Modo Imediato, porém requer estruturas de dados mais complexas, como grafos de cena.

Já no modelo Mestre-Escravo, uma cópia da aplicação é executada em cada nó do agrupamento de PCs. Estas execuções devem ser sincronizadas

para garantir a consistência entre os nós, o que é feito elegendo-se um nó como "mestre". Este fica responsável por tratar as entradas do usuário e de sistemas externos e propagar mudanças de estado aos nós "escravos". Neste modelo de distribuição, o modelo 3D é comumente replicado nos nós "escravos", evitando a transmissão de primitivas pela rede. A vantagem deste modelo de distribuição é o seu relativo baixo custo de comunicação, porém neste modelo é necessário tratar qualquer evento que afete a execução do programa como entrada (incluindo mudanças no modelo 3D), o que pode aumentar a sua complexidade [25]. Diagramas dos dois modelos encontram-se na Figura 2.5

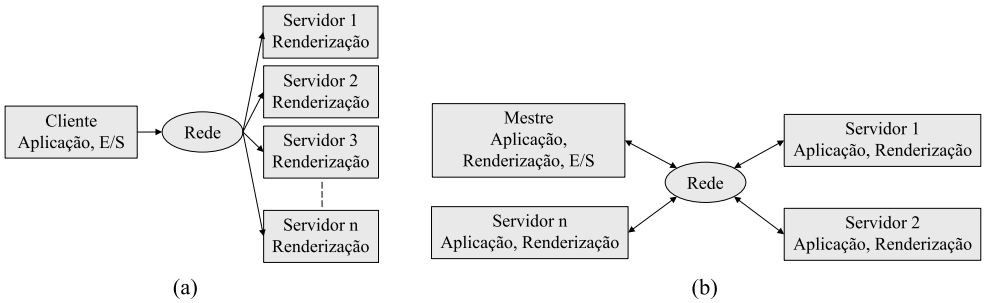

Figura 2.5: Arquiteturas: (a) Cliente-Servidor; (b) Mestre-Escravo.

Humphreys et al. apresentaram Chromium [22], sucessor do WireGL [19], um sistema para a manipulação de *streams* de comandos de APIs gráficas em agrupamentos de PCs. Esse sistema possibilita a construção de processadores de comandos do OpenGL [29] (SPUs). Duas de suas grandes vantagens são o encadeamento de SPUs, o que possibilita a construção de filtros de comandos, e a herança de SPUs, o que traz grande flexibilidade e reuso de código na implementação de arquiteturas para renderização distribuída. Alguns SPUs foram pré-implementados e são distribuídos junto com o pacote do Chromium (que é software livre), tais como SPUs para:

- empacotamento e desempacotamento de comandos gráficos visando o envio e recebimento destes pela rede;
- divisão da geometria em ladrilhos, visando a implementação de sistemas com ordenação no início;
- leitura do *framebuffer*, para sistemas que necessitam disso para compor imagens em outros processos.

Um dos grandes atrativos desse sistema é sua implementação, que é feita como uma biblioteca que substitui a biblioteca do OpenGL, havendo um SPU pré-implementado que é uma ponte para a biblioteca original do

OpenGL. Isso permite que uma aplicação que utiliza OpenGL seja executada sem modificações em um agrupamento de PCs.

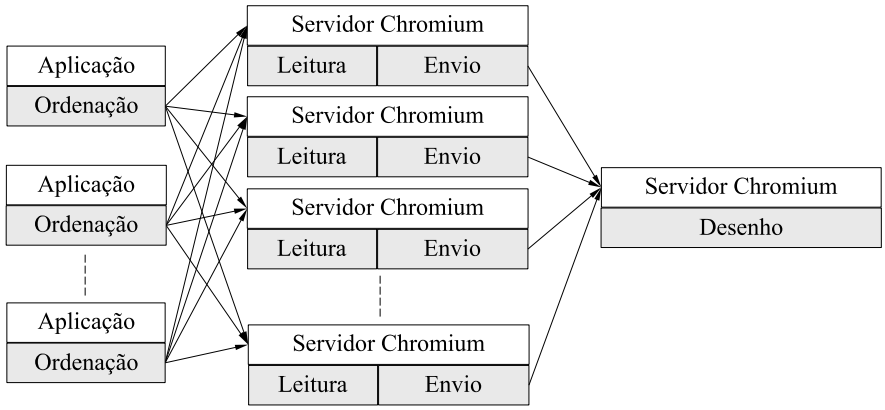

Figura 2.6: Um sistema com ordenação no início implementado utilizando o Chromium. Em cinza, as correntes de SPUs utilizadas.

Van der Schaaf et al. [23] compararam o uso de arquiteturas em Modo Imediato, como o WireGL [19] (antecessor do Chromium), com arquiteturas em Modo Retido para o uso em sistemas com múltiplos projetores. Eles demonstraram que o Modo Imediato expõe o sistema a problemas sérios de escalabilidade, o que eles tentaram resolver utilizando o Modo Retido. Eles experimentaram dois métodos: replicar os dados nos nós e transmitir comandos gráficos pela rede. O método com replicação de dados foi o melhor em termos de desempenho enquanto o método de transmissão de comandos aumentou a transparência da aplicação e facilitou o tratamento de entradas do usuário.

Chen et al. [17] apontaram um dos grandes desafios para a construção de sistemas de renderização distribuída: a construção de algoritmos escaláveis de partição e distribuição de trabalho que utilizem eficientemente a largura de banda de rede disponível. Eles concluíram isto comparando sistemas do tipo Mestre-Escravo com sistemas Cliente-Servidor, além de apresentar outra solução apenas aplicável a sistemas multi-projetor.

Com relação à distribuição do modelo 3D nos nós do agrupamento de PCs, Samanta et al. [20] propuseram uma solução para o problema de replicação armazenando cópias de pedaços do modelo em apenas k dos n nós de renderização, sendo $k < n$. Eles levaram em conta a distribuição espacial dos pedaços do modelo para garantir que nenhum nó do agrupamento fique ocioso. Utilizam o sistema híbrido desenvolvido por Samanta et al. [15] para a renderização, tendo cada nó do agrupamento desenhando a parte do modelo que possui em memória e que lhe é atribuída nesse sistema. Com esse

sistema, eles obtiveram com fatores pequenos de replicação uma eficiência similar à medida com a replicação total do modelo. Corrêa et al. [24] desenvolveram um sistema com ordenação no início capaz de renderizar modelos maiores do que a memória total do agrupamento, utilizando técnicas para gerenciar o que deve ser obtido de armazenamento secundário (disco) para a visualização.

Alguns desses sistemas optaram na sua implementação pelo uso de mais de uma linha de execução (*thread*) com o intuito de paralelizar operações executadas na CPU, GPU e operações de IO (entrada-saída), aumentando assim a taxa de renderização efetiva da aplicação. Todos mencionaram que este aumento na taxa de renderização traz como desvantagem um aumento na latência de resposta ao usuário, a qual deve ser controlada para não prejudicar a interação do usuário com a aplicação. Nguyen et al. [21] apresentaram um sistema para renderização distribuída em agrupamentos de PCs com um algoritmo de partição baseado na decomposição da imagem em camadas. O algoritmo de partição constrói um grafo de oclusão que, uma vez obtido, é particionado em sub-grafos, com base nos tempos de renderização do quadro anterior. Este sistema utiliza duas linhas de execução para paralelizar o envio de camadas da imagem de um quadro e a geração das imagens do quadro seguinte.

2.3 Balanceamento de Carga

Desde os primórdios da renderização paralela e distribuída (mesmo em sistemas que não visavam tempo real), um dos seus maiores desafios tem sido distribuir o esforço de renderização igualmente entre os processadores de renderização. Isso se torna necessário já que a taxa efetiva de renderização da aplicação fica limitada ao tempo de processamento do processador mais sobrecarregado. Isto causa ociosidade nos demais processadores, o que é indesejado, conforme ilustrado na Figura. 2.7. Esta seção visa listar os principais trabalhos anteriores sobre balanceamento de carga.

Mueller [11] apresentou um estudo sobre as arquiteturas com ordenação no início, onde avaliou os algoritmos de balanceamento de carga propostos para esta arquitetura até então [5, 6, 9]. A partir destes algoritmos, ele desenvolveu um algoritmo que primeiramente conta a interseção das primitivas do modelo com cada célula de uma grade que divide a tela. Para evitar os erros decorrentes de se contar múltiplas vezes primitivas grandes, a quantidade acrescentada a cada célula da grade é inversamente propor-

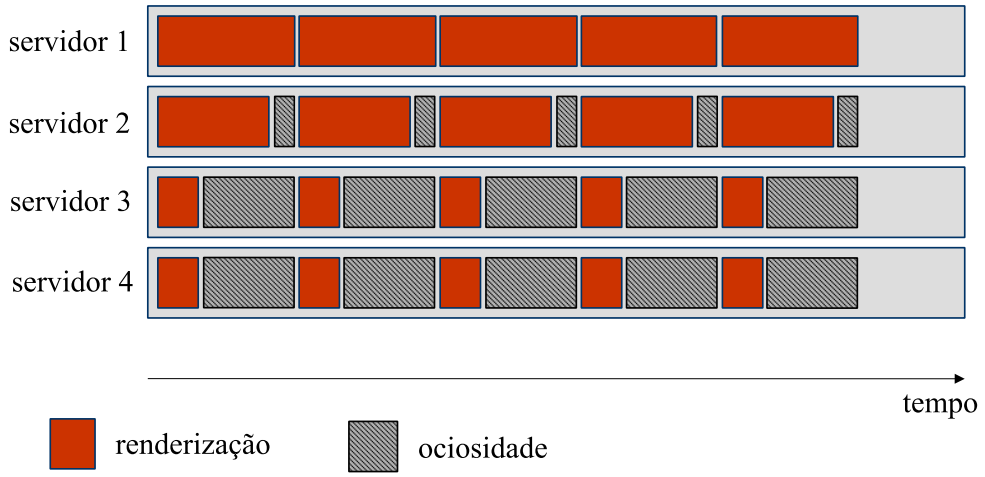

Figura 2.7: Ociosidade nos processadores causada pelo desbalanceamento de carga.

cional ao número de células que a projeção da primitiva intersecta. Uma vez que todas as primitivas foram contadas, as células são somadas em uma estrutura de dados chamada Tabela de Área Somada [3]. Finalmente, a tela é dividida em torno das bordas das células, dividindo-se as áreas de igual carga recursivamente em torno da maior dimensão, até que o número de áreas iguale o número de processadores. A estrutura da Tabela de Área Somada permite que a procura pelo ponto de divisão das áreas seja feita com uma busca binária, o que é bastante eficiente. Este algoritmo foi chamado MAHD (Mesh-based Adaptive Hierarchical Decomposition).

Neste trabalho, Mueller mediu o desbalanceamento de carga como:

$$imbal = \frac{dt_{max}}{dt_{avg}}$$

onde dt_{max} é o esforço do processador mais lento e dt_{avg} é o esforço médio entre os processadores. Ele concluiu que um balanceamento de carga é aceitável se essa razão for menor que 1.5.

Samanta et al. [14] apresentaram algoritmos de balanceamento de carga para sistemas multi-projetor. Dentre eles, o de melhor desempenho é o KD-SPLIT. Nesse algoritmo, a tela é dividida em P ladrilhos, um para cada processador. A divisão começa com a tela inteira e é feita recursivamente com $P - 1$ linhas, formando uma KD-Tree com P regiões. No início de cada quadro, o cliente calcula a projeção 2D de cada nó 3D do grafo de cena na tela. Posteriormente, ele utiliza um algoritmo de varredura para dividir a carga igualmente utilizando uma linha. A linha começa à esquerda e vai se movendo para a direita até que a carga seja dividida ao meio.

Este procedimento é repetido recursivamente $P - 1$ vezes para construir P ladrilhos, alternando entre linhas verticais e horizontais. Esse algoritmo obteve bons resultados em termos de balanceamento de carga, porém o tempo gasto no cliente é grande devido à necessidade de se utilizar todo o modelo visível para balancear a carga.

Samanta et al. [15] posteriormente apresentaram um sistema híbrido, que particiona o modelo 3D visível entre ladrilhos 2D na tentativa de minimizar a área de interseção entre os pedaços do modelo na tela, reduzindo a necessidade de se transferir informações de visibilidade (Z-Buffer) pela rede. O algoritmo é parecido com o mencionado acima, porém são utilizadas duas linhas, uma que vai da esquerda para a direita e outra que vai da direita para a esquerda. A cada passo, uma das linhas anda na sua direção e um objeto 3D é adicionado à lista de objetos da linha. Este procedimento é feito até que as duas linhas se cruzem. Nesse momento, a partição está completa e a área de interseção é a área formada entre as duas linhas após o seu cruzamento. Este procedimento é repetido também $P - 1$ vezes para se construir P ladrilhos, alternando entre linhas verticais e horizontais. Novamente, o algoritmo alcançou bons resultados, conseguindo balancear a carga e diminuir a necessidade de transferência de informações de visibilidade, porém o tempo gasto no cliente com o algoritmo de partição é grande, sendo algumas vezes o gargalo da aplicação nos resultados apresentados.