UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Smart Visible Sets para Ambientes de Rede

por

FÁBIO OSÓRIO MOREIRA

Dissertação submetida à avaliação, como
requisito parcial para a obtenção do grau de
Mestre em Ciência da Computação

Prof[a]. Dra. Carla Maria Dal Sasso Freitas
Orientadora

Prof. Dr. João L. D. Comba
Co-orientador

Porto Alegre, março de 2003.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Moreira, Fábio Osório

Smart Visible Sets para Ambientes de Rede / por Fábio Osório Moreira. – Porto Alegre: PPGC da UFRGS, 2003.

86 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul, Programa de Pós-Graduação em Computação, Porto Alegre, BR – RS, 2003. Orientadora: Freitas, Carla M. D. S.; Co-orientador: Comba, João L. D.

1. SVS 2. PVS 3. Algoritmo de visibilidade I. Freitas, Carla M. D. S. II. Comba, João L. D. III. Título.

"O maior estímulo para ter disciplina é o desejo de atingir um objetivo"

Içami Tiba

Agradecimentos

O mestrado representou um período em que muitas coisas aconteceram em minha vida, durante o qual muitas pessoas participaram e ajudaram.

Agradeço aos meus pais Hilton e Maria Luiza e meu irmão Guilherme pelo amor, carinho, apoio e incentivo que recebi em todos os momentos de minha vida.

Agradeço a minha avó Luiza pela companhia constante.

Agradeço a minha orientadora e co-orientador pelo incentivo, apoio técnico e ensinamentos transmitidos durante o desenvolvimento dos trabalhos.

Enfim gostaria de agradecer a todos os colegas de mestrado, aos funcionários e professores do Instituto de Informática por toda a ajuda e lições que me deixaram.

Sumário

Lista de Abreviaturas

2D	Bidimensional
3D	Tridimensional
BSP-Tree	Binary Space Partitioning Tree
DDO	Obstáculos direcionais discretizados
HOM	Hierarchical Occlusion Map
HSR	Hidden Surface Removal
HZB	Hierarchical Z-Buffer
LOD	Level of Detail
Open GL	Open Graphics Library
PLP	Prioritized-Layered Projection
PVS	Potentially Visible Sets
SVBSP	Shadow Volume BSP-Tree
SVS	Smart Visible Sets

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Resumo

A visualização em tempo real de cenas complexas através de ambientes de rede é um dos desafios na computação gráfica. O uso da visibilidade pré-computada associada a regiões do espaço, tal como a abordagem dos *Potentially Visible Sets* (PVS), pode reduzir a quantidade de dados enviados através da rede. Entretanto, o PVS para algumas regiões pode ainda ser bastante complexo, e portanto uma estratégia diferente para diminuir a quantidade de informações é necessária. Neste trabalho é introduzido o conceito de *Smart Visible Set* (SVS), que corresponde a uma partição das informações contidas no PVS segundo o ângulo de visão do observador e as distâncias entre as regiões. Dessa forma, o conceito de "visível" ou de "não-visível" encontrado nos PVS é estendido. A informação referente ao conjunto "visível" é ampliada para "dentro do campo de visão" ou "fora do campo de visão" e "longe" ou "perto".

Desta forma a informação referente ao conjunto "visível" é subdividida, permitindo um maior controle sobre cortes ou ajustes nos dados que devem ser feitos para adequar a quantidade de dados a ser transmitida aos limites impostos pela rede. O armazenamento dos SVS como matrizes de bits permite ainda uma interação entre diferentes SVS. Outros SVS podem ser adicionados ou subtraídos entre si com um custo computacional muito pequeno permitindo uma rápida alteração no resultado final. Transmitir apenas a informação dentro de campo de visão do usuário ou não transmitir a informação muito distante são exemplos dos tipos de ajustes que podem ser realizados para se diminuir a quantidade de informações enviadas.

Como o cálculo do SVS depende da existência de informação de visibilidade entre regiões foi implementado o algoritmo conhecido como "Dual Ray Space", que por sua vez depende do particionamento da cena em regiões. Para o particionamento da cena em uma *BSP-Tree*, foi modificada a aplicação QBSP3. Depois de calculada, a visibilidade é particionada em diferentes conjuntos através da aplicação SVS. Finalmente, diferentes tipos de SVS puderam ser testados em uma aplicação de navegação por um cenário 3D chamada BSPViewer. Essa aplicação também permite comparações entre diferentes tipos de SVS e PVS.

Os resultados obtidos apontam o SVS como uma forma de redução da quantidade de polígonos que devem ser renderizados em uma cena, diminuindo a quantidade de informação que deve ser enviada aos usuários. O SVS particionado pela distância entre as regiões permite um corte rápido na informação muito distante do usuário. Outra vantagem do uso dos SVS é que pode ser realizado um ordenamento das informações segundo sua importância para o usuário, desde que uma métrica de importância visual tenha sido definida previamente.

Palavras-Chave: SVS, PVS, algoritmo de visibilidade

TITLE: “SMART VISIBLE SETS FOR NETWORK ENVIRONMENTS”.

Abstract

Real-time visualization of complex virtual environments across a network is a challenging problem in Computer Graphics. The use of pre-computed visibility associated to regions in space, such as in the *Potentially Visible Sets* (PVS) approach, may reduce the amount of data sent across the network. However, a PVS for a region may still be complex, and a different strategy for reducing the volume of network data will be necessary. We introduce the concept of Smart Visible Set (SVS), which corresponds to a partition of PVS information into subsets according to the user’s field of view and the distance between the regions. The “visible” or “non-visible” concept found in the PVS is extended. The information in the visible set is partitioned into “inside the field-of-view” or “outside the field-of-view” and “near” or “far”.

This partitioning allows a better control over culling and adjustments on the data that must be made to conform to the amount of information that needs to be sent to a user due to the network limitations. The storage of the SVS as bit arrays allows an easy combination among them. SVSs can be added or subtracted with small computational cost allowing fast changes of the final result. Examples of adjustments that can be made to reduce the amount of broadcast data include sending only information inside the user’s field of view or culling the information that lies too far away from the user.

Since SVS computation depends on the existence of visibility information among regions, the “Dual Ray Space” algorithm was implemented. This algorithm requires the scene to be partitioned into regions. This partitioning using a *BSP-Tree* required changes in the QBSP3 algorithm. After the visibility is computed, the visible set is partitioned into different sets by the SVS application. Different SVS were tested in a walkthrough application named BSPViewer. This application also allows the comparison among different SVS and PVS.

The results show that the SVS can be used to reduce the number of rendered polygons, decreasing the amount of data that needs to be sent to the users. The SVS partitioned by distance between regions allows fast culling of the information located too far from the user. Another advantage of the SVS is that information can be ordered according to its importance to the user, as long as a visual importance metrics had been previously defined.

Keywords: SVS, PVS, visibility culling

1 Introdução

1.1 Motivação

Jogos de computadores são uma das principais forças motrizes do desenvolvimento de hardware e software gráficos nos dias de hoje. Os recentes avanços no hardware gráfico permitem a criação de cenários elaborados contendo efeitos complexos de texturas e iluminação.

Um dos aspectos mais promissores nessa área são os jogos *online* nas quais milhares de pessoas interagem umas com as outras em um ambiente virtual através de uma rede de comunicação. Nos jogos *online*, há necessidade de enviar informações como as posições dos jogadores e suas ações a todos os usuários. Como o número de usuários nesse tipo de aplicação costuma ser muito grande é extremamente importante reduzir substancialmente a quantidade de informações enviada. Dezenas de jogos *online* foram desenvolvidos nos últimos anos, dentre os quais podem ser citados Ultima Online[1], Everquest[2] e Ashron's Call[3] (Figura 1.1).

FIGURA 1.1– Exemplos de jogos *online* (http://everquest.station.sony.com/screenshot.jsp)

Outro tipo de aplicação que incorpora os novos desenvolvimentos de hardware e software gráficos é o passeio virtual [WON 99] [KOL 2000] [KOL 2001] [WON 2001], onde clientes navegam por um ambiente virtual. Nesses ambientes o cenário

[1] http://www.everquest.com
[2] http://www.ultimaonline.com
[3] http://www.asheronscall.com

normalmente é bastante complexo, em muitos casos impossibilitando que o clientes possuam cópias locais da cena em suas máquinas. Isso obriga que a estrutura do cenário seja enviada aos clientes através da rede.

Em ambos os casos a estrutura da rede pode ficar sobrecarregada, fazendo com que os limites desta conexão sejam atingidos e gerando atraso no envio de informações (problema de latência [WOL 2002]). Esse atraso causa perda no realismo da simulação e problemas na conexão cliente-servidor (como por exemplo a queda da conexão).

Estratégias comuns para se diminuir a quantidade de dados enviadas aos clientes são:

- Algoritmos de visibilidade [WON 99] [KOL 2000] [KOL 2001] [WON 2001]: calculam um conjunto de objetos potencialmente visíveis para um ponto de vista ou uma região, enviando ao usuário apenas informações relevantes ao seu campo visual.
- Ordenação da informação quanto a sua importância visual [LU 2002]: permite que a informação mais relevante ao campo visual do usuário seja enviada antes das demais, possibilitando a eliminação das supérfluas.
- Estratégias de nível de detalhe (LOD) [LUE 2002]: permitem que objetos menos detalhados sejam utilizados de acordo com sua importância visual para o usuário, permitindo uma diminuição no número de vértices da cena.
- *Rendering* baseado em imagens [MAC 95], [DEC 99]: possibilita a diminuição do número de polígonos na cena através da substituição de parte da geometria por imagens 2D.

Integrar algumas dessas técnicas para a solução do problema de latência (notadamente algoritmos de visibilidade e ordenação de informação) é o objetivo deste trabalho.

O pré-cálculo da visibilidade e o seu armazenamento em *Potentially Visible Sets* (PVS) [TEL 91] têm sido utilizados em diversas aplicações como os jogos Doom[4], Quake[5] e Half-Life[6]. Um PVS é nada mais do que uma lista de visibilidade. Ele representa uma lista de objetos que indicam o que pode ser visto a partir de um ponto ou região do ambiente. Caso se utilize um PVS baseado em regiões, deve-se utilizar uma estrutura de dados espacial que particione a cena, como por exemplo *BSP-Trees* [FUC 80] ou *Octrees* [SUT 99].

Uma estratégia utilizada em passeios virtuais onde o cenário é transmitido aos clientes é enviar inicialmente o cenário do PVS da região onde o usuário se encontra e, enquanto ele percorre essa região, transmitir os cenários dos PVS das regiões adjacentes. Essa estratégia mantém a rede em uso constante e permite que a coerência espacial seja explorada. Se um único PVS ainda contiver mais informações do que a rede consegue transmitir podem ser feitos novos ordenamentos ou simplificação do nível geométrico de detalhe baseado numa métrica de importância.

[4] http://www.idsoftware.com/games/doom
[5] http://www.idsoftware.com/games/quake
[6] http://www.half-life.com

1.2 Objetivos

Neste trabalho são propostos os *Smart Visible Sets* (SVS) como forma de permitir que a visibilidade pré-computada seja adaptada às necessidades dos parâmetros de visualização do cliente. Normalmente, um PVS não está armazenado de uma maneira que possa ser eficientemente adaptada. No SVS, objetos são inicialmente agrupados segundo a posição em que se encontram em relação ao hemisfério de direções de visualização. Essa quebra nos ângulos de visualização permite que objetos localizados dentro do campo de visão do cliente sejam enviados antes dos demais. A quebra do hemisfério de visualização é bastante flexível, podendo ser realizada em quantas regiões quanto necessário e de maneira dependente da região (diferentes regiões quebram o hemisfério de visualização de modo diferente).

Adicionalmente, a visibilidade pode ser dividida em grupos segundo a distância da região de origem. Esse pré-cálculo da distância permite que estratégias de nível de detalhe ou de substituição de geometria por imagens sejam utilizadas rapidamente. Diferentes tipos de SVS podem ainda ser combinados rapidamente entre si gerando conjuntos mais complexos e possibilitando que as características de visibilidade do cliente sejam alteradas durante o tempo de execução da aplicação.

Assim, o SVS representa uma estrutura de dados indexada da informação de visibilidade, classificada em subgrupos divididos por ângulos e distâncias. Com essa representação, pode ser utilizado um mecanismo que se baseie numa métrica de importância pré-definida (talvez definida pelo usuário) para enumerar a informação a ser enviada ao cliente.

1.3 Organização do Texto

No próximo capítulo será realizada uma revisão das técnicas de cálculo de visibilidade dando-se atenção especial aos algoritmos que trabalham com conjuntos potencialmente visíveis (PVS). Será realizada uma análise mais detalhada dos trabalhos recentes nessa área.

O Capítulo 3 apresenta e discute a proposta dos SVS como um refinamento dos PVS. Os SVS baseados em distâncias e ângulos são explicados e os algoritmos envolvidos nos seus cálculos detalhados. Finalmente, mostra-se como o SVS pode ser utilizado para realizar o ordenamento de informações.

No Capítulo 4 o ambiente desenvolvido para testes e validação dos SVS é descrito. Esse ambiente inclui um gerador de cidades, um gerador de *BSP-Trees*, um módulo que realiza o cálculo dos PVS, um módulo que realiza o cálculo dos SVS e, finalmente, um ambiente de visualização destas estruturas. Cada um destes módulos é explicado detalhadamente.

Os resultados obtidos em termos de eficiência com diferentes tipos de SVS são analisados no Capítulo 5. São também comparados com os resultados do PVS clássico. Para estas comparações é utilizado um cenário de 75 prédios. O Capítulo 5 encerra com comentários sobre as limitações e problemas encontrados durante o trabalho.

Finalmente, no Capítulo 6 é realizada uma avaliação dos resultados obtidos e são sugeridos trabalhos futuros a serem realizados com o objetivo de melhorar a eficiência dos SVS.

2 Técnicas de Visibilidade Pré-Computada

2.1 Introdução

Algoritmos de determinação de visibilidade representam uma área de grande importância na computação gráfica [APP 68] [SUT 74]. Inicialmente, algoritmos de remoção de faces ocultas (HSR) foram desenvolvidos para resolver o problema da determinação das superfícies visíveis na cena. Atualmente, o *Z-Buffer* [CAT 74] é a técnica padrão de remoção de superfícies visíveis, disponível em todo hardware gráfico. O foco dos estudos nessa área, atualmente, está nos algoritmos de *occlusion culling* que rapidamente rejeitam as partes da cena que não contribuem para a geração da imagem final.

O *pipeline* gráfico tradicional contém duas técnicas de *occlusion culling* bastante simples: *view-frustum culling* e *back-face culling* [FOL 90]. Algoritmos de b*ack-face culling* evitam a "renderização" da geometria voltada para o lado contrário do observador, enquanto que algoritmos de *view-frustum culling* evitam a "renderização" da geometria que se encontra fora do campo de visão do usuário. Essas técnicas são classificadas como locais, pois podem ser aplicadas a quaisquer polígonos, independente da posição dos demais polígonos da cena.

Occlusion culling é uma técnica que busca descartar do processo de "renderização" as primitivas geométricas que estejam ocultas por componentes da cena. Como essa técnica depende da posição dos demais polígonos ela é classificada como global e é muito mais complexa do que os métodos locais. A Figura 2.1 mostra os três tipos de corte existentes.

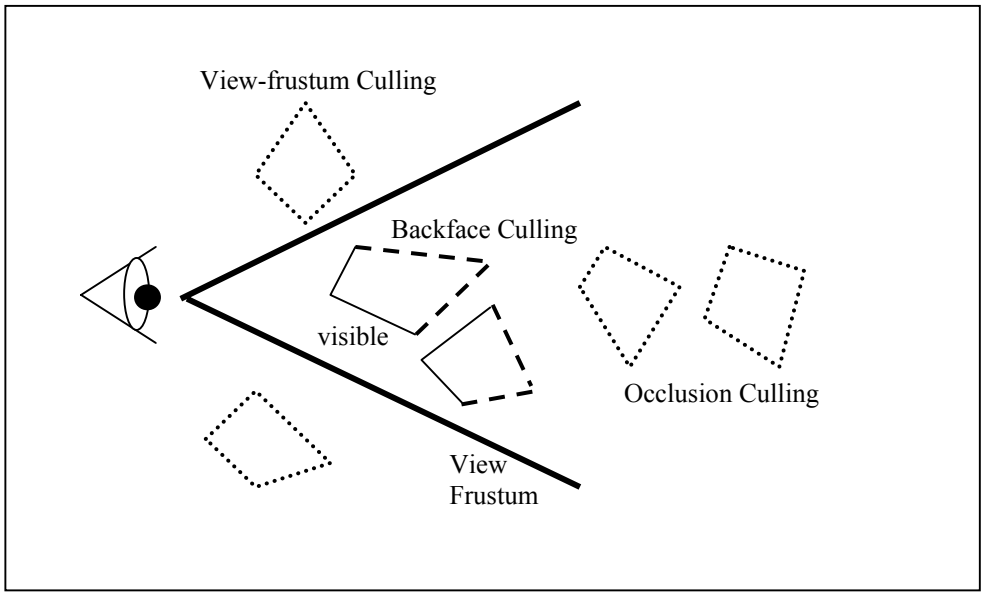


FIGURA 2.1 – Três tipos de *occlusion culling*: *view-frustum culling*, *back-face culling* e *occlusion culling* [COH 2003] p.2

Um conceito importante é visibilidade conservativa [AIR 91] [TEL 91]. O conjunto da visibilidade conservativa é aquele que inclui pelo menos todos os objetos visíveis além de um certo número (preferencialmente pequeno) de objetos não-visíveis.

Nele pode-se classificar um objeto oculto como visível, mas jamais um objeto visível como oculto. Uma estimativa conservativa do conjunto visível pode definir um conjunto potencialmente visível (PVS) [BIT 98] [COO 97] [HUD 97]. O PVS pode ser definido com relação a um único ponto de vista ou a uma região do espaço.

Ao contrário dos algoritmos de HSR que investem em cálculos para identificar as partes visíveis dos polígonos, as técnicas de *occlusion culling* tentam simplesmente identificar quais polígonos são visíveis. Seu objetivo é diminuir o custo de "renderização" de uma cena ao grau de complexidade da sua porção visível, independentemente do tamanho total da cena [FEI 76] [GRE 2001].

2.2 Técnicas de *Occlusion Culling*

Técnicas de *occlusion culling* consideram vários aspectos [COH 2003]:

- **Região de interesse:** define a geometria da região de interesse para cálculos de visibilidade. Existem técnicas baseadas em pontos e em regiões. Técnicas baseadas em pontos são mais simples de serem implementadas, contudo seu desempenho normalmente não possibilita sua utilização em aplicações que requerem navegação por um cenário em tempo real. Já nas técnicas baseadas em regiões o custo do cálculo de visibilidade é amortizado ao longo do tempo em que o usuário se encontra dentro daquela região. Além disso, é possível prever quais serão as próximas regiões a serem visitadas e realizar um pré-cálculo da visibilidade. Essa característica é de grande valia para aplicações de rede ou para antecipar o carregamento de dados pré-armazenados no disco para a memória. Contudo os algoritmos envolvidos são geralmente mais complexos.
- **Precisão de imagem ou precisão de objeto** [SUT 74]: este critério é particularmente relevante a métodos de visibilidade a partir de ponto. Métodos com precisão de objeto usam as coordenadas dos próprios objetos nos seus cálculos de visibilidade. Métodos de precisão de imagem operam com representações discretizadas dos objetos quando divididos em partes durante o processo de rasterização.
- **Células e portais**: este critério é particularmente relevante a métodos de visibilidade a partir de região. Alguns algoritmos exploram cenas que são organizadas em células (ou salas), ligadas por portais (portas ou janelas). O método de visibilidade leva em conta que outras células serão visíveis apenas através dos portais.
- **Conservativo ou Aproximativo**: técnicas conservativas superestimam o conjunto visível. Nas técnicas aproximativas, não há garantia de que todos os polígonos visíveis sejam identificados. Uma vez que, em sua maioria esses algoritmos são conservativos, é interessante se estudar o grau de superdimensionamento do conjunto visível.
- **Parte dos obstáculos ou todos obstáculos**: alguns métodos consideram a oclusão gerada por todos obstáculos da cena enquanto outros requerem a seleção de apenas parte deste conjunto (normalmente formada por obstáculos grandes).
- **Obstáculos convexos ou gerais**: Alguns métodos requerem que os obstáculos sejam convexos.

- **Obstáculos simples ou agregados**: dadas três primitivas A, B e C, pode acontecer de nem A ou B ocultarem C, mas juntos sim. Alguns métodos realizam a chamada fusão de obstáculos enquanto que outros consideram apenas obstáculos simples.
- **2D ou 3D**: alguns métodos são restritos a ambientes 2D ou 2.5D, enquanto que outros são capazes de tratar cenas 3D.
- ***Hardware* gráfico especial**: diversas técnicas podem ser otimizadas pelo uso de hardware gráfico, tanto na etapa de pré-processamento como na etapa de "renderização".
- **Pré-processamento de visibilidade**: a maioria dos métodos que calculam visibilidade para regiões pré-computam e armazenam a informação de visibilidade. Algumas técnicas baseadas em pontos também necessitam de alguns dados pré-computados.
- **Cenas dinâmicas**: alguns algoritmos são capazes de lidar com cenas dinâmicas. Uma das maiores dificuldades é lidar com a mudança na hierarquia dos objetos que a maioria dos algoritmos de visibilidade usam [SUD 99]. Se o algoritmo requer pré-processamento, os dados necessitam ser recalculados. Como a maioria dos métodos baseados em regiões geralmente pré-calculam o PVS, é muito difícil para eles o tratamento de cenas dinâmicas.

A discussão será centrada em quatro técnicas de visibilidade: métodos baseados em regiões que exploram estruturas de células e portais, métodos baseados em pontos que trabalham com precisão de objeto, métodos baseados em pontos que trabalham com precisão de imagem e métodos baseados em regiões que trabalham com cenas arbitrárias.

2.2.1 Métodos Baseados em Regiões que Exploram Estruturas de Células e Portais

Os trabalhos de Airey et al. [AIR 90] [AIR 91] e Teller e Séquin [TEL 91] apresentam técnicas avançadas de *occlusion-culling* e desenvolveram muitas das bases dos trabalhos recentes. Alguns conceitos chave introduzidos foram: a noção de "conjunto potencialmente visível" para uma região do espaço, "visibilidade conservativa" e "ambientes com muita oclusão".

Esses trabalhos se baseiam nas características arquitetônicas das cenas, aproveitando-se, em especial, do fato delas serem naturalmente divididas em células e a visibilidade ocorrer através de aberturas (portais). O conjunto potencialmente visível é computado para cada célula e usado durante a etapa de navegação em tempo real. Mais recentemente, Jimenez et al. [JIM 2000] propôs um método mais rápido do que o de [TEL 92] para computar a visibilidade conservativa através de portais utilizando-se de uma adaptação da estrutura conhecida como *Visibility Skeleton* [DUR 97].

2.2.2 Métodos Baseados no Espaço do Objeto

Luebke e Georges [LUE 95] propuseram uma estratégia com base em pontos e portais, inspirada numa idéia anterior de Jones [JON 71] e em métodos baseados em regiões discutidos acima [AIR 91] [TEL 92]. Em vez de pré-computar a geometria

potencialmente visível para cada célula, o algoritmo percorre primeiro as células mais próximas seguindo em direção às mais afastadas durante a etapa de navegação usando projeções dos portais para realizar *occlusion-culling* conservativa.

O trabalho de Coorg e Teller [COO 96] [COO 97] calcula a oclusão causada por um subconjunto de grandes obstáculos convexos. A fusão de obstáculos não é tratada, mas eles se aproveitam da coerência temporal que permite a verificação da visibilidade dos obstáculos somente quando ocorre alguma alteração.

Uma maneira de olhar para as relações de oclusão é usar o fato que o observador não pode ver o objeto na área de sombra gerada por obstáculos. Hudson et al. [HUD 97] propôs uma abordagem baseada na escolha dinâmica de um conjunto de obstáculos e no cálculo de suas sombras, que é utilizado para eliminação das *bounding-boxes* de uma hierarquia de objetos.

O método descrito por Hudson et al. [HUD 97] pode ser otimizado pelo uso de *BSP-Trees*. Bittner et al. [BIT 98] combinam a sombra dos obstáculos em uma árvore de oclusão (*occlusion tree*). Isto é feito de modo semelhante ao *Shadow Volume BSP-Tree* (SVBSP) de Chin e Feiner [CHI 89]. Esse método é conservativo; um método exato foi proposto muito antes por Naylor [NAY 92], o qual envolvia a fusão da árvore de oclusão com a *BSP-Tree* que representa a geometria da cena.

2.2.3 Métodos Baseados no Espaço da Imagem

Os algoritmos dessa classe realizam a eliminação de objetos usando a representação discretizada da imagem. Durante a etapa de "renderização" da cena, uma imagem é preenchida com os objetos mais próximos. Se outros objetos forem mapeados para a mesma região da imagem, eles poderão ser descartados pois estarão ocultos.

Uma das formas mais simples de algoritmos de síntese de imagens é conhecida como *ray casting* onde a imagem é gerada através da determinação do objeto da cena visível para cada *pixel*. Ao emanar um raio que parte do observador passando pelo *pixel* em direção à cena, o primeiro objeto em que colidir irá determinar a superfície mais próxima; ao parar no primeiro obstáculo evita-se que uma parte oculta da cena seja processada.Uma das desvantagens de usar *ray casting* é sua alta complexidade. Contudo, métodos de aceleração são usados e a "renderização" pode ser bastante rápida [BAL 99a] [BAL 99b] [COH 96] [COH 95] [PAR 99].

O *Z-Buffer* Hierárquico (HZB) [GRE 93] [GRE 94] é uma extensão do popular método de HSR, o *Z-Buffer*. Ele utiliza duas hierarquias: uma *octree* em precisão de objeto e uma pirâmide Z em precisão da imagem. A pirâmide Z é um *buffer* em camadas com diferentes resoluções em cada nível. No seu nível mais detalhado, ele apresenta exatamente o conteúdo do *Z-Buffer*. Cada nível de menor detalhe é criado dividindo-se ao meio a resolução em cada dimensão. Em cada elemento desse nível será mapeado o elemento da janela 2 x 2 do nível abaixo que tiver o valor Z mais distante. Isto é realizado em direção ao topo, onde existe apenas um valor correspondendo ao valor Z mais distante do *buffer*. Durante a conversão das primitivas, se o conteúdo do *Z-Buffer* é alterado, então os novos valores Z são propagados através da pirâmide até os níveis menos precisos. Em [GRE 93] a cena é arranjada em uma *octree* que é percorrida de cima para baixo e de frente para trás e cada nodo é testado para oclusão . Se um nodo for oculto, então ele é descartado; caso contrário, seus filhos são recursivamente testados. Qualquer primitiva associada a uma folha não oculta é "renderizada" e a

pirâmide Z, atualizada. Para determinar se um nodo é visível, cada uma de suas faces é testada hierarquicamente contra a pirâmide Z. Iniciando-se com o nível menos detalhado, o valor Z mais próximo da face é comparado contra o valor na pirâmide Z. Se a face estiver mais distante, então é considerada oculta; caso contrário os outros níveis da pirâmide Z são recursivamente percorridos até que a visibilidade possa ser determinada.

O método de mapeamento de oclusão hierárquica [ZHA 97] é similar em seus princípios ao HZB. A oclusão é organizada hierarquicamente em uma estrutura chamada mapa de oclusão hierárquica (HOM) e a hierarquia dos *bounding volumes* da cena é testada contra ela. Entretanto, diferentemente do HZB, o HOM armazena apenas a informação de opacidade enquanto as distâncias dos obstáculos são armazenadas separadamente. O algoritmo, portanto, precisa testar independentemente objetos para sobreposição com as regiões ocultas do HOM e para profundidade. Durante a etapa de pré-processamento, é montado um banco de dados dos obstáculos potenciais. Em tempo de execução, o algoritmo realiza dois passos: construção do HOM e *occlusion culling* da geometria de cena usando o HOM.

A abordagem dos obstáculos direcionais discretizados (DDOs) é semelhante aos métodos HZB e HOM ao usar hierarquias para os objetos e para as imagens. Bernardini et al. [BER 2000] introduz um método para gerar obstáculos eficientes para um ponto de vista. Esses obstáculos são, então, recursivamente usados para eliminar nodos da *octree* durante a "renderização".

Bartz et al. [STA 99] [BAR 98] descrevem outro método onde a representação hierárquica da cena é testada contra a parte oculta da imagem (similar ao HZM e HOM). Contudo, ao contrário desses métodos, não existe representação hierárquica da oclusão. Ao invés disso, é utilizado o *stencil buffer* para se fazer os testes. Um *buffer* separado (*buffer* de oclusão virtual) é associado ao *frame-buffer* para se detectar possíveis contribuições de cada objeto ao *frame-buffer* (isso é implementado através do *stencil buffer*). As *bounding boxes* dos objetos da cena são hierarquicamente "renderizadas". Enquanto são "rasterizadas", os *pixels* correspondentes são enviados ao *buffer* de oclusão virtual se o teste do *Z-Buffer* ocorrer com sucesso. O *frame-buffer* e o *Z-Buffer* permanecem inalterados durante esse processo. O *buffer* de oclusão virtual é, então, lido e qualquer *bounding box* que estiver presente nesse *buffer* é considerada (ao menos parcialmente) visível e suas primitivas podem então ser "renderizadas".

Projetistas de hardware gráfico começaram a adotar funções do *occlusion culling* em seus projetos. Elas são baseadas num laço de feedback para o *hardware* que permite verificar se o *Z-Buffer* foi alterado durante a "renderização" de alguma primitiva. Através do uso dessa função pode-se evitar o processamento de um objeto muito complexo verificando-se sua visibilidade potencial, usando por exemplo um invólucro simples (como uma *bounding box*) e "renderizando" o objeto complexo apenas se seu invólucro também for visível. Consultas ao hardware gráfico são lentas e podem piorar a performance da aplicação caso não sejam feitas com cuidado. Pesquisadores [BAR 2001] [KLO 2001] têm estudado maneiras de diminuir o número de consultas.

Uma abordagem para determinar de modo aproximado a visibilidade é baseada no uso de representação volumétrica. Em vez de realizar cálculos de visibilidade geométrica, pode-se calcular um volume que possui propriedades intrínsecas relacionadas à "densidade" da geometria no ambiente e aproximar a visibilidade entre regiões pelo cálculo do volume de opacidade entre regiões. Essa abordagem foi inicialmente proposta por Sillion [SIL 95a] com o objetivo de aumentar a velocidade

dos cálculos de visibilidade em um sistema de radiosidade e estendido [SIL 95b] em um *framework* de multi-resolução. A visiblidade geométrica foi desenvolvida independentemente por Klosowski e Silva [KLO 2000] [KLO 99] no seu sistema *Prioritized-Layered Projection* (PLP), onde é usada para fazer uma estimativa grosseira da ordem da projeção da geometria.

Muitas cenas 3D têm, de fato, apenas duas dimensões e meia como por exemplo as cenas do tipo *terrain* ou *height field*, ou seja, aquelas que representam funções do tipo z = f(x,y). Wonka e Schmalstieg [WON 99] exploram essa característica para calcular oclusão com respeito a um ponto usando um *Z-Buffer* com uma visão superior e paralela à cena.

2.2.4 Métodos de Regiões com Cenas Arbitrárias

Schaufler et al. [SCH 2000] introduziram uma técnica conservativa para o cálculo da visibilidade da célula de visualização. O método usa uma representação discretizada do espaço e do interior opaco dos objetos como obstáculos. As regiões do espaço ocultas por obstáculos são detectadas de forma eficiente, usando a propriedade que obstáculos podem ser estendidos em espaços vazios, desde que esse espaço também esteja oculto da célula de visualização. Isso é uma forma eficiente de computar a oclusão de um conjunto de obstáculos.

Durand et al. [DUR 2000] apresenta uma extensão dos métodos baseados em pontos com precisão de imagem como o HOM [ZHA 97] ou HZB [GRE 93] utilizando visibilidade volumétrica das células de visualização. Obstáculos e objetos são projetados em um plano e um objeto é declarado oculto se sua projeção estiver completamente coberta pela projeção cumulativa dos obstáculos. A projeção é mais complexa em casos de visibilidade volumétrica. Para assegurar a conservatividade, a projeção estendida de um obstáculo subestima sua projeção a partir de qualquer ponto da célula de visualização, enquanto que a projeção estendida de um objeto é superestimada.

Koltun et al. [KOL 2000] introduziram a noção de obstáculos virtuais e propuseram uma implementação 2.5D. Dada uma cena e uma célula de visualização, um obstáculo virtual é um objeto convexo, que está oculto por inteiro a partir de qualquer ponto dentro da célula de visão e que serve como um obstáculo efetivo. Obstáculos virtuais representam a oclusão agregada de forma compacta. O trabalho demonstra que um número pequeno de obstáculos virtuais é suficiente para computar o PVS de forma eficiente durante a navegação.

Wonka et al. [WON 2000] apresentam uma abordagem baseada na observação de que é possível calcular uma aproximação conservativa da sombra para uma célula de visualização a partir de um conjunto discreto de pontos localizados nas bordas da célula de visualização. Uma condição necessária, para que um objeto esteja oculto é que esteja completamente contido na intersecção da sombra dos pontos amostrados. Esta condição não é suficiente pois podem existir posições de visualização entre os pontos amostrados na qual o objeto considerado é visível. Contudo, o encolhimento de um obstáculo por um valor de escala **E** resulta em uma sombra com uma propriedade interessante: um objeto classificado como oculto pelo obstáculo encolhido permanecerá oculto com respeito ao obstáculo original (não encolhido) enquanto a posição do observador não se mover mais do que **E** com relação a sua posição original. Conseqüentemente, um ponto amostrado utilizado em conjunto com um obstáculo encolhido é uma aproximação

conservativa para uma célula de visualização pequena com raio **E** centrada no ponto amostrado.

Koltun et al. [KOL 2001] introduzem um método que melhora drasticamente as técnicas baseadas em regiões para cenas representadas por modelos 2.5D, tanto em termos de precisão quanto de velocidade. O método utiliza uma transformação de dualidade que possibilita a representação da visibilidade em um espaço bidimensional, o que permite o uso de hardware gráfico para acelerar os cálculos envolvidos. É utilizado um servidor de visibilidade que realiza o cálculo dos PVS das regiões adjacentes ao usuário e envia a geometria contida nesse conjunto de PVS antes que o usuário saia da região em que se encontra. Segundo [KOL 2001] a velocidade e precisão do algoritmo tornam desnecessário o pré-processamento da cena e armazenamento do PVS no servidor. Uma importante vantagem desse método é que ele mantém sua velocidade e precisão mesmo quando aplicado a grandes células de visualização. Uma vez que o algoritmo descrito por esse trabalho foi implementado e utilizado para determinação da visibilidade entre regiões das cenas nesta dissertação (servindo de base para a determinação dos SVS) ele será detalhado na Seção 2.3.

Cohen-Or et al. [COH 2003] realizaram uma comparação entre diversos algoritmos de visibilidade com relação a diversos parâmetros. O quadro comparativo pode ser visto na Tabela 2.1.

TABELA 2.1 – Tabela comparativa dos algoritmos de visibilidade [COH 2003], p.2

Método	2D/3D	Conservativo	Obstáculos	Fusão	Pré-computação	Hardware	Dinâmico
Métodos baseados em regiões que exploram estruturas de células e portais							
Airey ‘90	2D & 3D	sim ou não	portais		PVS	não	não
Teller ‘91	2D & 3D	sim	portais		PVS	não	não
Métodos baseados em pontos que trabalham com precisão de objeto							
Luebke & Georges ‘95	3D	sim	portais		conectividade de células	não	(1)
Coorg & Teller ‘96	3D	sim	grandes e convexos	se tiver contorno convexo	seleção de obstáculos	não	(1)
Hudson et al. ‘97	3D	sim	grandes e convexos	não	seleção de obstáculos	não	(1)
Brittner at al. ‘98	3D	sim	grandes	sim	seleção de obstáculos	não	(1)
Klosowski & Silva ‘00	3D	não	todos	sim	volumétrico	não	(1)
Métodos baseados em pontos que trabalham com precisão de imagem							
Greene et al. ‘93	3D	sim	todos	sim	não	precisa de leitura do *Z-Buffer*	sim, coerência temporal
Zhang et al. ‘97	3D	sim ou agressivo	grade subconjunto	sim	banco de dados de obstáculos	*Z-Buffer* e *Texture-Mapping*	(1)
Bernardini at al. ‘00	3D	sim	pré-processado	sim	obstáculos	não	não
Bartz at al. ‘98	3D	não	todos	sim	não	*buffer* secundário	(1)
Klosowski & Silva ‘01	3D	sim	todos	sim	volumétrico	sim	(1)
Wonka et al. ‘99	2.5D	sim	grande subconjunto	sim	não	*Z-Buffer*	sim
Métodos baseados em regiões que trabalham com cenas arbitrárias							
Schaufler et al. ‘00	2D & 3D	sim	todos	sim	PVS	não	(2)
Durand et al. ‘00	3D	sim	grande subconjunto	sim	PVS	sim	(2)
Koltun et al. ‘00	2D & 2.5D	sim	grande subconjunto	sim	obstáculos virtuais	não	(2)
Wonka et al. ‘00	2D	sim	grande subconjunto	sim	PVS	sim	(2)
Koltun et al. ‘01	2.5D	sim	todos	sim	não	sim	(1)

(1) Necessita de atualização da hierarquia
(2) Volume da movimentação para objetos

2.3 Hardware-Accelerated From-Region Visibility Using a Dual Ray Space

O trabalho de [KOL 2001] apresentou uma solução inovadora para o cálculo da visibilidade entre regiões em ambientes 2.5D. A idéia básica é que raios de visibilidade entre o topo de duas regiões podem ser mapeados em pontos em uma imagem. Esse algoritmo permite uma discretização do problema da visibilidade, e a utilização de hardware gráfico padrão para desenho de imagens 2D é trivial.

Uma grande vantagem desse algoritmo é que ele pode ser aplicado em células de visualização que são muito maiores do que os obstáculos individuais. Em outras palavras, ele executa a fusão de obstáculos implicitamente (Figura. 2.2). Outra vantagem é que sua velocidade possibilita que o cálculo de visibilidade seja realizado em tempo real, sem necessidade de pré-processamento e armazenamento do PVS em arquivos, com isto permitindo que objetos sejam dinamicamente adicionados ou removidos da cena.

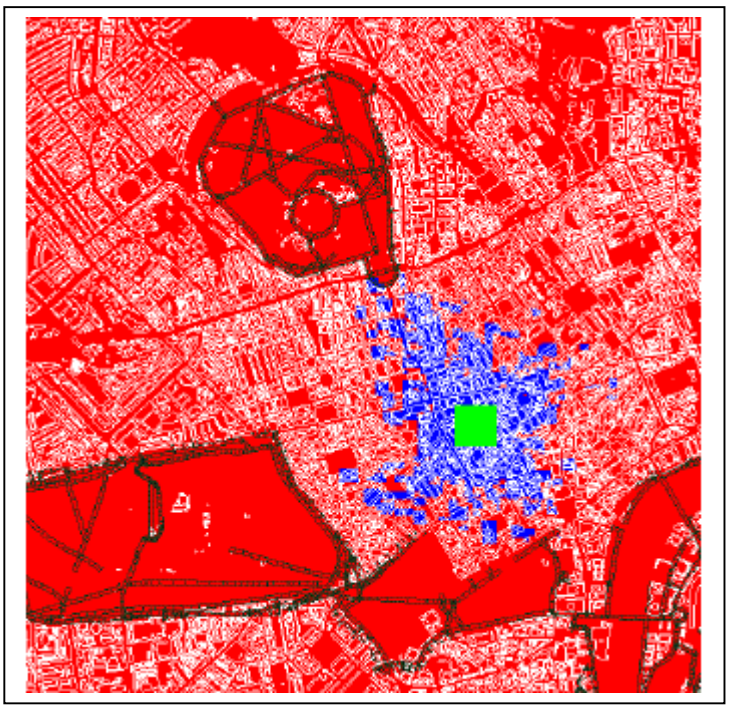


FIGURA 2.2 – Visibilidade determinada a partir de uma região (representada em verde) [KOL 2001] p.215

Foi utilizada uma *kd-tree* padrão para o particionamento da cena em regiões. Cada nodo armazena todos os objetos que estão dentro do volume (mesmo que parcialmente). O algoritmo utiliza a estrutura da *kd-tree* para eliminar grandes partes da cena durante a etapa de determinação de visibilidade entre células. Quando a visibilidade de uma célula está sendo determinada, a *kd-tree* é percorrida de cima para baixo. Em cada nodo, a visibilidade entre a célula de origem e a *bounding-box* do nodo é testada. Se não existir visibilidade, então a recursão termina, pois todos os filhos daquele nodo estarão ocultos.

2.3.1 Cálculo de visibilidade entre duas regiões

O algoritmo de cálculo de visibilidade entre duas regiões reduz o problema da determinação da visibilidade a um problema planar. Dado um conjunto de obstáculos, duas regiões serão visíveis entre si se existir um raio de visibilidade entre elas que seja disjunto dos obstáculos.

Como as técnicas de determinação de visibilidade exatas não são rápidas o suficiente, é utilizado um algoritmo conservativo para determinação da visibilidade [COH 98] [DUR 2000] [KOL 2000] [SCH 2000] [WON 2000]. Esse tipo de algoritmo discretiza ou simplifica o problema da determinação de visibilidade de alguma maneira. Uma forma de simplificação (utilizada neste trabalho) é assumir que a cena (mais precisamente o conjunto de obstáculos) é 2.5D [COH 98] [KOL 2000] [WON 2000].

Em [KOL 2001] foi provado que a visibilidade entre duas regiões pode ser simplificada para a determinação da visibilidade entre seus limites superiores caso os obstáculos sejam do tipo 2.5D. Para a determinação da visibilidade entre esses limites, é construído um volume chamado *shaft* a partir de 6 planos de suporte entre as duas regiões: um plano de suporte superior, um inferior, dois laterais e dois de separação, como pode ser visto na Figura 2.3 onde são indicados por **T**, **V**, **L**, **R**, **Fa** e **Fb** respectivamente.

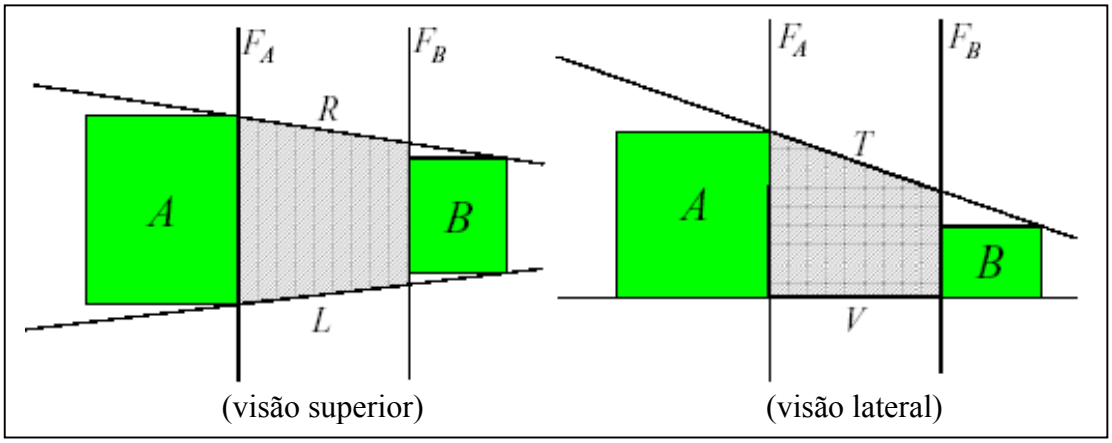


FIGURA 2.3 – Construção do *shaft* ***AB*** [KOL 2001] p.209

Para se determinar a visibilidade entre duas regiões, utiliza-se o polígono que representa o topo do *shaft*. É definido um espaço bidimensional limitado, chamado de *dual ray space*, de modo que todo o raio que inicia na parte do shaft pertencente a região de origem e termina na parte de shaft pertence a região de destino será mapeado para exatamente um ponto nesse espaço dual. O algoritmo irá "marcar" todos os pontos no *dual ray space* que representam raios que passam através de obstáculos. A visibilidade é, então, determinada pela existência de pelo menos um ponto não "marcado".

Para cada obstáculo, é desenhado um polígono que representa todos os raios de visibilidade ocultos por aquele obstáculo. O polígono bidimensional desenhado para um obstáculo **v** será chamado de **T(v)**. Ess polígono geralmente é um trapézio (Figura 2.4a) ou um duplo triângulo (Figura 2.4b), dependendo se a linha que contém o obstáculo intersecta o começo e o fim do topo *shaft* ou não. Depois que todos os polígonos que representam oclusão foram desenhados (um para cada obstáculo que intersecta o topo

do *shaft*), verifica-se a existência de alguma parte não desenhada no *dual ray space*. Se todo o *dual ray space* houver sido preenchido (Figura 2.4c) então as duas regiões não são visíveis entre si. Caso contrário (Figura 2.4d) elas serão visíveis.

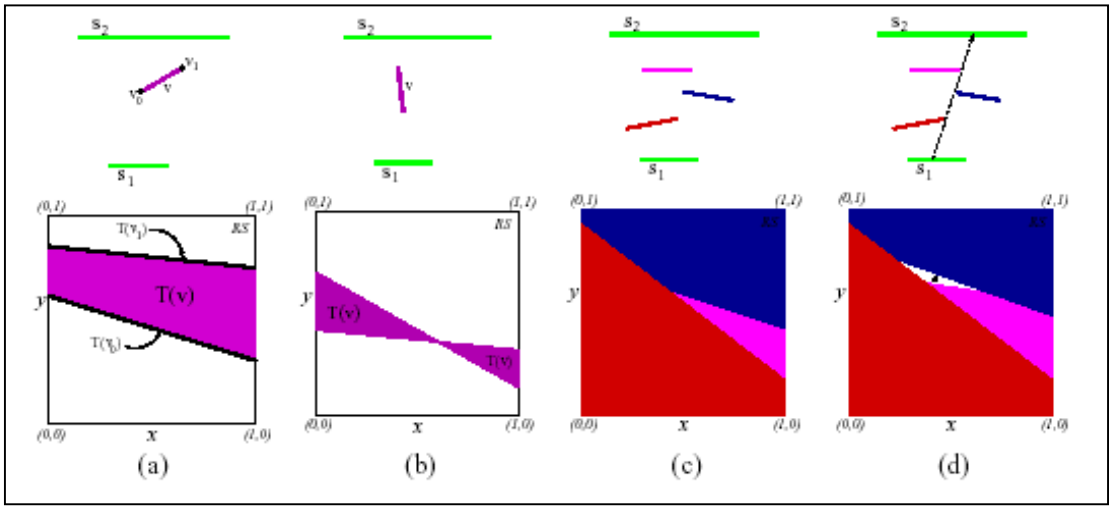


FIGURA 2.4 – Cenas simples no topo e sua representação no *dual ray space*. Nesta figura cada trapézio no *dual ray space* tem a cor do segmento ao qual corresponde. (a) e (b) mostram um obstáculo. Em (c) s2 é oculto a partir de s1 e visível em (d); o ponto preto no *dual ray space* de (d) corresponde ao raio de visibilidade tracejado. [KOL 2001] p.211

Como o comportamento básico da OpenGL é colorir um pixel se o seu centro estiver no interior de um polígono desenhado, deve ser realizado um ajuste para evitar imprecisões. Isto ocorre porque quando é gerada a imagem bidimensional, diversos raios de visão são mapeados nos mesmos pixels. Como o preenchimento de um pixel indica que não existe visibilidade através de nenhum dos raios por ele representado, deve ser realizado um "encolhimento" dos polígonos da imagem bidimensional antes de sua "renderização". Suas arestas são deslocadas para dentro por $\sqrt{2}a$, onde **a** é metade do tamanho do *pixel* (Figura 2.5).

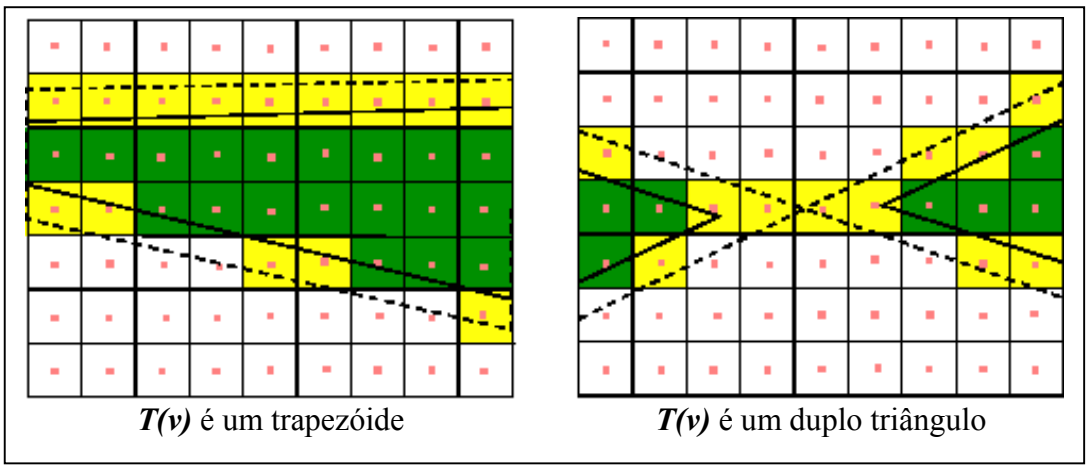


FIGURA 2.5 – Encolhimento dos duplos triângulos e trapézios. Os *pixels* verdes estão completamente contidos em ***T(v)*** (mostrado em tracejado), e seus centros estão contidos no ***T(v)*** encolhido (mostrado em sólido). Os *pixels* amarelos estão apenas parcialmente contidos em ***T(v)*** mas seus centros estão contidos em ***T(v)***; o encolhimento evita que eles sejam coloridos. [KOL 2001] p. 212

A verificação da existência de um *pixel* não preenchido na imagem final pode ser acelerada pela utilização da operação *minmax* do OpenGL 1.2. Esta operação permite uma rápida determinação do cor mínima e máxima presente no *frame-buffer*. Se o *frame-buffer* for completamente colorido de preto e então todos os polígonos **T(v)** forem desenhados de branco, pode-se utilizar essa função para determinar se ainda existe um pixel preto (não preenchido) ou não. Contudo, como será visto na Seção 4.4, existem outras alternativas para determinação da cor mínima da imagem que podem ser mais eficientes, dependendo do hardware gráfico utilizado.

3 Smart Visible Sets

Os **Smart Visible Sets** (SVS) são uma forma de armazenamento de informação de visibilidade alternativa aos PVS. Os SVS baseiam-se na divisão do espaço de uma cena em regiões e no cálculo da visibilidade entre essas regiões. Uma subdivisão adicional dessas regiões é realizada usando o ângulo de visão do observador e a distância entre as células de forma a refinar a informação de visibilidade.

Os SVS podem ser vistos como uma generalização dos PVS, para conter informações adicionais, além da visibilidade entre regiões. Essa informação adicional permite que escolhas mais precisas sejam feitas sobre quais regiões da cena são relevantes. Os SVS ampliam o conceito de visibilidade binária dos PVS do tipo "visível" / "não-visível" para informações do tipo "dentro do campo de visão" / "fora do campo de visão" e "longe" / "perto".

Outra característica importante é que diferentes tipos de SVS podem ser combinados para permitir um controle ainda mais preciso das regiões de interesse da cena. Assim como os PVS, os SVS podem ser calculados em uma etapa de pré-processamento. O cálculo dos SVS depende da existência de dados espaciais de informação de visibilidade entre as regiões de uma cena. Existem diversas estruturas que permitem a divisão de uma cena em regiões, como por exemplo, *Octrees* [SUT 99] e *BSP-Trees* [FUC 80]. Neste trabalho, foi utilizada a ferramenta QBSP3 [ID 2001], que a partir de um conjunto de volumes, gera uma estrutura *BSP-Tree* onde as folhas representam regiões não-convexas. O cálculo de visibilidade entre essas regiões corresponde exatamente ao cálculo dos PVS, discutido no Capítulo 2. Inicialmente foi utilizada a ferramenta QVIS para a geração dos PVS, posteriormente sendo implementado o algoritmo *Dual-Ray Space* [KOL 2001] para cálculo de visibilidade, como será detalhado no Capítulo 4.

O foco deste trabalho está em cenas do tipo 2.5D, especialmente cidades. Nesse tipo de ambiente o uso do ângulo de visão do observador como parâmetro para subdivisão da visibilidade é reduzido a um problema planar. Essa característica dos cenários utilizados não impede que o conceito dos SVS seja explorado.

Na próxima seção será explicado como o SVS baseado no particionamento do ângulo de visão é calculado. Os algoritmos relevantes serão detalhados, especialmente os algoritmos que tratam da escolha dos particionadores da visibilidade. A seguir o SVS baseado no particionamento da distância será abordado. Na última seção deste capítulo serão detalhados os diferentes usos do SVS e como diferentes SVS podem ser combinados.

3.1 SVS-Angle

Uma das formas de subdividir a informação de visibilidade é quebrar o hemisfério de direções de visualização em diferentes grupos. Nos PVS a região em que a câmera se encontra define o índice para acessar a visibilidade pré-computada e recuperar as regiões visíveis. Essa visibilidade não depende da orientação ou do campo de visão, isto é, corresponde à visibilidade de um observador em qualquer ponto daquela região com um campo de visão de 360^{o}.

A subdivisão da visibilidade de uma região através do particionamento do campo de visão gera diferentes conjuntos de visibilidade, um para cada partição. Cada

um desses conjuntos corresponde à visibilidade obtida por um observador em qualquer ponto daquela região com um campo de visão igual ao da partição utilizada.

Como se trabalhou com cenas 2.5D, foi utilizada apenas a informação do azimute para particionar a visibilidade, sem levar em consideração a elevação ou inclinação da câmera. Na Figura 3.1(a) é apresentada uma região central representada por uma circunferência e as regiões visíveis a partir daquela região representadas por quadrados. Esta é exatamente a informação contida em um PVS. Já um SVS-Angle irá dividir a informação de visibilidade em subconjuntos (três subconjuntos no caso da Figura 3.1(b)). Elementos que são cortados pela fronteira dos ângulos são incluídos em ambos os conjuntos.

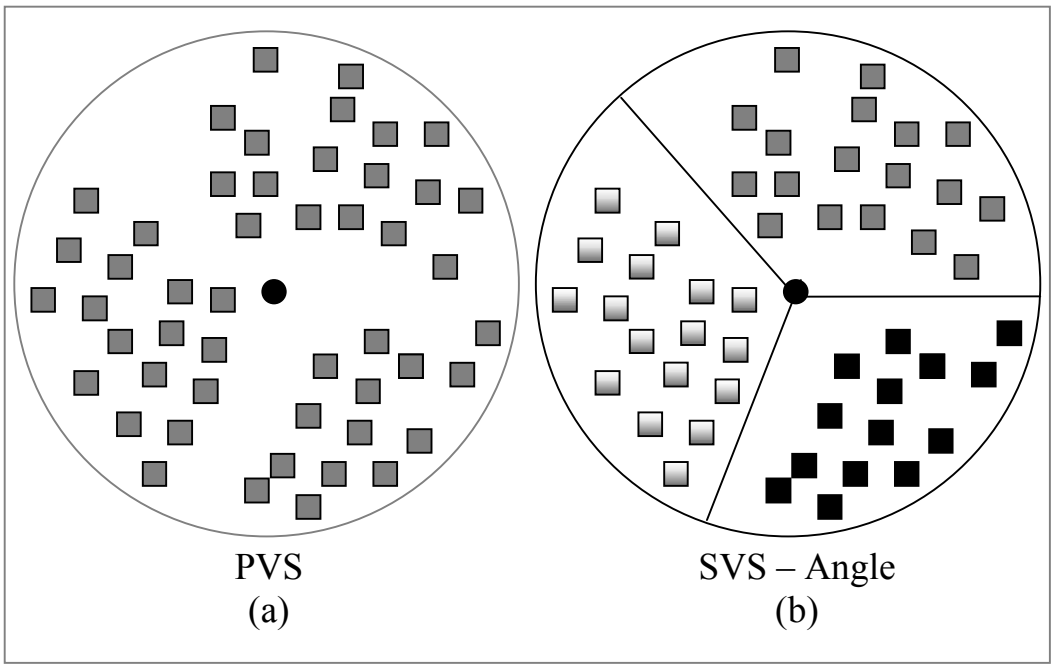


FIGURA 3.1 – Visibilidade de uma região: (a) PVS, observado localizado no centro e regiões visíveis representadas por quadrados; (b) subdivisão do PVS em 3 subconjuntos utilizando o campo de visão

Seja uma região **r** pertencente a uma cena. Se a informação de visibilidade para aquela região for não vazia, então ela pode ser dividida através de um conjunto de particionadores. Cada elemento do conjunto de particionadores é um par de ângulos que representa um campo de visão e irá gerar um novo conjunto de visibilidade. Para se particionar a visibilidade dessa região a partir de um conjunto de partições do campo de visão definido pelo conjunto de ângulos **A**, onde cada elemento é definido pela tupla (**ai**, **af**) (ângulo inicial e ângulo final) segue-se os seguintes passos:

1. Calcular os *Bounding-Volumes* de todas as regiões da cena
2. Calcular a *Bounding-Box* da região **r**
3. Para cada elemento de **A**:
    a. Criar 2 equações de reta **e1** e **e2** que representam uma abertura do campo de visão desde o ângulo **ai** ao ângulo **af**.
    b. Classificar os *Bounding-Volumes* de todas as regiões visíveis quanto à **e1** e **e2**

c. Armazenar a nova informação de visibilidade

Os passos acima serão detalhados nas próximas seções.

3.1.1 Calculando os *Bounding-Volumes*

No presente trabalho foi usada uma representação do tipo *BSP-Tree* da cena, onde as regiões correspondem a folhas nesta árvore. O cálculo dos *Bounding-Volumes* destas regiões segue um procedimento recursivo, chamado `Calc_BVolume`.

A função auxiliar `Split_Volume` divide um volume em dois utilizando um plano que é passado como o 2º parâmetro. Os dois volumes resultantes são armazenados nos 3º e 4º parâmetros.

```
Calc_BVolume (Node: node, BVolume: volume)
{
      BVolume1, Bvolume2: volume
      If Node = leaf
      Then Store BVolume
      Else
            Split_Volume (BVolume, Node.SplitPlane, BVolume1, BVolume2)
            Calc_BVolume(Node.leftSon, BVolume1)
            Calc_BVolume(Node.rightSon, BVolume2)
      EndIf
}
```

CÓDIGO 3.1 – Função de cálculo de bounding-volumes

Inicia-se essa função passando como parâmetro a raiz da *BSP-Tree* e um *Bounding-Volume* que contém toda a cena. A árvore é, então, percorrida da esquerda para a direita. A cada nodo não-folha o volume é dividido em dois (usando o plano particionador armazenado no nodo). São realizadas então duas novas chamadas de função usando os filhos do nodo e o resultado da divisão do volume como parâmetros. Quando uma folha é alcançada, o *Bounding-Volume* daquela região é armazenado em uma matriz. No final, teremos os *Bounding-Volumes* de todas as folhas armazenados nesta matriz.

3.1.2 Calculando a *Bounding-Box*

O cálculo da *Bounding-Box* de uma região **r** é trivial uma vez calculado o seu *Bounding-Volume* no passo anterior. Basta percorrer os vértices de todos os polígonos que formam o *Bounding-Volume* de **r** armazenando os máximos e mínimos das coordenadas X, Y e Z.

3.1.3 Calculando a Nova Visibilidade

Uma vez calculada a *Bounding-Box* de uma região pode-se calcular um novo conjunto de visibilidades para cada elemento do conjunto de particionadores **A**. Cada

elemento de **A** é um conjunto de dois ângulos que indicam os limites de um campo de visão.

Para gerar as equações de reta **e1** e **e2** que representam um campo de visão utilizamos as coordenadas X e Z de um dos vértices da *Bounding-Box* e os ângulos que compõem o elemento de **A** (um ângulo para cada reta). O vértice a ser considerado para a geração da equação da reta varia de acordo com o valor do ângulo e se o ângulo representa o início ou o fim do campo de visão (Figura 3.2). Nessa mesma figura tem-se a escolha dos pontos corretos para uma subdivisão do campo de visão definida por três conjuntos: $\{0^o, 45^o\}$, $\{45^o, 180^o\}$ e $\{180^o, 0^o\}$.

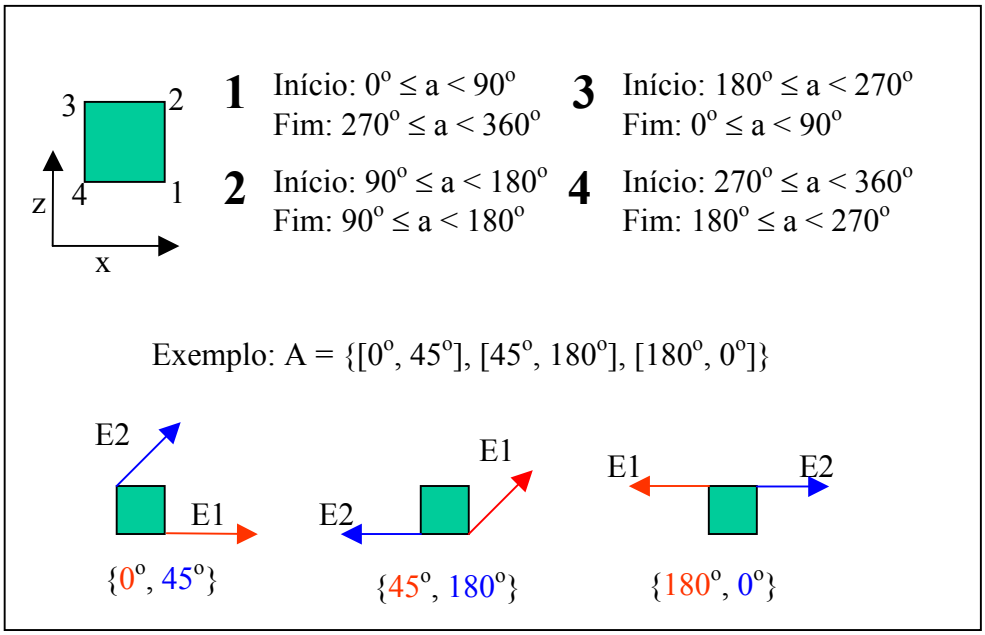


FIGURA 3.2 – Escolha dos vértices corretos da *Bounding-Box* para a geração das retas e1 e e2

Para gerar uma equação da reta a partir de um ponto **p** e um ângulo **a**, usa-se a equação reduzida da reta: **y = mx + b**. O valor do coeficiente angular **m** corresponde à tangente de **a**. Aplicando o valor de **m** e substituindo os valores de **y** e **x** pelas coordenadas do ponto **P** encontra-se o valor do coeficiente linear b. Com os valores de **m** e **b** pode-se montar a equação da reta.

De posse das duas equações de reta e dos *Bounding-Volumes* de todas as regiões da cena pode-se calcular a nova visibilidade através do seguinte algoritmo:

```
For each visible region
For each polygon that belongs to the Bounding-Volume of that region
      For each edge of that polygon
            Classify_Edge (edge, equation1, equation2)
If any edge was classified as INSIDE
Then Region is Visible
Else Region is Not Visible
EndIf
```

CÓDIGO 3.2 – Algoritmo para cálculo da nova visibilidade

A função `Classify_Edge` verifica se algum ponto da aresta passada como 1º parâmetro se encontra dentro do espaço delimitado pelas duas equações de reta passadas como 2º e 3º parâmetros. Caso afirmativo, a função retorna o valor `DENTRO`, caso contrário o valor retornado será `FORA`.

Após calculada, a nova visibilidade é armazenada em uma matriz, de forma similar ao PVS, acrescentada da subdivisão do campo de visão usada. Como cada particionador presente no conjunto **A** irá gerar uma matriz diferente, o tamanho da nova informação de visibilidade para cada região será:

$$Visibilidade = \mathrm{r} \times |A| \times X \qquad \text{(Eq. 3.1)}$$

onde **r** representa o número de regiões da cena, |**A**| representa a cardinalidade do conjunto **A** e **X** representa a quantidade de informação necessária para se armazenar a visibilidade referente a uma região.

Como as informações de visibilidade são armazenadas como matrizes de bits, onde cada bit ligado indica uma região visível, o valor de **X** será 1 bit, permitindo a simplificação do cálculo do tamanho da nova visibilidade para:

$$Visibilidade = r \times |A| bits \qquad \text{(Eq. 3.2)}$$

Um exemplo do armazenamento de um SVS gerado a partir de um conjunto particionador **A** = {(0º , 120º), (120º , 240º), (240º , 0º)} pode ser visto na Figura 3.3 onde uma cena com 10 regiões teve sua visibilidade particionada.

3.1.4 Escolhendo os Particionadores

Uma questão que ainda deve ser resolvida é como escolher os elementos do conjunto particionador S. Diferentes conjuntos **A** irão gerar diferentes matrizes de visibilidade. Escolher quais ângulos devem ser utilizados como particionadores não é o único problema. Também deve ser escolhido quantos serão os elementos do conjunto **A**. Quanto maior o conjunto, maior será a subdivisão da visibilidade, mas também maior será o espaço necessário para se armazenar estas informações.

Foram testadas duas formas de geração do conjunto **A**. Uma fixa, onde um usuário informa quais serão os ângulos que devem ser utilizados no particionamento e outra adaptativa, onde o usuário informa apenas quantos ângulos serão utilizados e um algoritmo faz a escolha de quais serão os ângulos particionadores.

Ângulos Fixos

Nessa abordagem os elementos do conjunto **A** são informados pelo usuário. A visibilidade de cada uma das regiões é particionada segundo o mesmo conjunto **A**. A grande vantagem dessa solução é permitir o uso de quaisquer quantidades e valores de particionadores.

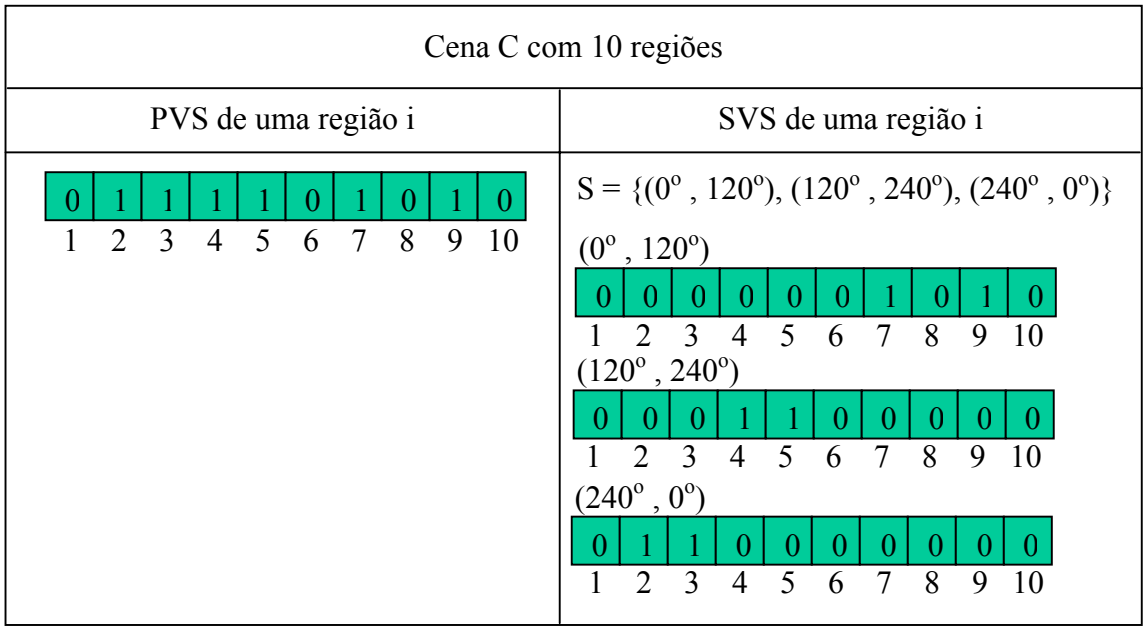


FIGURA 3.3 – Exemplo de um PVS e de um SVS em uma cena com 10 regiões

Inicialmente essa abordagem foi adotada, mas a existência de um conjunto constante de particionadores tornou-se um problema uma vez que muitas das matrizes de visibilidade geradas não eram bem balanceadas. Em muitas regiões, especialmente naquelas localizadas às margens da cena, alguns dos particionadores geram um grande número de regiões visíveis enquanto que outros particionadores geram matrizes com poucas. Na Figura 3.4 pode-se ver um exemplo de visibilidade mal balanceada, com o conjunto representado na cor cinza armazenando a grande maioria das regiões visíveis.

Ângulos Adaptativos

Para solucionar o problema do balanceamento deficiente da visibilidade encontrado na abordagem de divisão por ângulos fixos foi desenvolvido um algoritmo que escolhe quantos serão os elementos de **A** e quantos candidatos serão testados. Esse algoritmo é executado antes do cálculo da nova visibilidade de cada região. Logo, cada região irá possuir um conjunto particionador próprio. Na Figura 3.5 tem-se o exemplo de um particionamento adaptativo da mesma cena vista na Figura 3.4. Pode-se notar que os quatro conjuntos gerados estão bem balanceados quanto ao número de regiões armazenadas.

A função auxiliar `Choose_Candidates` escolhe um número **n** de particionadores (**n** é passado como 2º parâmetro) para dividirem o campo de visão (passado como 1º parâmetro). Os valores escolhidos são armazenados na matriz passada como 3º parâmetro.

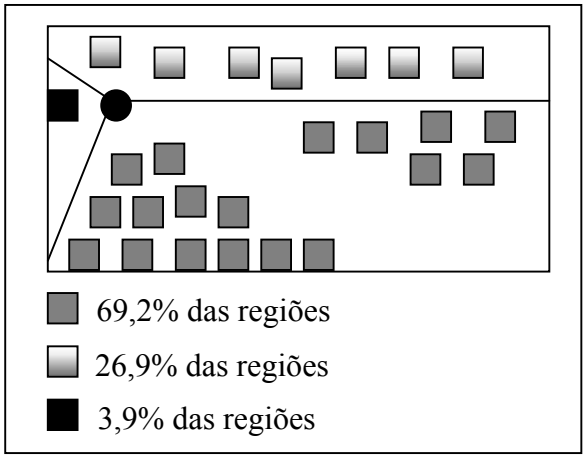


FIGURA 3.4 – Exemplo de matrizes de visibilidade mal balanceadas

A função `Check_Candidate` retorna um valor que indica quão bem balanceados seriam os dois campos de visão resultantes caso o campo de visão passado como 1º parâmetro fosse dividido em dois pelo particionador passado como 2º parâmetro. Esse valor é encontrado através da seguinte fórmula:

$$Result = | NCel_{V1} - NCel_{V2} | \qquad \text{(Eq. 3.3)}$$

onde $NCel_{V1}$ é o número de células visíveis no 1º campo de visão e $NCel_{V2}$ é o número de células visíveis no 2º campo de visão.

Quanto menor o valor de `Result,` melhor balanceada será a divisão.

A função `Split_Frustum` divide o campo de visão passado como 1º parâmetro em dois, utilizando o particionador passado como 2º parâmetro. Os dois campos de visão resultantes são armazenados nos 3º e 4º parâmetros.

A função recursiva que realiza a escolha dos particionadores, `Calc_Splitters,` recebe o campo de visão total (0º, 360º), a profundidade e número de candidatos.

```
Calc_Splitters (Frustum: frustum, int: depth, int: candidates)
{
int: bestResult, i, result
float: bestSplitter
float: candidate_array[MAX_CANDIDATES]
If depth = 0
Then
      Store frustum1;
      Store frustum2;
EndIf
Choose_Candidates (frustum, candidates, candidate_array)
For i = 0 until i < candidates do
      result = Check_Candidate (frustum, candidate_array[i])
      If result < bestResult
      Then
            bestResult = result
            bestSplitter = candidate[i];
      EndIf
EndFor
Split_Frustum (frustum, bestSplitter, frustum1, frustum2)
Calc_Splitters(frustum1, depth-1, candidates)
Calc_Splitters(frustum2, depth-1, candidates)
}
```

CÓDIGO 3.3 – Função de cálculo de ângulos particionadores

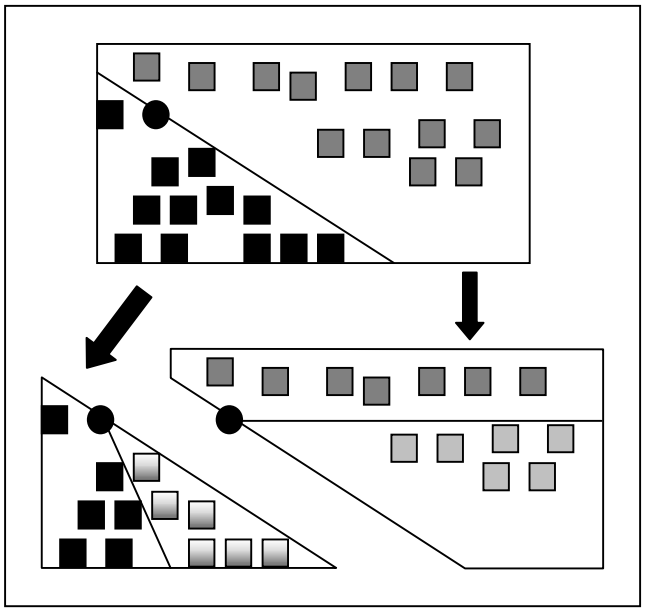


FIGURA 3.5 – Exemplo de utilização de ângulos adaptativos

3.2 SVS-Distance

Outra forma de subdividir a informação de visibilidade é usar a distância entre as regiões. Os novos conjuntos de visibilidade gerados indicam não somente quais regiões são visíveis a partir de um determinado ponto, mas as distâncias em que se encontram as regiões visíveis. Esse tipo de informação é extremamente importante em cenas representadas usando o conceito de níveis de detalhe (LOD – *Level of Detail*), ou onde o corte de informações é necessário.

A subdivisão da visibilidade de uma região através da distância irá gerar um conjunto de visibilidade para cada distância. Cada conjunto armazena as regiões que estão localizadas a uma distância mínima do observador, como pode-se observar na Figura 3.6. A definição do conjunto de distâncias particionadoras **D** utilizado para dividir a visibilidade é definida pelo usuário.

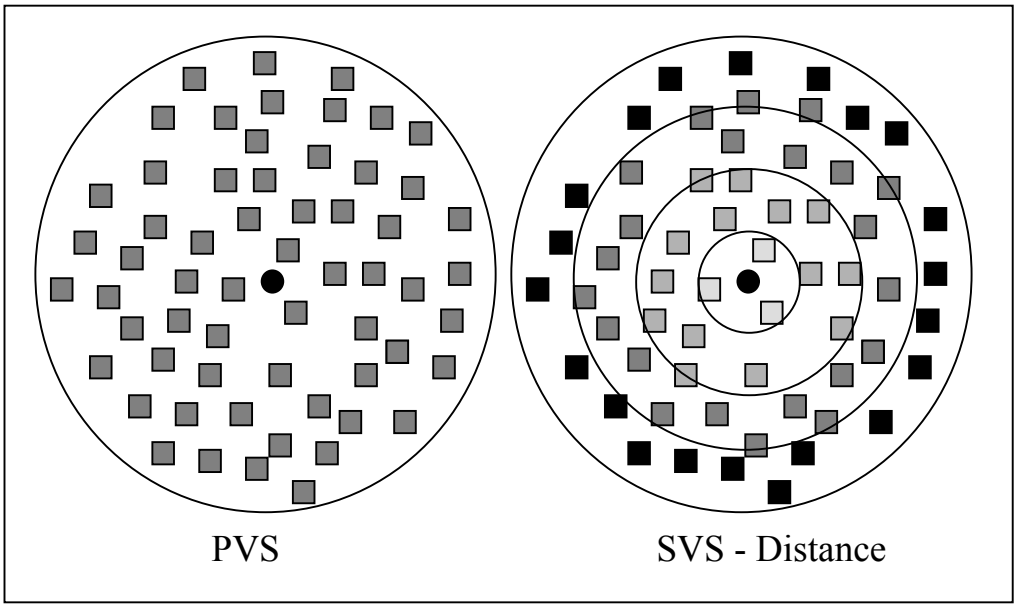


FIGURA 3.6 – Particionamento da visibilidade de uma região através da distância

O cálculo de distância entre regiões é bastante simplificado pelo cálculo dos *Bounding-Volumes* e *Bounding-Boxes*, já discutido na seção anterior. Uma das maneiras mais simples de estimar a distância entre duas regiões é utilizar a distância entre os centros das *Bounding-Boxes* das duas regiões..

Contudo, essa solução apresenta problemas, pois o valor de distância desejado é a distância mínima entre as regiões. A Figura 3.7 mostra duas situações de cálculo de distância entre regiões utilizando o centro de *Bounding-Boxes*. O segundo exemplo apresenta uma grande discrepância entre a distância entre os centros das *Bounding-Boxes* e a distância mínima entre as regiões.

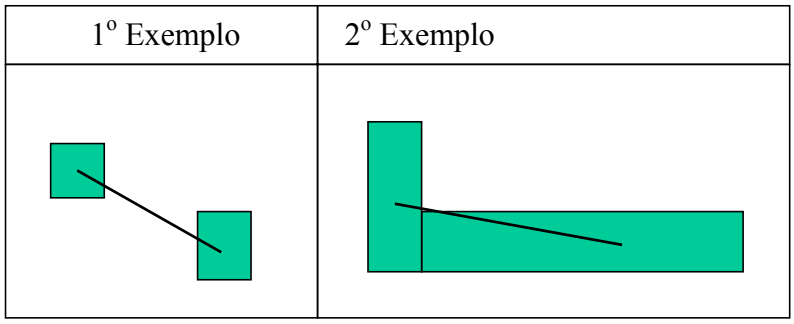


FIGURA 3.7 – Aproximação da distância mínima entra regiões

Uma solução mais precisa é comparar as distâncias entre as faces dos *Bounding-Volumes* das regiões e utilizar a menor distância encontrada. A nova visibilidade também é armazenada na forma de matrizes de bits. Como cada elemento no conjunto **D** irá gerar uma matriz diferente, o tamanho da informação de visibilidade particionada por distância será:

$$Visibilidade = r \times |D| \, bits \qquad \text{(Eq. 3.4)}$$

onde **r** representa o número de regiões da cena e | **D** | representa a cardinalidade do conjunto **D.**

3.3 Utilização do SVS

Uma vez calculados, tanto os SVS baseados em campos de visão ou distâncias podem ser utilizados separadamente ou combinados. Um dos usos mais comuns do SVS baseado em campos de visão é descobrir quais regiões são visíveis a partir de uma determinada posição e ângulo de visão.

Para gerar uma matriz com as regiões visíveis dados uma posição e um campo de visão segue-se os seguintes passos:

- Usa-se a posição da câmera para calcular a região que ela se encontra (utilizando para isso a *BSP-Tree*)
- Realiza-se uma operação lógica OR entre todas as matrizes de visibilidade daquela região cuja informação de particionamento interceptar o campo de visão dado

As chaves da Figura 3.8 demonstram as operações realizadas para se descobrir as regiões visíveis a partir de dois campos de visão (80^{o}-170^{o} e 190^{o}-280^{o}) em um SVS que foi criado a partir de um conjunto S de cardinalidade 3.

Uma outra possibilidade para o uso dos SVS é combiná-los com estratégias de nível de detalhe, onde objetos podem ser armazenados em diferentes resoluções. Antes da geração da imagem, pode-se rapidamente verificar em qual região da cena esses objetos se encontram e, com o uso de SVS baseados em distância, decidir qual a resolução a ser utilizada.

O uso mais importante dos SVS é o ordenamento de informações. Através da combinação de diferentes SVS pode-se rapidamente determinar quais regiões estão mais próximas de um cliente e/ou dentro do seu campo de visão.

Em ambientes do tipo cliente-servidor esse ordenamento de informações é de grande valia. A informação classificada como “mais importante” pode receber maior prioridade de envio aos clientes. Se a houver necessidade do corte de parte das informações que estão sendo enviadas devido a sobrecarga da rede ou estouro da quota de banda de um cliente, a informação “menos importante” pode ser cortada.

Outra vantagem dos SVS é que o ordenamento de informações pode ser feito de forma dinâmica. Através de mudanças no tipo de operações realizadas entre os SVS ou de mudanças nos operandos, o tipo de informação resultante pode ser rapidamente alterado. Por exemplo, uma cena pode ser enviada a um cliente de acordo com a velocidade de sua conexão. Se a velocidade for muito baixa, pode-se utilizar a combinação de um *SVS-Angle* e um *SVS-Distance* para enviar apenas os dados que estão próximos e dentro do campo de visão do cliente.

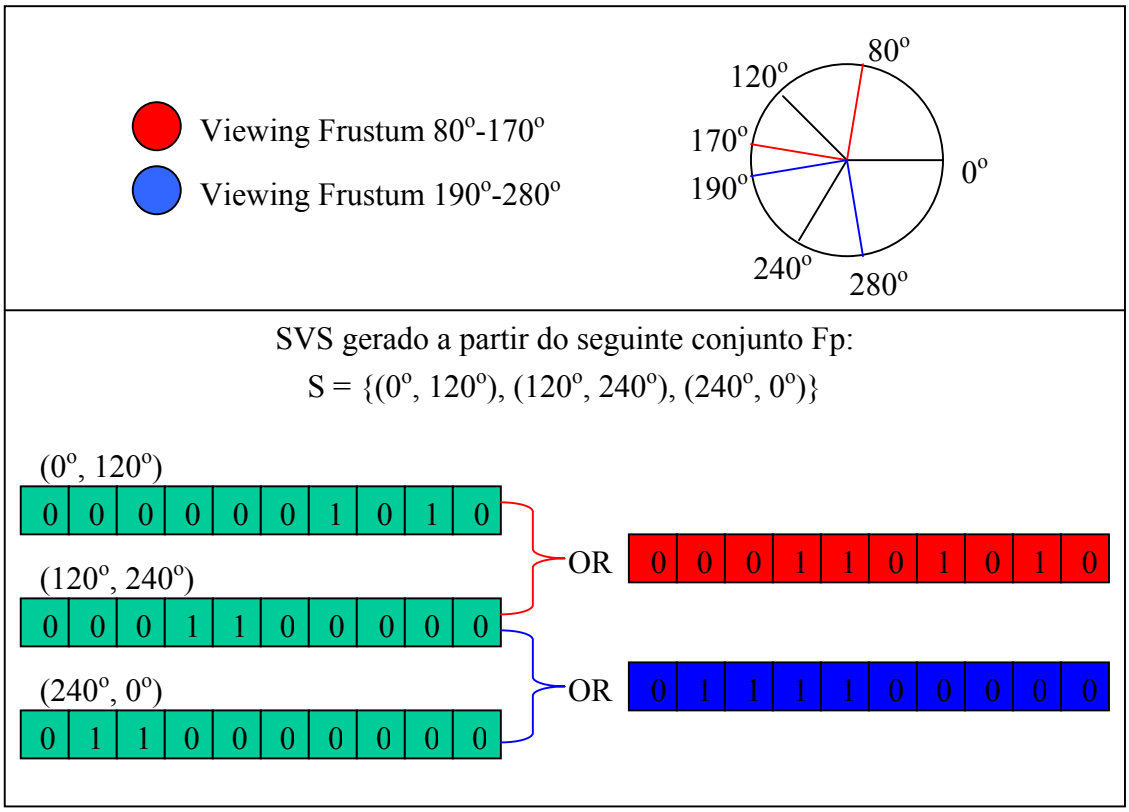


FIGURA 3.8 – Combinando SVS através de operações OR

4 BSP Viewer

A realização de experimentos com os algoritmos de geração de SVS foi desenvolvida em ambiente composto de:

- Gerador de cenas 3D
- Gerador de cenas em *BSP-Trees*
- Módulo de cálculo de visibilidade entre regiões
- Módulo de cálculo de SVS
- Ambiente de experimentação

Para a criação das cenas 3D foi desenvolvido um aplicativo chamado CityGenerator, que gera uma cidade com um número de prédios definidos pelo usuário. Para o particionamento da cena em diferentes regiões foi utilizado o aplicativo QBSP3 [ID 2001]. Para o cálculo de visibilidade utilizou-se inicialmente o aplicativo QVIS [ID 2001], mas devido às limitações de performance foi desenvolvida uma nova aplicação FastPVS que utiliza o algoritmo *dual ray space* [KOL 2001] para a geração da informação de visibilidade de maneira mais eficiente. A geração dos SVS é feita pelo aplicativo SVS que implementa os algoritmos descritos no Capítulo 3.

Foi desenvolvido também um ambiente gráfico BSPViewer que permitiu a execução de testes para se verificar a correção da visibilidade gerada pelos aplicativos QVIS, FastPVS e SVS. Esse ambiente permite também a comparação da performance de diversos tipos de visibilidade e a navegação pelo usuário nos ambientes 3D gerados pela aplicação CityGenerator. Na Figura 4.1 podemos observar o fluxograma desse ambiente.

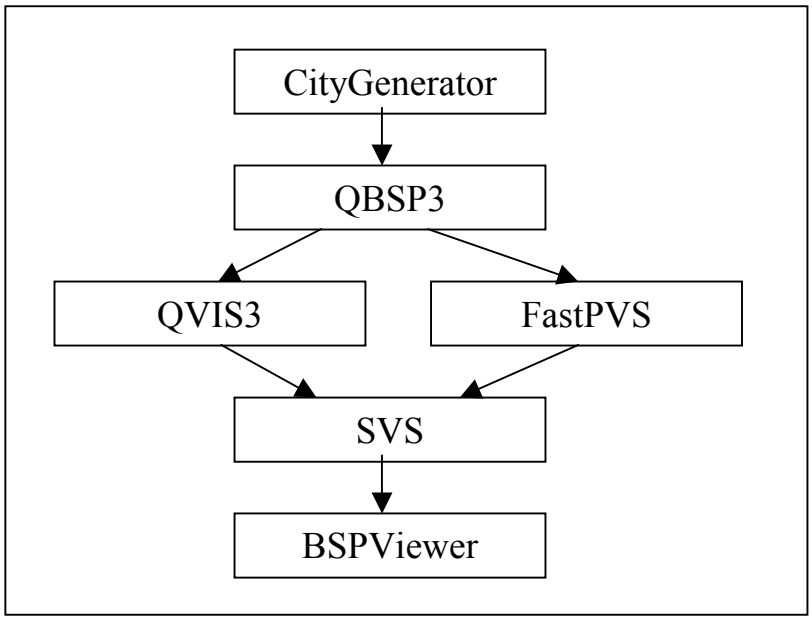


FIGURA 4.1 – Fluxograma do ambiente desenvolvido

Nas próximas seções cada um desses aplicativos será detalhado. A entrada e saída de cada uma das aplicações é definida e os detalhes relevantes na implementação de cada aplicativo são explicados.

4.1 CityGenerator

O primeiro passo na criação de um ambiente que permitisse o teste da estrutura SVS foi o desenvolvimento de uma aplicação que gerasse cenários 3D. Esses cenários são utilizados pela aplicação QBSP3 e representam a base sobre a qual todas as demais aplicações irão trabalhar.

Tendo-se como um dos maiores interesses e motivação deste trabalho as aplicações do tipo passeio virtual, decidiu-se que os ambientes de teste seriam cidades. A aplicação CityGenerator foi então criada para gerar cenários no formato esperado pela aplicação QBSP3, compostas de uma coleção de blocos simples.

Esta aplicação possui um funcionamento bastante simples. O usuário fornece como parâmetro de entrada o número de prédios desejado e a aplicação gera proceduralmente uma cena 3D com o número de prédios especificados. A cena gerada é gravada em um arquivo do tipo MAP, formato utilizado pela aplicação QBSP3. Foi utilizada a aplicação *DeepExploration*[7] para se observar as cenas gearadas. Na Figura 4.2 podemos observar uma cidade com 13500 prédios sendo visualizada.

4.2 QBSP3

O passo seguinte, após a geração de uma cena no formato MAP, é a subdivisão em regiões. Para acelerar o desenvolvimento do sistema, foi utilizado um aplicativo já existente, o QBSP3, baseada na representação da cena através de uma árvore BSP. Essa aplicação foi utilizada em várias jogos comerciais como Quake[5], Half-Life[6] e Daikatana[8].

Esse aplicativo recebe um arquivo MAP como entrada e gera um arquivo com as informações da estrutura da *BSP-Tree* e outro com as informações dos portais, que representam áreas de conectividade entre folhas da árvore. Foi utilizada a opção –nosubdiv para que faces muito grandes não fossem divididas em faces menores. A *BSP-Tree* criada por essa aplicação possui as seguintes características:

- Armazenamento das equações dos nodos particionadores em cada nodo não-folha da árvore;
- Armazenamento da estrutura geométrica nas folhas da *BSP-Tree*;
- *Bounding-box* de cada sub-árvore armazenada no nodo raiz de cada sub-árvore.

[5] http://www.idsoftware.com/games/quake
[6] http://www.half-life.com/
[7] http://www.righthemisphere.com/products/dexp/
[8] http://www.gamespot.com/guides/daikatana_gg/

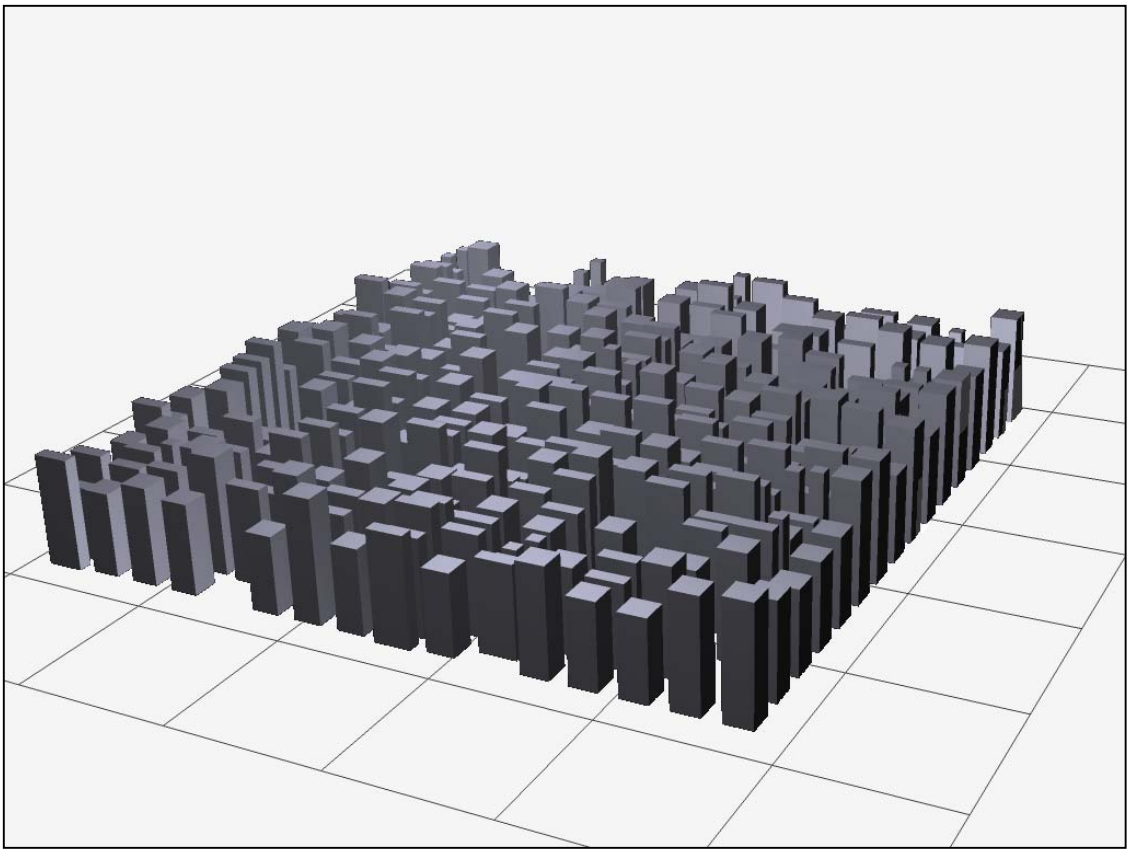


FIGURA 4.2 – Cena com 350 prédios visualizada com a aplicação DeepExploration

Após alguns testes de divisão de cidades em *BSP-Trees* observou-se que as células geradas eram de tamanhos heterogêneos, sendo algumas delas demasiado grandes. Essas células iriam gerar problemas de visibilidade nos próximos passos pois podem ser vistas por muitas células e ao mesmo tempo pode visualizar um grande número de células.

Isso acontece porque o aplicativo QBSP3 utiliza apenas a geometria da cena para realizar o particionamento da *BSP-Tree*. Para resolver esse problema optou-se por alterar o algoritmo de particionamento. A função particionadora foi modificada de:

```
Split_Node (node)
If node = leaf
Then Save leaf
Else
      Split node in two children nodes
      Split_Node (child_1)
      Split_Node (child_2)
EndIf
```

CÓDIGO 4.1 – Função particionadora original

Para:

```
Split_Node (node)
If node = leaf
Then
      If X dimension of the leaf > maximum allowed X value
      Then
            Creates a splitting plane perpendicular to X
            Split the leaf in two children nodes using that plane
            Split_Node (child_1)
```

```
            Split_Node (child_2)
      Else
            If Y dimension of the leaf > maximum allowed Y value
            Then
                  Creates a splitting plane perpendicular to Y
                  Split the leaf in two children nodes using that plane
                  Split_Node (child_1)
                  Split_Node (child_2)
            Else Save leaf
            EndIf
      EndIf
Else
      Split node in two children nodes
      Split_Node (child_1)
      Split_Node (child_2)
EndIf
```

CÓDIGO 4.2 – Função particionadora modificada

Os resultados dessa subdivisão podem ser vistos no Seção 5.1

4.3 QVIS3

O cálculo de visibilidade entre as regiões após a cena ser subdividida através da geração de uma *BSP-Tree* é o próximo passo. Para cada região descobre-se quais outras regiões são visíveis. Optou-se por utilizar uma aplicação já disponível para se realizar esse cálculo, o QVIS3 [ID 2001].

O QVIS3 recebe um arquivo BSP como entrada. Ele lê a informação da *BSP-Tree* e o conjunto de portais calculados pelo QBSP3 e então determina quais folhas são visíveis a partir de cada folha da árvore. Para uma BSP3 com **n** folhas, o PVS consiste de uma matriz de bits **n** por **n**.

Para determinar a visibilidade entre as folhas, o algoritmo determina inicialmente a visibilidade entre portais, ou seja, quais portais são visíveis mutuamente. Para cada dois portais adjacentes (um em uma folha de origem e outro numa folha adjacente), o QVIS3 constrói um volume que passa exatamente entre os dois portais. Ele tenta, então, continuar desenhando esse volume através de todos os portais que puder encontrar naquela direção. O volume é recortado durante este processo, de modo que possa passar exatamente pelos portais. Esse volume descreve a linha de visão do portal original. Quando esse volume não puder mais ser estendido através de outros portais, a visibilidade é bloqueada e cada portal que esse volume atravessou é adicionado ao conjunto de portais visíveis do portal de origem.

Depois que essa etapa termina, cada portal "sabe" quais outros portais podem ser vistos a partir de si. A visibilidade das folhas é, então, calculada através da visibilidade dos portais. Cada portal de cada folha sabe quais outros portais podem ser vistos. Pode-se saber então quais outras folhas são visíveis a partir da folha de origem. A união dos conjuntos de folhas visíveis a partir de cada portal de uma folha é a visibilidade para aquela folha. Uma vez calculado, o *Potentially Visible Set* (PVS) tem seu tamanho reduzido através de compactação e é armazenado no próprio arquivo BSP.

4.4 FastPVS

Apesar do aplicativo QVIS3 gerar a visibilidade de maneira eficiente nas cenas iniciais, ele é ineficiente em cenas maiores. Isso ocorre porque o aplicativo QVIS3 foi desenvolvido tendo como alvo cenas com grande quantidade de oclusão, como calabouços e interiores de ambientes. Em cenas onde não ocorre tanta oclusão (como cidades, por exemplo) o tempo gasto no cálculo da visibilidade pelo aplicativo QVIS3 é muito grande. Isso ocorre porque a performance do algoritmo de cálculo de visibilidade utilizado depende diretamente do número de portais de cada região. Como em cenas com menos oclusão o número de portais é bem grande, o algoritmo torna-se inviável em cenas maiores.

Para permitir que cenas maiores fossem exploradas, foi implementado o algoritmo *Dual-Ray Space* [KOL 2001], já discutido no Capítulo 2. Esse algoritmo apresenta excelente performance pois reduz o problema do cálculo de visibilidade 3D a um problema 2D. A grande restrição desse algoritmo é exigir que os obstáculos sejam do tipo 2.5D. Como as cenas testadas eram de cidades (onde os únicos obstáculos são prédios) esse algoritmo pôde ser utilizado.

Para se testar a existência de um raio de visibilidade é construído um espaço bidimensional (*dual ray space*) de tal modo que cada raio partindo de uma região e chegando em outra corresponda a um ponto no espaço bidimensional. O algoritmo irá marcar todos os pontos que representam raios que passam através de um obstáculo. A visibilidade é detectada checando se pelo menos um dos pontos não foi marcado (ou seja, existe um raio entre as duas regiões que não passa através de nenhum obstáculo).

Na Figura 4.3 podem ser observados os vários passos do algoritmo *dual ray space* (as imagens foram capturadas do aplicativo BSPViewer):

1. Estabelecimento das regiões sobre as quais se deseja calcular a visibilidade (região de origem em amarelo e região de destino em verde)
2. Cálculo dos planos de suporte (para posterior calculo do volume *shaft*)
3. Identificação do topo do *shaft*
4. Cálculo da intersecção entre as faces da cena e o *shaft* (faces que não intersecionam o *shaft* não são mostradas)

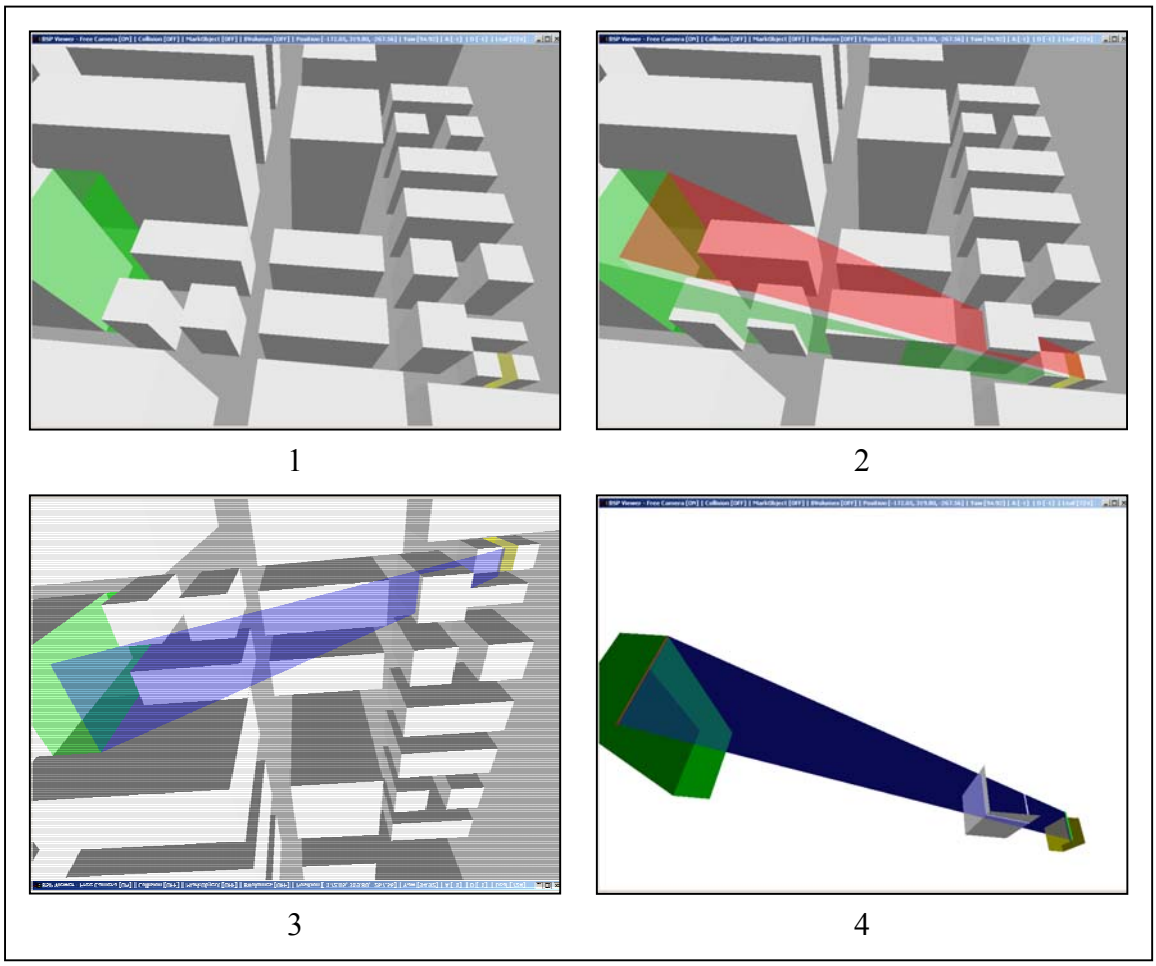


FIGURA 4.3 – Passos do algoritmo d*ual ray space*

Seja o *shaft* entre dois volumes quaisquer **v1** e **v2** chamado de **s1**. Chama-se o seu topo de **t1** e os segmentos de **t1** contidos em **v1** e **v2**, **r1** e **r2,** respectivamente (Figura 4.4 (a)). Sejam as extremidades de **r1** e **r2** chamadas de **r11**, **r12** e **r21** e **r22**. Seja a intersecção entre um obstáculo 3D e **t1** (segmentos de reta) chamada de **o**. Sejam as extremidades de **o** chamadas de **o1** e **o2**. Seja a reta suporte de **o** chamada de **eo**. Para se criar um polígono que represente a oclusão causada por **o** foi utilizado o seguinte algoritmo:

Para cada **o**:

Construir 8 segmentos de reta usando cada uma das extremidades de **r1** e **r2** e cada uma das extremidades de **o**. Esses segmentos se prolongarão até interceptarem os segmentos **r1** e **r2** ou suas retas suporte (Figura 4.4 (b)). Calcular o ponto em **r1** ou **r2** em que ocorre a intersecção;

Corrigir todos os valores que ultrapassam os limites de **r1** e **r2.** O valor máximo é 1 e o valor mínimo é 0. Na Figura 4.4 (b) o valor de **r11o2** será corrigido para 1 e o valor de **r12o1** será corrigido para 0;

Construir os lados de um polígono da seguinte forma:
Lado Superior: (r22o1, 1) (r22o2, 1)
Lado Direito: (1, r12o2) (1, r12o1)
Lado Inferior: (r21o2, 0) (r21o1, 0);
Lado Esquerdo: (0, r1101) (0, r1102).

Se os dois vértices de um dos lados estiverem no mesmo limite (por exemplo, os dois vértices com valor 0), o lado é

descartado. Isso ocorre na Figura 4.4 (e) onde o lado direito foi eliminado e na Figura 4.4 (h) onde o lado inferior foi eliminado;

Caso a extensão de **o** não interceptar **r1** e **r2** (Figura 4.4 (a) e Figura 4.4 (d));

Então desenhar o polígono da seguinte forma: Lado Superior, Lado Direito, Lado Inferior, Lado Esquerdo. Se dois vértices em seqüência forem iguais, adicioná-lo apenas uma vez. Exemplos: Figura 4.4 (b) e 4.4 (e);

Senão (Caso a extensão de **o** interceptar **r1** e **r2** (Figura 4.4 (f)))

Desenhar o polígono da seguinte forma: Lado Superior, Lado Esquerdo, Lado Direito, Lado Inferior. Se dois vértices em seqüência forem iguais, adicioná-lo apenas uma vez. Exemplos: Figura 4.4 (g).

CÓDIGO 4.3 – Algoritmo para determinação da área de oclusão de um obstáculo

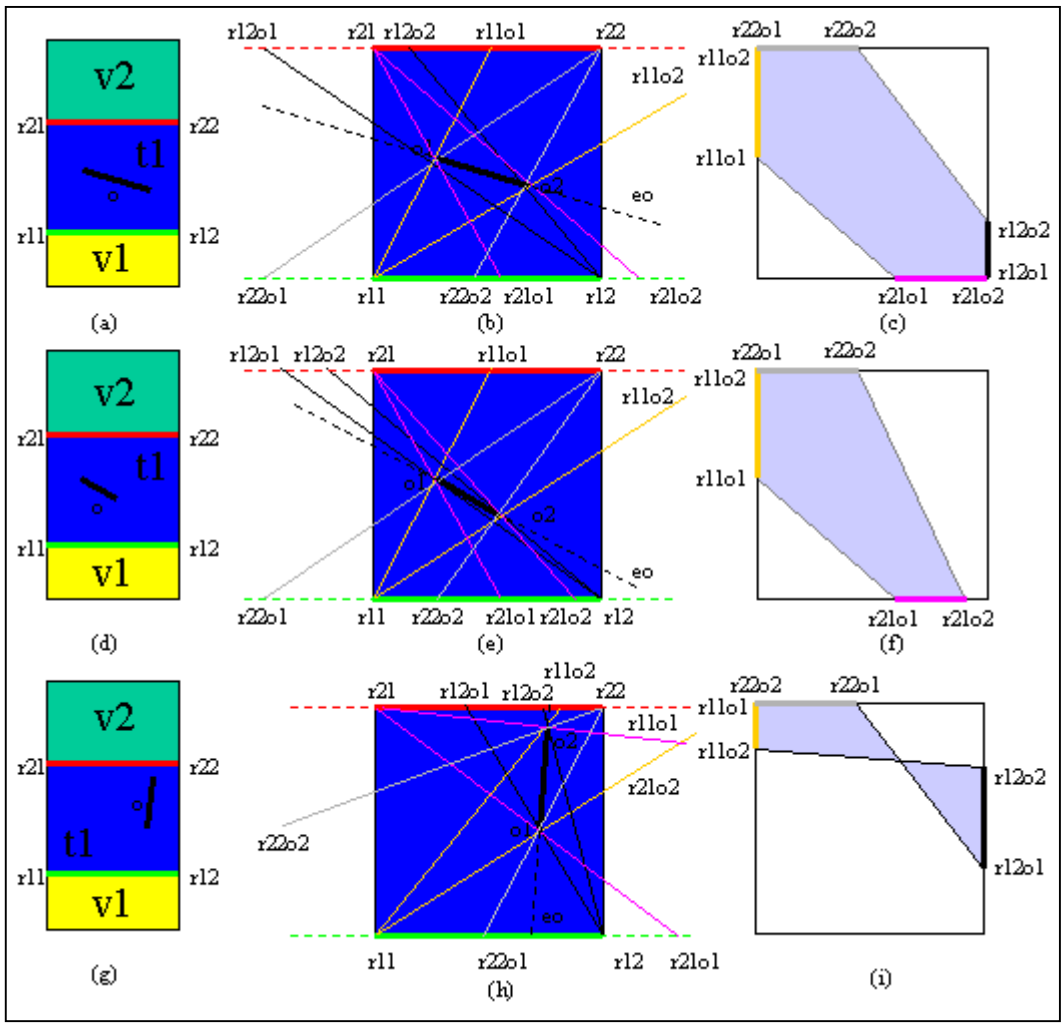


FIGURA 4.4 – Passos da construção do polígono que representa a oclusão de um obstáculo

Depois que o espaço bidimensional foi construído e que todos os pontos que representam raios que passam através de um obstáculo foram marcados, deve-se

descobrir se existe pelo menos um ponto que não foi marcado. Existem várias formas de se realizar essa operação:

Verificação Exaustiva

Consiste em se realizar uma leitura do espaço bidimensional e analisar cada pixel através de dois laços encadeados. O espaço bidimensional é inicialmente preenchido com pixels pretos e cada pixel que representar um raio onde não existe visibilidade será marcado de branco. No exemplo, a seguir, o espaço bidimensional é armazenada em um *buffer* chamado image. Se encontrarmos qualquer pixel preto então sabemos que existe visibilidade entre as duas regiões.

```
GlReadPixels(0,0,        RESOLUTION,        RESOLUTION,        GL RGB,
            GL_UNSIGNED_BYTE, &image);

for (i=0; i<RESOLUTION; i++)
      for (j=0; j<RESOLUTION; j++)
            if (image[i][j][0]==0)
                  return VISIBLE;
return NOT_VISIBLE;
```

CÓDIGO 4.4 – Checagem manual

MinMax

Consiste na utilização da função glMinMax. Essa função só está disponível em placas gráficas que suportem a extensão GL_ARB_imaging. A função glMinMax computa os valores máximos e mínimos dos pixels em um retângulo.

Como a imagem foi inicialmente preenchida com pixels pretos (valor RGB 0, 0, 0) e os pixels que representam raios onde não existe visibilidade são marcados de branco (valor RGB 255, 255, 255) pode-se utilizar essa função para descobrir qual o valor mínimo encontrado nos pixels do espaço bidimensional. Como estamos usando apenas preto e branco, se o valor mínimo de alguma das componentes for 255, isso significa que todos os pixels da imagem foram preenchidos de branco (ou seja, não existe visibilidade).

```
glEnable (GL_MINMAX);
glReadPixels (0,0, RESOLUTION, RESOLUTION, GL_RED, GL_UNSIGNED_BYTE, &image);
glDisable (GL_MINMAX);
glGetMinmax(GL_MINMAX, GL_TRUE, GL_RGB, GL_UNSIGNED_BYTE, values);
if (values[0]==255)
return NOT_VISIBLE;
else
return VISIBLE;
```

CÓDIGO 4.5 – Utilização da função MinMax

Visibility Test

Consiste na utilização da extensão GL_HP_visibility_test. Essa extensão permite a realização de um teste que indica se um objeto foi desenhado no *frame- buffer* ou não. Nesse algoritmo os pontos do espaço bidimensional não são inicialmente marcados de preto. Depois que os pixels que representam raios onde não existe visibilidade são marcados é ligado o *Z-Buffer* e desenhado um retângulo atrás desse espaço bidimensional. Como o retângulo tem as mesmas dimensões do espaço

bidimensional, se algum pixel desse quadrado for desenhado isso significa que um dos pixels não havia sido marcado anteriormente, ou seja, existe visibilidade.

```
glEnable(GL_DEPTH_TEST);
glEnable(GL_OCCLUSION_TEST_HP);
glBegin(GL_POLYGON);
      glVertex3f(0.0, 0.0, -1.0);
      glVertex3f(1.0, 0.0, -1.0);
      glVertex3f(1.0, 1.0, -1.0);
      glVertex3f(0.0, 1.0, -1.0);
glEnd();
glDisable(GL_OCCLUSION_TEST_HP);
glDisable(GL_DEPTH_TEST);
glGetBooleanv(GL_OCCLUSION_TEST_RESULT_HP,&occlusionResult);
glClear (GL_DEPTH_BUFFER_BIT);
return (occlusionResult);
```

CÓDIGO 4.6 – Utilização da função Visibility Test

Os dois fatores que regem a utilização de cada um dos métodos são a disponibilidade de hardware gráfico suportando estas operações e a performance. O método de verificação exaustiva pode ser utilizado em qualquer hardware gráfico que suporte OpenGL enquanto que os outros dois necessitam que suas extensões estejam implementadas.

A performance de cada um dos métodos varia de acordo com a forma como foram implementados no hardware gráfico. O método MinMax havia sido escolhido antes da etapa de implementação por ser bastante simples e por ter sido sugerido por [KOL 2001]. Contudo sua performance no hardware gráfico disponível foi muito aquém da esperada, sendo bem mais lento do que o método de verificação manual como pode ser observado na Tabela 4.1[9]. O método *visibility test* apresentou ganhos razoáveis em relação ao da verificação manual no hardware testado. Como a performance dos três métodos é dependente do hardware gráfico que está sendo utilizado, sugere-se testar os três métodos para se determinar qual deles possui a melhor performance antes de se realizar o cálculo de visibilidade de uma cena.

TABELA 4.1 – Tempo de processamento (em segundos) para o cálculo de visibilidade de regiões

	Verificação Manual	**MinMax**	**Visibility Test**
Região 1	6,844s	64,781s	5,813s
Região 2	8,812s	94,781s	7,265s
Região 3	5,438s	94,719s	3,594s
Região 4	4,797s	84,406s	3,141s
Região 5	4,671s	83,766s	3,015s

Já que a visibilidade entre regiões é uma propriedade simétrica, apenas metade das visibilidades precisa ser calculada. O laço que realiza o cálculo da visibilidade entre as regiões foi implementado de tal forma que apenas a matriz triangular superior é calculada e armazenada na memória. Quando é necessário o cálculo de visibilidade de algum elemento da matriz triangular inferior ele é simplesmente copiado da memória (uma vez que ele já foi calculado anteriormente para a matriz triangular superior) como podemos ver na Tabela 4.2 em uma cena com cinco regiões.

[9] Testes realizados em uma máquina com processador Pentium 4 2GHZ com 640 MB RAM, placa de vídeo NVidia QUadro4 900 XGL e sistema operacional Windows XP.

TABELA 4.2 – Visibilidades calculadas ou copiadas da memória

Região	1	2	3	4	5
1	X	Calculada	Calculada	Calculada	Calculada
2	Idem (1,2)	X	Calculada	Calculada	Calculada
3	Idem (1,3)	Idem (2,3)	X	Calculada	Calculada
4	Idem (1,4)	Idem (2,4)	Idem (3,4)	X	Calculada
5	Idem (1,5)	Idem (2,5)	Idem (3,5)	Idem (4,5)	X

Depois que os PVS de todas as regiões foram calculados, eles são armazenados em disco. Assim como o aplicativo QVIS3, o FastPVS recebe um arquivo BSP como entrada e armazena os PVS calculados no mesmo arquivo.

4.5 SVS

Uma vez que a visibilidade básica foi calculada (através do aplicativo QVIS3 ou FastPVS) pode-se utilizar o aplicativo SVS para a realização de um particionamento dessa informação em diferentes grupos. Esse aplicativo implementa os algoritmos descritos no Capítulo 3, dividindo a visibilidade utilizando ângulos de visibilidade ou distância entre as regiões.

Esse aplicativo recebe como entrada um arquivo BSP que já possui os PVS calculados. Também é informado o modo como a visibilidade deve ser particionada. Depois de realizado o particionamento a nova visibilidade é armazenada no mesmo arquivo.

Para se particionar a visibilidade em ângulos estáticos utiliza-se a opção `–anglelist` seguida de uma lista dos ângulos a serem utilizados (em graus). Por exemplo:

```
SVS City.BSP –anglelist 0 90 180 270
SVS City.BSP –anglelist 0 120 240
```

Esses parâmetros irão particionar a visibilidade presente no arquivo City.BSP em ângulos estáticos. No primeiro exemplo serão criados quatro subconjuntos de visibilidade [(0^o, 90^o), (90^o, 180^o), (180^o, 270^o), (270^o, 0^o)]. No segundo exemplo, os três subconjuntos de visibilidade serão [(0^o, 120^o), (120^o, 240^o), (240^o, 0^o)].

Para se calcular a visibilidade usando o algoritmo adaptativo para a escolha dos particionadores utiliza-se a opção `–autosplit` seguida da profundidade até a qual se deseja executar o algoritmo e o número de candidatos a serem testados em cada passo. A profundidade determina em quantos ângulos o campo de visão será particionado. Em cada nível do algoritmo o campo de visão é particionado em dois e cada um dos campos de visão resultantes é enviado ao próximo passo. O número de candidatos indica quantas partições serão testadas a cada passo. Quanto maior o número de candidatos maiores serão as chances de se encontrar um particionador próximo do ideal e mais demorada será a execução do algoritmo. Por exemplo:

```
SVS City.BSP –autosplit 2 5
SVS City.BSP –autosplit 2 10
```

Ambos os exemplos irão particionar a visibilidade contida no arquivo City.BSP em quatro subconjuntos. Mas, no primeiro exemplo, apenas cinco partições serão testadas a cada passo enquanto que serão testadas dez no segundo.

Para a realização do particionamento da visibilidade em distâncias utiliza-se a opção `–distancelist` seguida de uma lista das distâncias a serem utilizadas. Por exemplo:

```
SVS City.BSP –distancelist 30 70 150 5000
```

Esses parâmetros irão particionar a visibilidade contida no arquivo City.BSP em quatro subconjuntos: regiões com distância inferior a 30, 70, 150 e 5000 unidades respectivamente. Pode-se também combinar particionamentos de ângulo e de distância, mas não dois particionamentos diferentes de ângulos. Por exemplo:

```
SVS City.BSP –anglelist 0 120 240 –distancelist 30 70 150 5000
SVS City.BSP –autosplit 2 5 –distancelist 30 70 150 5000
```

Os dois exemplos irão particionar a visibilidade contida no arquivo City.BSP em dois subconjuntos: um baseado no particionamento do campo de visão e outro baseado no particionamento da distância. A única diferença é que, no segundo exemplo, os campos de visão particionadores serão escolhidos através do algoritmo adaptativo.

4.6 BSPViewer

Esse aplicativo foi desenvolvido para permitir a visualização e análise dos diferentes tipos de estruturas utilizados: *BSP-Tree*, PVS e SVS. Ele foi de grande importância para o desenvolvimento da aplicação SVS pois permitiu que diversos algoritmos fossem testados (como, por exemplo, o algoritmo de geração de *bounding volumes*). Durante a implementação da aplicação FastPVS, o ambiente BSPViewer foi utilizado para verificar a correção dos algoritmos, como o algoritmo de geração do *shaft* (Figura 4.3).

Essa aplicação foi projetada para possuir dois módulos: um de controle e um gráfico. O módulo de controle é responsável por todo a funcionamento da aplicação e inclui funções como abertura de arquivos e controle de câmera. O módulo gráfico é responsável pela exibição das imagens na tela e atende a requisições do módulo de controle como, por exemplo, “desenhe uma *BSP-Tree*” ou “desenhe um volume” como pode ser visto na Figura 4.5.

Dessa forma o módulo gráfico pode ser facilmente alterado para utilizar diferentes bibliotecas gráficas. Basta que o módulo continue oferecendo as mesma funções ao módulo de controle. A biblioteca gráfica utilizada para se desenvolver o módulo gráfico foi a OpenGL e o ambiente de desenvolvimento para ambos os módulos foi o Visual C++.

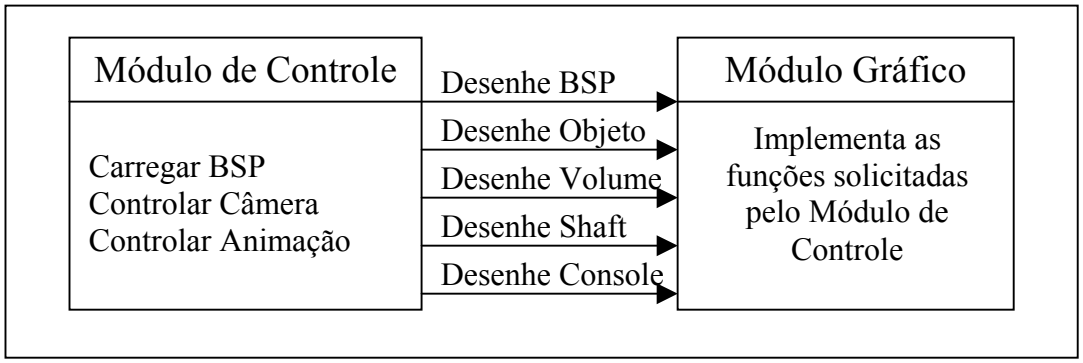


FIGURA 4.5 – Relacionamento entre os módulos de controle e gráfico

As funções básicas oferecidas pela aplicação são as seguintes:

1. Carregar um arquivo *BSP-Tree*;
2. Escolher o algoritmo de visualização desejado;
3. Deslocar um objeto até um ponto desejado ou percorrer um caminho;
4. Observar as características do algoritmo de visualização para aquele determinado ponto ou caminho a partir de uma outra câmera.

Para escolher um determinado algoritmo de visualização, o arquivo BSP deve conter os dados necessários. Por exemplo: caso se deseje utilizar os PVS para visualização, o arquivo BSP carregado deve ter passado pelas aplicações QVIS3 ou FastPVS.

A Figura 4.6 ilustra a idéia de se observar as características de um algoritmo de visualização a partir de um determinado ponto. A imagem gerada utiliza os PVS da região onde a pirâmide se encontra e não da região onde a câmera se encontra. As faces visíveis estão sendo desenhadas com cores sólidas e as faces que não são visíveis a partir daquela região são representadas pelas áreas escuras.

Dessa maneira pode-se navegar pelo ambiente tridimensional enquanto a região de interesse (onde a pirâmide se encontra) permanece inalterada. Da mesma forma pode-se fixar a câmera em uma posição e se alterar a região de interesse movimentando-se a pirâmide. A movimentação é realizada através do teclado (teclas direcionais e teclas A e Z). A mudança de orientação é realizada através do mouse. O chaveamento entre movimentação da câmera ou movimentação da pirâmide é realizado pelo botão esquerdo do mouse.

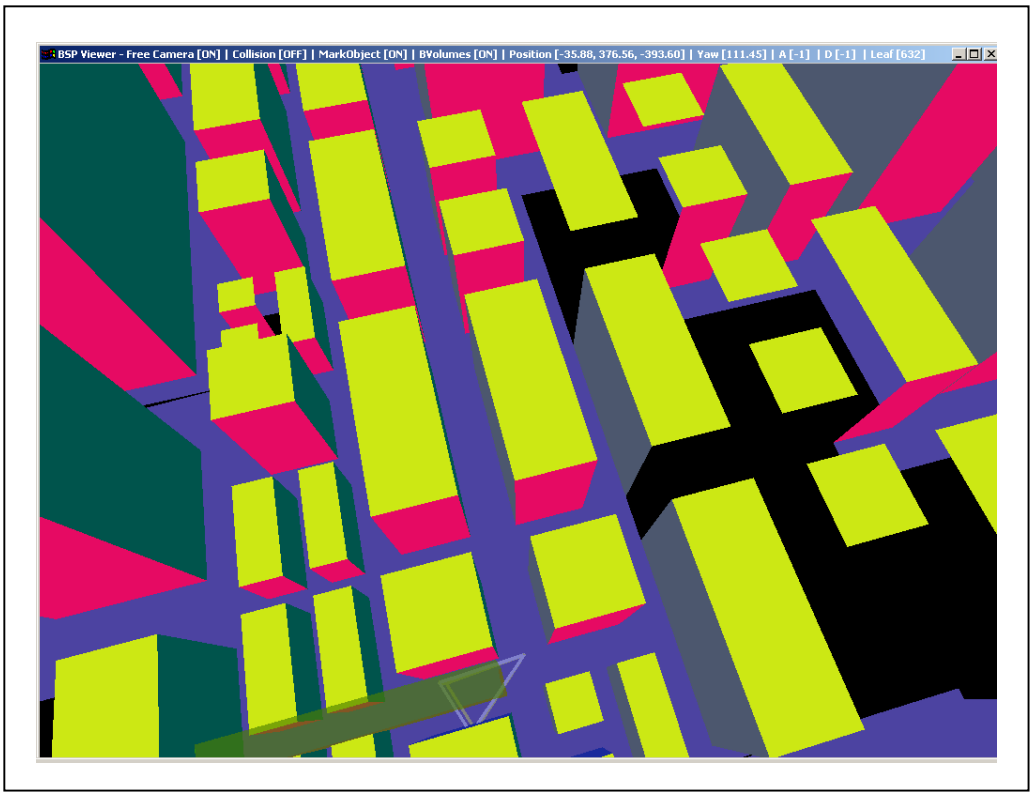


FIGURA 4.6 – PVS de uma região observado através de uma câmera

O BSPViewer oferece também uma série de outras funções que permitem:

- Criar um caminho a ser percorrido;
- Executar o caminho e verificação de quantas faces são visíveis a cada frame e ao longo do caminho;
- "Renderizar" as faces visíveis ou invisíveis como pontos, wireframe ou faces sólidas ou texturizadas;
- Iluminar a cena;
- Executar scripts (conjuntos de comandos);
- Visualizar o *bounding-volume* de cada região;
- Construir o *shaft* entre duas regiões;
- Verificar a visibilidade entre duas regiões utilizando o algoritmo Dual-Ray Space;
- Mudar rapidamente o modo de visualização (PVS, SVS-Angle ou SVS-Distance);
- Detectar a colisão entre a câmera ou pirâmide e os objetos da cena;
- Utilizar projeção ortográfica ou perspectiva;
- Informar a posição da pirâmide ou da câmera, bem como em qual região da BSP eles se encontram.

A comunicação entre a aplicação e o usuário é feita através de uma interface textual. Nela o usuário pode digitar comandos e a aplicação retorna informações sobre os resultados das operações requisitadas. A interface textual pode ser acessada através da tecla de aspas. Na Figura 4.7 podemos observar a execução de uma série de comandos especificados pelo usuário através dessa interface. Para se retornar ao modo de visualização deve-se pressionar a tecla de aspas novamente.

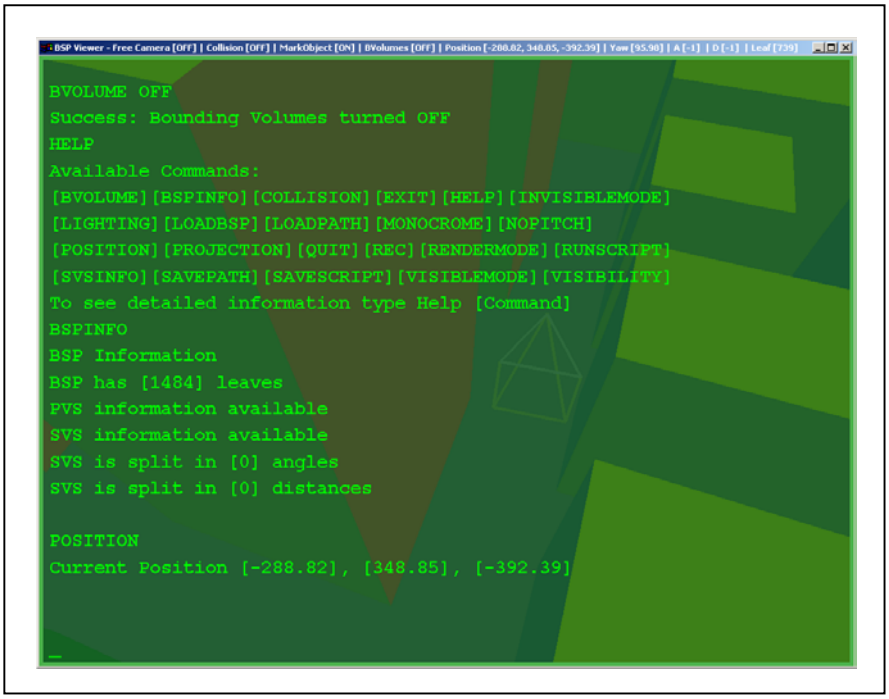


FIGURA 4.7 – Interface textual da aplicação BSPViewer

A seguir serão detalhadas cada uma das operações disponíveis na aplicação BSPViewer:

BSPInfo – Informa as características do arquivo BSP armazenado na memória. Informa número de regiões da *BSP-Tree* e também quais tipos de visibilidade pré-calculada estão disponíveis: PVS, SVS-Angle ou SVS-Distance.

BVolume [On/Off] – Define se o *bounding volume* da região na qual a pirâmide se encontra será "renderizado" ou não. Default Off.

Collision [On/Off] – Realiza ou não o cálculo de colisão entre objeto que estiver sendo movimentado (câmera ou pirâmide) e os objetos presentes no cenário. Se ocorrer uma colisão durante a movimentação, o objeto que irá retornar à posição anterior à colisão. Default Off.

End – Marca a região na qual a pirâmide se encontra como a região de destino a ser utilizada no teste de visibilidade.

Exit – Encerra a execução da aplicação.

FastVis – Realiza o cálculo de visibilidade entre as regiões definidas pelos comandos Start e End utilizando o algoritmo Dual-Ray Space. Após o cálculo é informado se as duas regiões são visíveis entre si ou não.

Help – Mostra uma listagem com todos os comandos disponíveis ao usuário. Informações detalhadas sobre cada comando podem ser obtidas através de **Help [Command]**.

InvisibleMode [Point/Wire/Solid/Texture] – Determina como as faces das regiões não visíveis serão "renderizadas". Como pontos, wireframe, faces sólidas ou faces texturizadas. Essa opção produz resultados apenas se a opção "Rendermode" houver sido utilizada como o parâmeto All. Default Wire.

LeafInfo – Retorna o número da região em que a pirâmide se encontra.

Lighting [On/Off] – Adiciona ou remove iluminação à cena. A luz criada é do tipo omni e sua posição acompanha a câmera de observação. Default Off.

LoadBSP [BSPName] – Carrega um arquivo do tipo BSP e de nome BSPName para a memória. O arquivo deve estar localizado no diretório BSP. Se já existir uma BSP armazenada na memória realiza o desalocamento de memória antes de carregar o novo arquivo.

LoadPath [Pathname] – Carrega o caminho armazenado no arquivo Pathname localizado no diretório Paths para a memória.

Monocrome [On/Off] – Define se o rendering da cena utilizará apenas cores monocromáticas ou não. Default Off.

NoPitch [On/Off] – Determina se a pirâmide poderá ter sua orientação relativa à elevação alterada ou não. A ativação dessa opção facilita a construção de caminhos bidimensionais. Default Off.

Play – Executa um caminho do primeiro ao último frame. Isso significa que a movimentação da pirâmide irá obedecer aos frames armazenados no caminho e não poderá ser movimentada pelo usuário até que o caminho se encerre. Para facilitar a observação do caminho, cópias da pirâmide são deixadas em intervalos regulares enquanto o caminho é percorrido.

Position – Retorna a posição em que a pirâmide se encontra.

Projection [Ortho/Perspective] – Altera a projeção para ortográfica ou perspectiva. Defaul Perspective.

Quit – Encerra a execução da aplicação

Rec [On/Off] – Inicia ou para a construção de um caminho. Cada movimento realizado pela pirâmide é armazenado como um novo frame do caminho. Movimentações da câmera de observação não são armazenadas no caminho. Default Off.

RenderFastVis – Realiza o rendering apenas das faces que colidiram com o *shaft* durante o cálculo de visibilidade. As intersecções entre essas faces e o *shaft* são "renderizadas" como segmentos de reta. Esse comando só pode ser executado após a execução do comando FastVis.

Rendermode [All/Visible] – Determina se a cada frame serão "renderizadas" todas as regiões da cena ou apenas aquelas que são visíveis a partir da região na qual a pirâmide se encontra. Default Visible.

RenderShaft [On/Off] – "Renderiza" o *shaft* entre as regiões marcadas pelos comandos Start e End. Default Off.

RunScript [ScriptName] – Executa seqüencialmente todos os comandos armazenados no arquivo ScriptName localizado no diretório Scripts.

SaveScript [NumberOfLines] [ScriptName] – Grava os últimos NumberOfLines comandos digitados em um arquivo de nome ScriptName no diretório Scripts.

SavePath [Pathname] – Armazena o caminho existente na memória no arquivo Pathname localizado no diretório Paths.

Start – Marca a região na qual a pirâmide se encontra como a região de origem a ser utilizada no teste de visibilidade.

SVSInfo – Retorna informações sobre os tipos de SVS contidos no arquivo BSP que está na memória. O opção Visibility deve estar setada para SVSDistance ou SVSAngle para que essa opção possa ser utilizada.

Test – Realiza um teste de visibilidade ao longo do caminho armazenado na memória. Executa o caminho da mesma forma descrita na função Play, mas a cada frame é feita uma análise de quantas regiões da cena são visíveis no momento e quantas regiões da cena foram vistas até esse frame (incluindo as regiões vistas em todos os frames anteriores). Esses valores são armazenados em um arquivo texto no diretório Results.

Visibility [PVS/SVSAngle/SVSDistance/AngleNumber] [Number] – Realiza o rendering da cena utilizando um dos tipos de visibilidade. A opção AngleNumber deve ser seguida de um número que indica qual dos ângulos deve ser considerado.

VisibleMode [Point/Wire/Solid/Texture] – Determina como as faces das regiões visíveis serão “renderizadas”. Como pontos, wireframe, faces sólidas ou faces texturizadas. Default Solid.

Na Figura 4.8a encontramos uma cidade sendo observada com todas as opções padrão. A Figura 4.8b possui as opções BVolume On, VisibleMode Texture, RenderMode All e InvisibleMode Wire diferentes das opções padrão. A Figura 4.8c possui as opções BVolume On, VisibleMode Wire, RenderMode All, InvisibleMode Solid e Monocrome On diferentes das opções padrão. A Figura 4.8d possui as opções BVolume On, VisibleMode Solid, RenderMode All e InvisibleMode Texture diferentes das opções padrão.

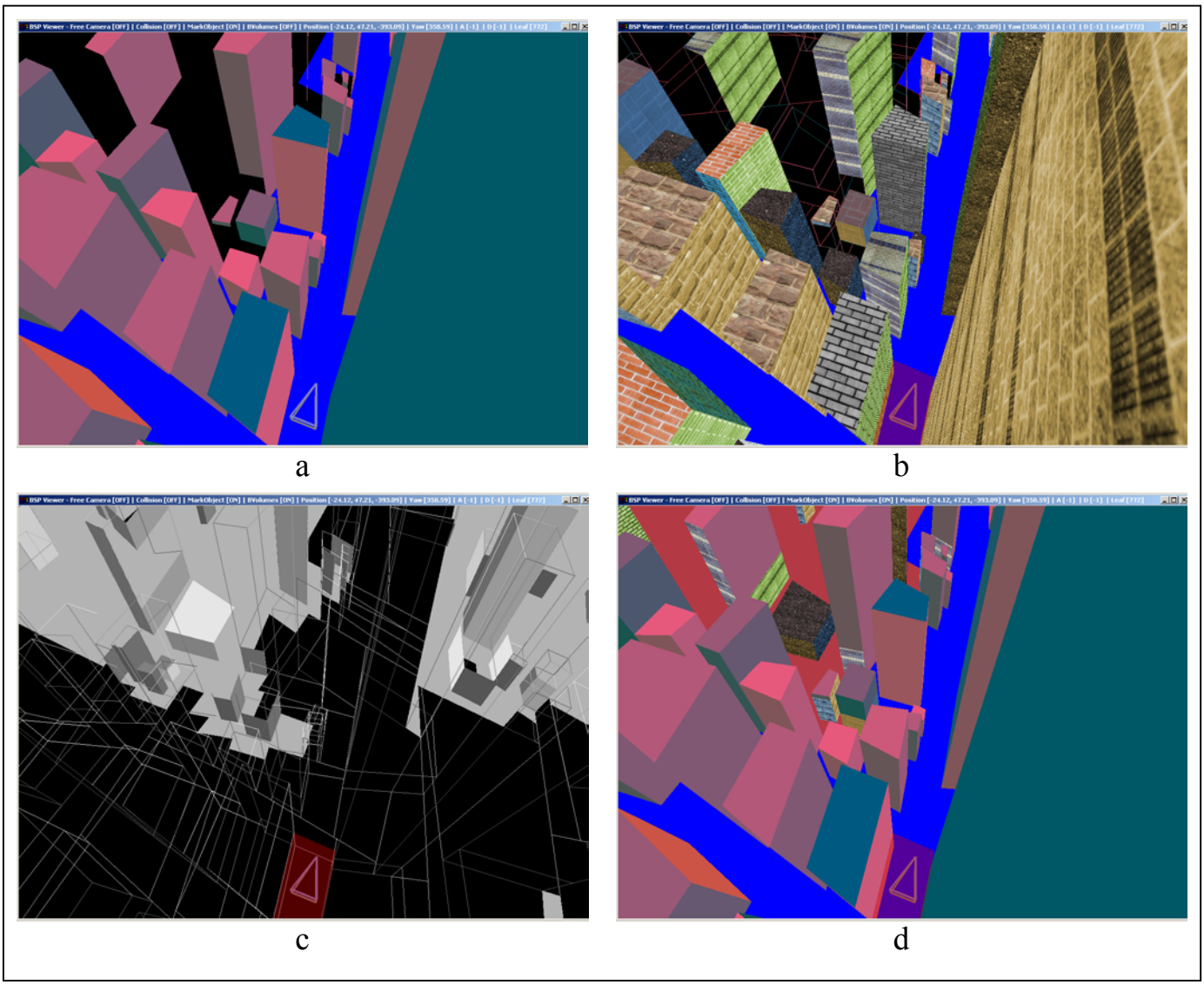


FIGURA 4.8 – Mesma cena observada com diferentes opções de rendering: (a) – Todas opções padrão (b) – *Bounding-volume* visível, faces visíveis texturizadas, faces não visíveis em aramado (c) – *Bounding-volume* visível, faces visíveis em aramado, faces não visíveis sólidas e modo monocromático (d) – *Bounding-volume* visível, faces visíveis sólidas, faces não visíveis texturizadas

Na Figura 4.9a temos uma visão superior da cidade. A única opção não padrão é BVolume On. Na Figura 4.9b temos a mesma visão com a opção Projection Ortho ligada. Na Figura 4.9c podemos observar um caminho sendo percorrido. Na Figura 4.9d temos o mesmo caminho visto com as opções Projection Ortho, RenderMode All e InvisibleMode Wire ligadas.

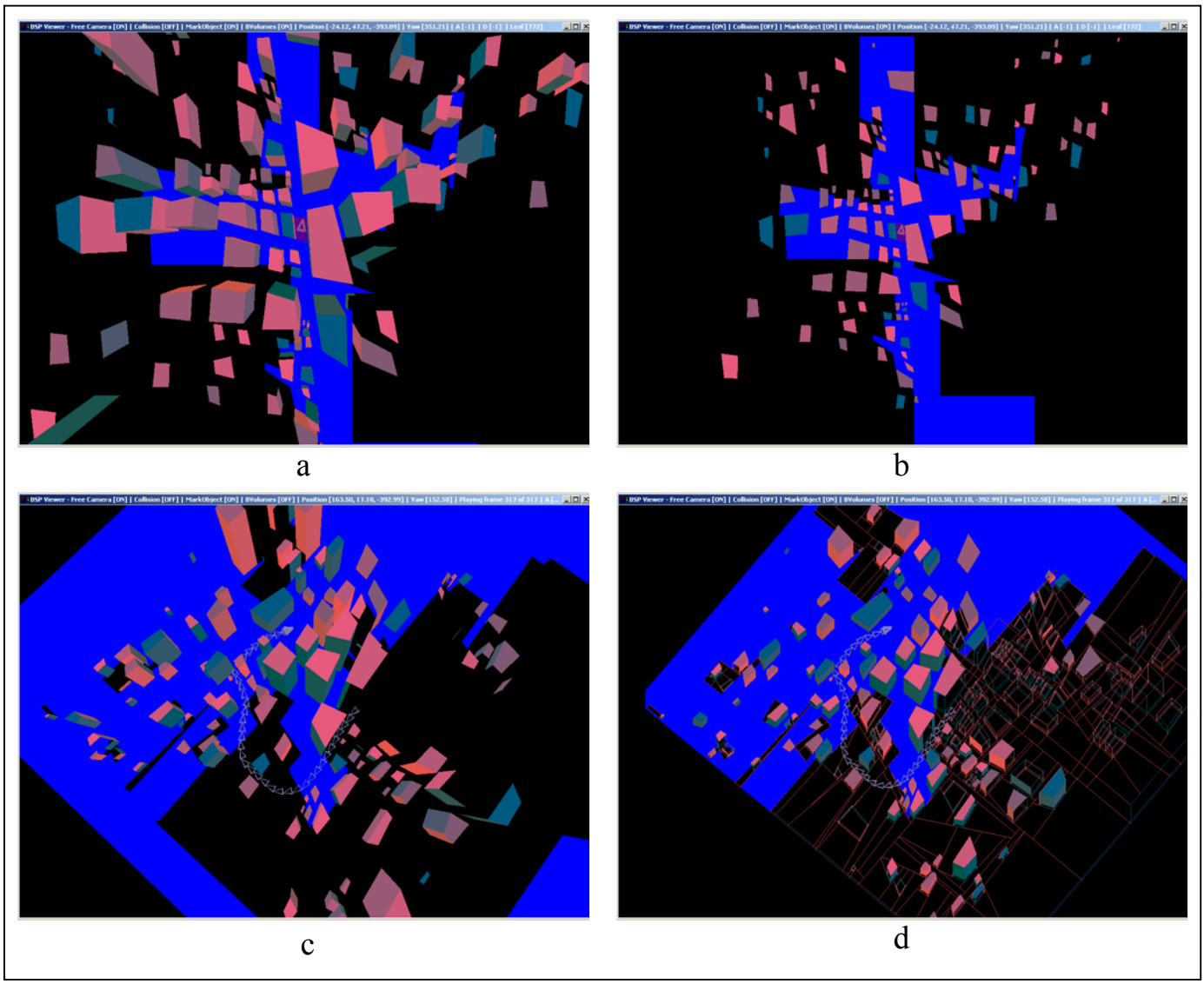


FIGURA 4.9 – Utilização da projeção ortográfica: (a) – *Bounding-volume* visível e projeção perspectiva (b) – *Bounding-volume* visível e projeção ortográfica (c) – Caminho sendo percorrido com projeção perspectiva (d) – Caminho sendo percorrido com projeção ortográfica

Na Figura 4.10 podemos observar a realização de um teste de visibilidade entre duas regiões através do algoritmo “Dual-Ray Space”. A Figura 4.10a mostra as duas regiões para as quais se deseja descobrir se são visíveis entre si ou não (marcadas por um volume amarelo e outro verde, respectivamente). A Figura 4.10b mostra a construção do *shaft* entre as duas regiões. A Figura 4.10c mostra a execução do comando fastvis que, baseado no *shaft*, verifica a visibilidade entre as duas regiões. A Figura 4.10d mostra a utilização do comando RenderFastVis On, onde as únicas faces desenhadas são aquelas que interceptam o *shaft* e a intersecção é desenhada como segmentos de reta.

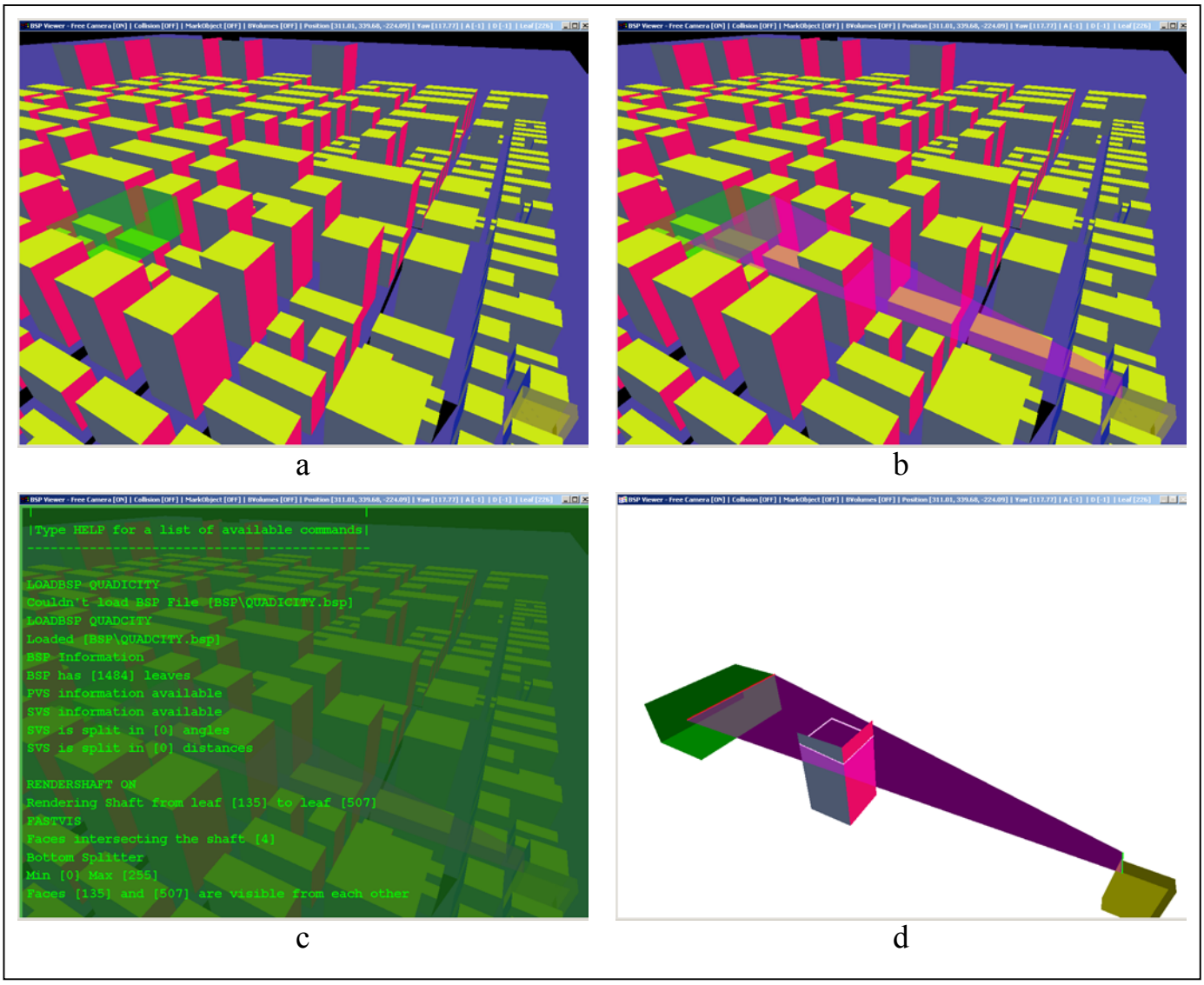


FIGURA 4.10 – Verificação de visibilidade entre duas regiões: (a) – Seleção de duas regiões (b) – Construção do *shaft* (c) – Execução do comando fastvis (d) – Rendering das faces que interceptam o *shaft* e das intersecções entre a face e o *shaft*

5 Resultados

Neste capítulo são apresentados os resultados obtidos. Inicialmente foi criada uma cena de testes onde os tempos gastos nas diversas aplicações descritas no Capítulo 4 foram medidos e analisados. A seguir são executados testes para se analisar a qualidade da informação de visibilidade no PVS gerado a partir da aplicação QBSP3, no PVS gerado a partir da aplicação QVIS3 e em diferentes tipos de SVS. Finalmente esses diferentes conjuntos de visibilidade são testados segundo caminhos percorridos na cidade de testes.

5.1 Apresentação dos Resultados

Para a geração da cena de testes foi utilizado o programa gerador de cidades – CityGenerator. A cidade foi criada com 75 prédios para que o tempo gasto com o processamento da visibilidade não fosse demasiado grande, inviabilizando a execução dos diferentes testes e ajustes que eram necessários nas diversas aplicações empregadas, uma vez que a visibilidade devia ser recalculada após cada ajuste.

Todos os resultados apresentados foram obtidos em um Pentium 4 de 2 GHZ com 640 MB de memória RAM e uma placa de vídeo NVIDIA Quadro4 900 XGL. O sistema operacional da máquina é o Windows XP.

Após a criação da cidade e armazenamento em um arquivo MAP, foi utilizado o aplicativo QBSP3 para particionamento da cena em regiões de uma *BSP-Tree*. Um problema encontrado foi que as regiões geradas eram muitas vezes extensas. Isso causava dois tipos de problemas:

1. Regiões muito extensas são visíveis por um grande número de outras células, aumentando o tamanho total da visibilidade da cena
2. Um PVS calculado a partir de uma região muito grande será bem maior do que um calculado a partir de uma região menor. A existência desse tipo de região não é aconselhável um vez que ela introduz uma grande variabilidade no tamanho da informação de visibilidade, piorando os resultados obtidos tanto em testes globais (realizados em todas as regiões da cena) como em caminhos percorridos através da cena (que consideram apenas algumas das regiões da cena).

Como foi explicado no Seção 4.2, foi realizada uma alteração no aplicativo QBSP3 a fim de particionar as regiões que eram muito extensas, desta forma homogeneizando o tamanho das regiões da cena. Na Figura 5.1 podemos observar uma mesma região criada a partir do particionamento da cena usando a aplicação QBSP3 original e a versão modificada.

A subdivisão das regiões grandes da cena aumenta o tempo gasto na etapa de criação da estrutura *BSP-Tree*. A etapa de geração de visibilidade também se torna uma pouco mais lenta, uma vez que o número de regiões da cena aumenta. Contudo a melhoria da qualidade na visibilidade da cena e a homogeneização das regiões da cena compensam essas perdas. Além disso, o tempo gasto na etapa de geração da *BSP-Tree* é desprezível em relação as demais etapas. Na Tabela 5.1 pode-se observar a diferença

entre número de folhas, tempo de processamento e tamanho dos arquivos gerados no processamento de uma cena com 75 prédios.

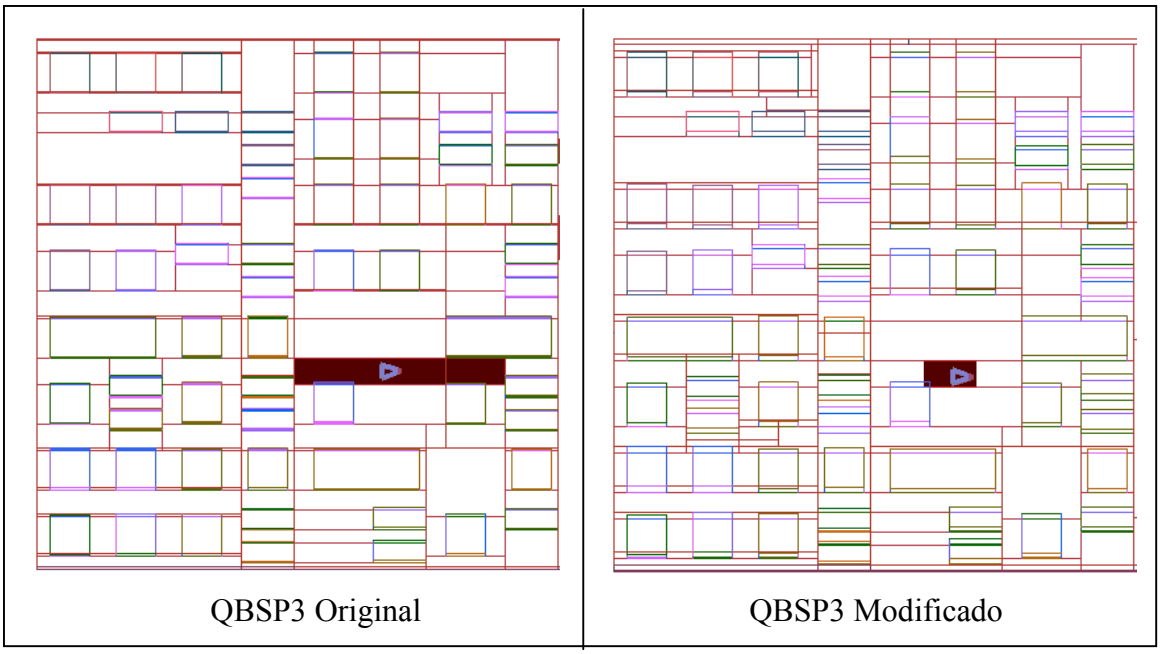


FIGURA 5.1 – Subdivisão das regiões muito extensas

TABELA 5.1 – Comparação entre QBSP3 original e modificado

	Número de Folhas	**Tempo de Processamento**	**Tamanho do arquivo**
QBSP3 Original	453	1s	107.656 bytes
QBSP3 Modificado	629	5s	647.476 bytes

Uma solução utilizada por outros trabalhos [KOL 2000, WON 2001, KOL 2001] foi o uso de células de visualização que não seguem a subdivisão espacial da cena. Entretanto essas abordagens costumam limitar a altura das células de visualização. Optou-se por explorar a propriedade de que as células de visualização podem ter alturas distintas e o usuário pode se deslocar livremente em todas as dimensões. A limitação de [KOL 2001] permitia a aceleração do cálculo de visibilidade, pois menos regiões seriam consideradas uma vez que elas não eram subdividas pela altura. Ressalta-se que a subdivisão das regiões pela altura não traz problemas para o cálculo da visibilidade através do algoritmo *Dual Ray* pois a única limitação existente é que os obstáculos sejam 2.5D.

A partir desta etapa foi utilizada apenas a cena com regiões homogêneas. Uma vez realizada a subdivisão da cena, foi feito o cálculo da visibilidade usando-se as aplicações QVIS3 e FastVis. Para cenas pequenas, como a testada, pode-se empregar o QVIS3. Entretanto, em cenas maiores, a utilização de portais pelo QVIS3 torna o algoritmo de visibilidade extremamente ineficiente. O desenvolvimento da aplicação FastVis permitiu que a visibilidade de cenas maiores fosse processada, e possibilitou ganhos de desempenho mesmo em cenas menores.

Na Tabela 5.2 pode-se comparar os tempos de processamento das duas aplicações em cidades de tamanhos diferentes. Pode-se observar também, o número médio de regiões visíveis a partir de cada célula.

TABELA 5.2 – Tempos de execução dos aplicativos QVIS3 e FastVis em cidades de diferentes tamanhos e visibilidade média das regiões.

	75 Prédios	**250 Prédios**	**Número Médio de Regiões Visíveis (Cidade de 75 Prédios com 468 regiões)**
QVIS3	2m 58s	3h 15m 20s	278,634
FastVis	1m 41s	15m 43s	287,155

Pode-se notar que o algoritmo implementado pela aplicação QVIS3 é um pouco menos conservativo (0,8% menos conservativo na cena com 75 prédios) do que o algoritmo *Dual Ray*. Por outro lado, os tempos de processamento inviabilizam a sua utilização em cenas maiores. O *Dual Ray* oferece uma solução para este problema, reduzindo enormemente o tempo de processamento e causando um aumento muito pequeno na visibilidade do PVS.

A partir desta etapa foram utilizadas apenas as cenas geradas utilizando-se a aplicação FastVis, o que permitiu que cenas maiores fossem testadas. Uma vez calculada a visibilidade entre as regiões, pode-se utilizar a aplicação SVS para realizar o particionamento da visibilidade em subconjuntos. Foram testados três particionamentos utilizando ângulo e um utilizando distância. A Tabela 5.3 mostra os tempos de processamento necessário para se gerar cada um desses subconjuntos para uma cena com 75 prédios.

TABELA 5.3 – Tempos de execução do aplicativo SVS com diferentes particionamentos

Particionamento	**Cena com 75 Prédios**
3 ângulos estáticos	2s
4 ângulos estáticos	1s
4 ângulos usando auto particionamento	20s
4 distâncias	1s

A opção de auto-particionamento é mais lenta que as demais porque a cada passo o algoritmo deve testar um número (especificado pelo usuário) de candidatos a ângulo particionador. No exemplo da Tabela 5.3 foram testados 10 particionadores a cada passo.

O próximo passo foi a execução de cenas com visibilidade particionada de diferentes maneiras no aplicativo BSPViewer. Esse aplicativo permite também que diferentes testes sejam realizados, tanto para todas as regiões quanto para caminhos executados dentro da cena. Com isso foi possível comparar o desempenho de diferentes tipos de SVS entre si e com o PVS.

Nas exemplos a seguir, é mostrada a visibilidade segundo diferentes partições em uma cena com 75 prédios. A célula de visualização corresponde ao volume amarelo, e a posição do usuário é representada pela pirâmide no centro da célula de visualização. Sua orientação é indicada pelo topo da pirâmide. Os polígonos das regiões visíveis são

"renderizados" em cor sólida enquanto que os polígonos das regiões não visíveis estão em aramado.

A Figura 5.2 mostra a visibilidade (PVS) de uma célula central. A visibilidade neste caso independe do ponto de vista do observador. Existem 266 regiões visíveis de um total de 468.

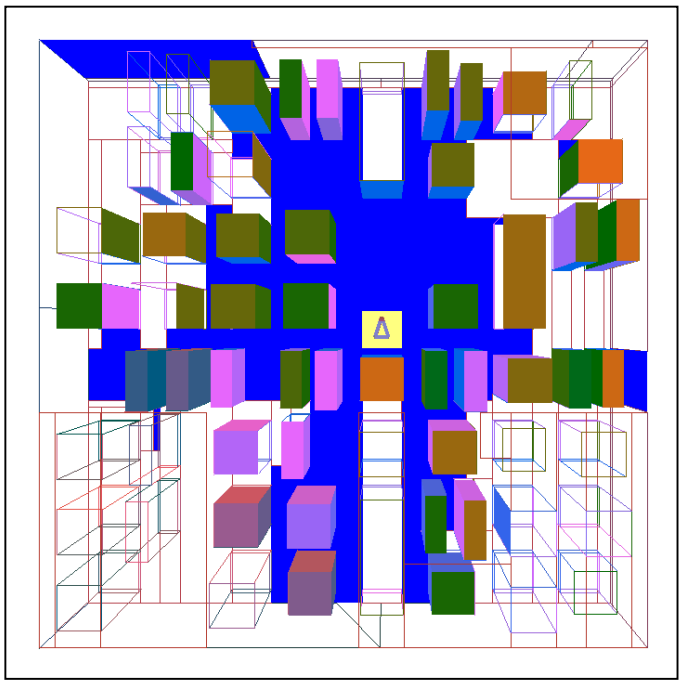


FIGURA 5.2 – PVS de uma região

Na Figura 5.3, a visibilidade foi particionada em três conjuntos: (a) 0^o – 120^o; (b) 120^o – 240^o; (c) 240^o – 360^o, escolhidos pelo usuário através do comando `SVS City.BSP -anglelist 0 120 240`. Nestes casos, o conjunto de visibilidade é selecionado segundo a orientação do observador. Convém notar que, para facilitar a navegação, o ápice da pirâmide foi utlizado para indicar a porção do espaço em frente ao observador.

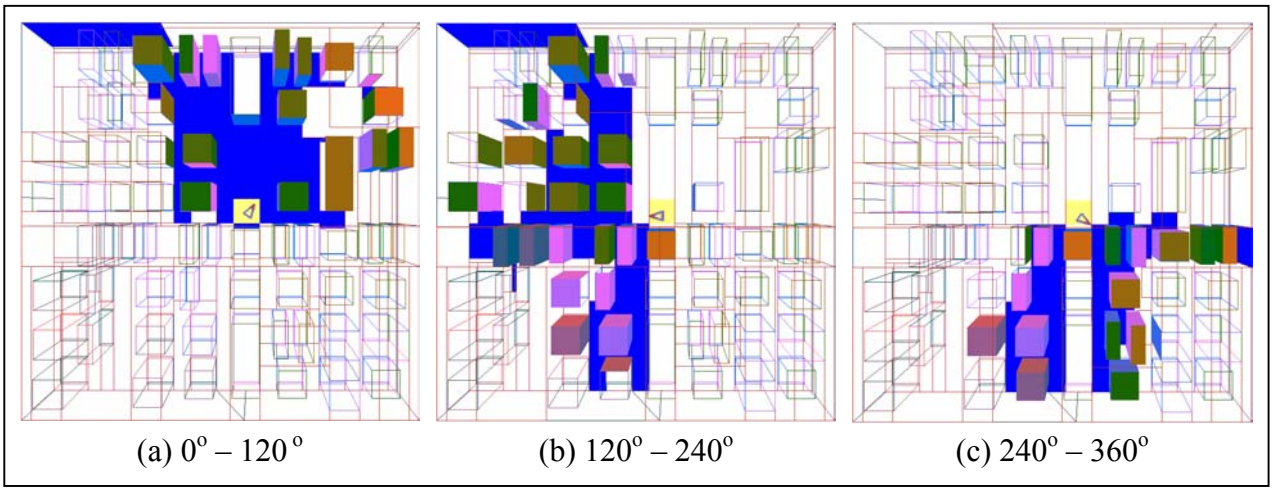


FIGURA 5.3 – SVS particionado segundo 3 ângulos escolhidos pelo usuário

Na Figura 5.4, a visibilidade foi particionada em quatro conjuntos: (a) 0^o – 90^o; (b) 90^o – 180^o; (c) 180^o – 270^o; (d) 270^o – 360^o, escolhidos pelo usuário através do comando `SVS City.BSP -anglelist 0 90 180 270`. Novamente o conjunto de visibilidade é selecionado segundo a orientação do observador.

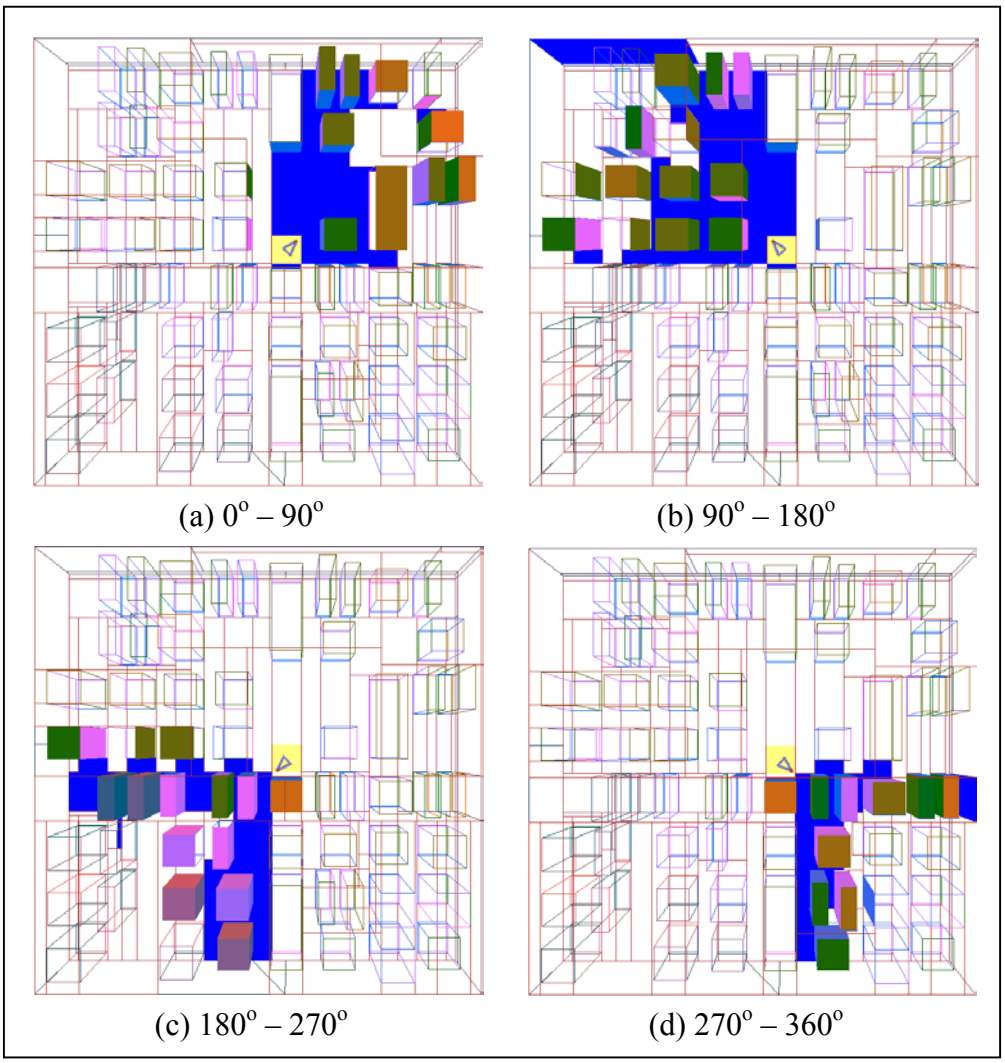


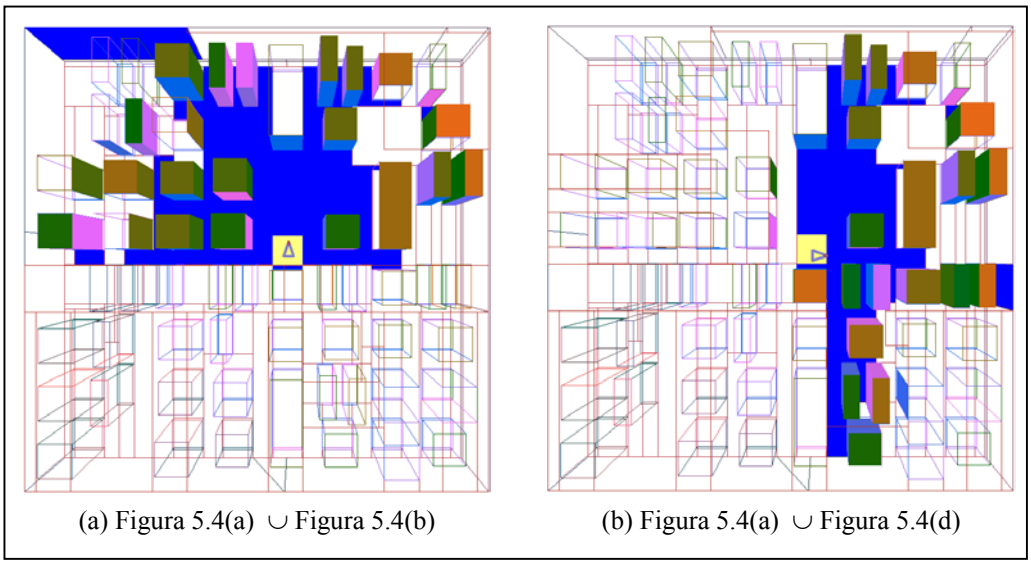
FIGURA 5.4 – SVS gerado a partir do particionamento da visibilidade segundo 4 ângulos escolhidos pelo usuário

O número de regiões visíveis em cada partição das Figuras 5.3 e 5.4 pode ser observado na Tabela 5.4. O número total de regiões da cena é 468 e o número de regiões visíveis no PVS a partir da região assinalada em amarelo é 266.

Quando o cone de visão do observador intersecta mais de uma partição do SVS, a visibilidade é gerada através da união das duas partições, que pode ser realizada rapidamente através de uma operação OR caso a visibilidade esteja armazenada como uma matriz de bits. A Figura 5.5 mostra a união das partições da Figura 5.4 (a) e (b).

TABELA 5.4 – Número de regiões visíveis para cada partição SVS

	Ângulo	**Número de Regiões Visíveis**
PVS		266
SVS 3 Ângulos	0^o – 120^o	136
	120^o – 240^o	123
	240^o – 360^o	103
SVS 4 Ângulos	0^o – 90^o	99
	90^o – 180^o	98
	180^o – 270^o	75
	270^o – 360^o	71



(a) Figura 5.4(a) ∪ Figura 5.4(b) (b) Figura 5.4(a) ∪ Figura 5.4(d)

FIGURA 5.5 – Visibilidade gerada a partir da união de diferentes partições

Na Figura 5.6 foi escolhida uma célula de visualização perto da borda superior direita de cena (a) com o objetivo de mostrar a existência de partições mal balanceadas, como pode ser visto em (b) onde a partição mostrada (0^o – 90^o) armazena poucas regiões visíveis. Essa partição foi determinada pelo usuário e divide todos os PVS da cena em quatro conjuntos. Para se resolver este problema foi utilizado um particionamento automático da visibilidade através do comando `SVS City.BSP -autoplit 2 10`. Através desse comando o algoritmo irá testar 10 particionadores a cada passo escolhendo aquele que melhor balancear o particionamento da visibilidade. Assim as partições serão diferentes para diferentes células de visibilidade. Na célula em questão o particionamento é (c) 0^o – 172^o; (d) 172^o – 216^o; (e) 216^o – 259^o; (f) 259^o – 360^o.

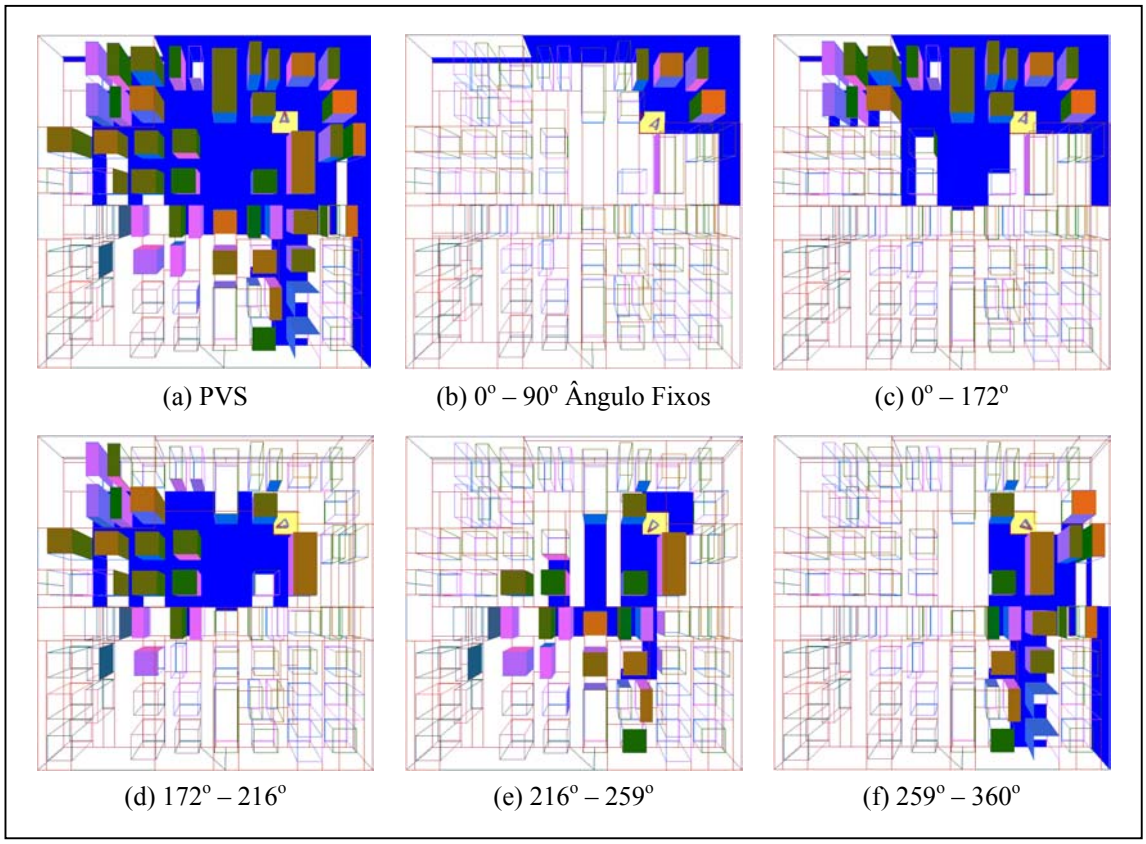


FIGURA 5.6 – SVS gerado a partir do particionamento da visibilidade segundo 4 ângulos usando auto-particionamento

O último tipo de particionamento de visibilidade implementado pode ser visto na Figura 5.7, onde a visibilidade foi dividida segundo a distância entre as regiões. As distâncias particionadoras escolhidas neste exemplo foram: (a) 0 – 50; (b) 50 – 100; (c) 100 – 150; (d) 150 – 5000. Diferentemente dos particionamentos baseados em ângulos, a orientação do observador não foi utilizada para se determinar qual partição será visualizada.

Para comparar a eficiência de diferentes tipos de SVS, foram criados caminhos atravessando a cidade (Figura 5.8). Em cada caminho é testado o número médio de regiões visíveis em cada frame. É testado também o número de regiões visíveis no caminho completo

Para se testar o número de regiões visíveis ao longo de um caminho, soma-se a visibilidade de cada frame. Tal teste equivale a verificar quais partes da cena teriam que ser enviadas pelo servidor ao cliente para possibilitar uma "renderização" correta da cena.

A Tabela 5.5 mostra o número médio de regiões visíveis em cada caminho percorrido. Como era de se esperar, o SVS apresenta um número médio de regiões visíveis muito menor, uma vez que apenas as partições que interceptam o ângulo de visão do observador são consideradas. Apesar do SVS com particionamento dinâmico realizar um melhor balanceamento da visibilidade em cada região, ele pode também criar partições com um ângulo de visão muito pequeno, o que facilita a existência de mais de uma intersecção entre o ângulo de visão do observador e o ângulo de alguma

partição. Dessa forma o SVS com particionamento dinâmico nem sempre apresenta o melhor resultado.

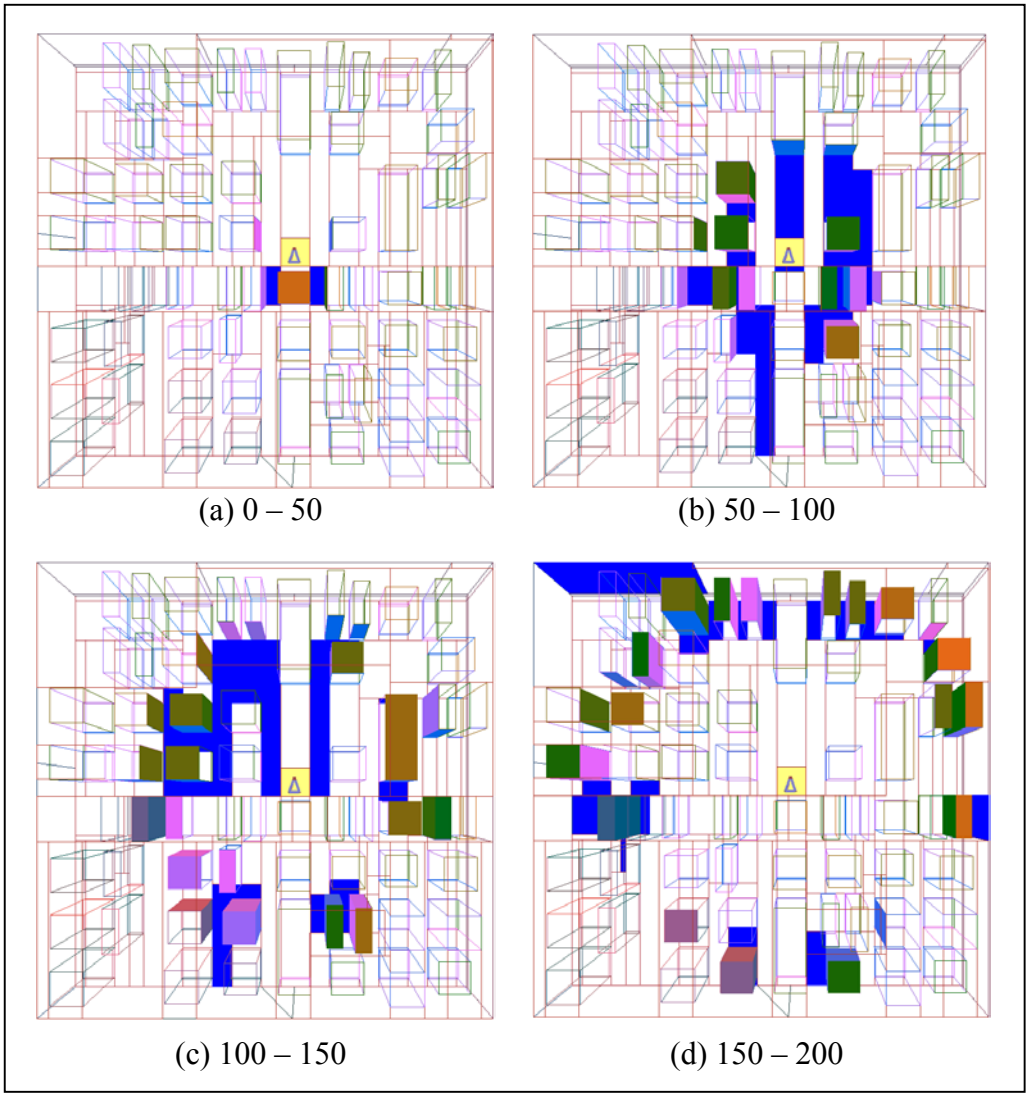


FIGURA 5.7 – SVS gerado a partir do particionamento da visibilidade segundo 4 distâncias

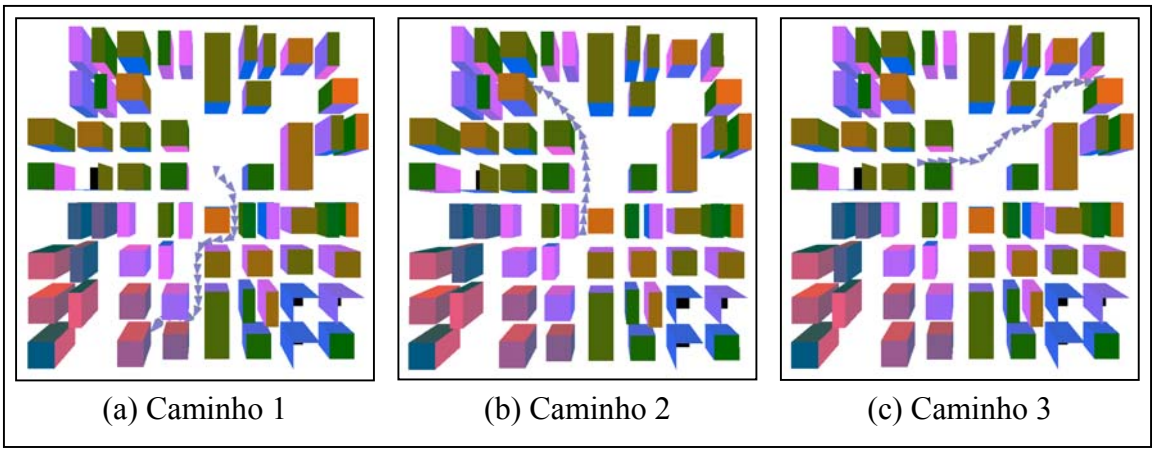


FIGURA 5.8 – Diferentes caminhos em uma cena com 75 prédios

TABELA 5.5 – Número médio de regiões visíveis em cada frame do caminho

Caminho	Número de Frames	PVS	SVS – 3 ângulos	SVS – 4 ângulos	SVS – 4 ângulos – auto-particionamento
1	525	196,735	88,608	88,253	81,741
2	181	263,856	147,779	121,989	152,923
3	602	268,613	137,103	100,846	159,123

Na Tabela 5.6 é apresentado o resultado do cálculo de todas regiões visíveis durante os diferentes percursos. Nota-se que a utilização do SVS chega a diminuir em até 35% (caminho 1 - PVS vs SVS auto-particionado) o número de regiões marcadas como visíveis ao longo do percurso. Em uma aplicação do tipo cliente-servidor, onde a cenário deve ser transmitido através da rede, isso representaria uma economia de 35% dos dados transmitidos.

TABELA 5.6 – Número total médio de regiões visíveis após ser percorrido um caminho

Caminho	Número de Frames	PVS	SVS – 3 ângulos	SVS – 4 ângulos	SVS – 4 ângulos – auto-particionamento
1	525	410	332	346	269
2	181	397	294	301	306
3	602	410	362	355	374

A grande variabilidade dos resultados, ora indicando um tipo de particionamento como o melhor, ora outro, deve-se à diferença entre os caminhos percorridos. Como as partições que interceptam o ângulo de visão do observador devem ser unidas, alguns caminhos irão privilegiar o particionamento que não obrigar a união de partições. Podemos ver um exemplo na Figura 5.9, onde o caminho percorrido não requer a união do primeiro tipo de particionamento, mas requer a união do segundo caso.

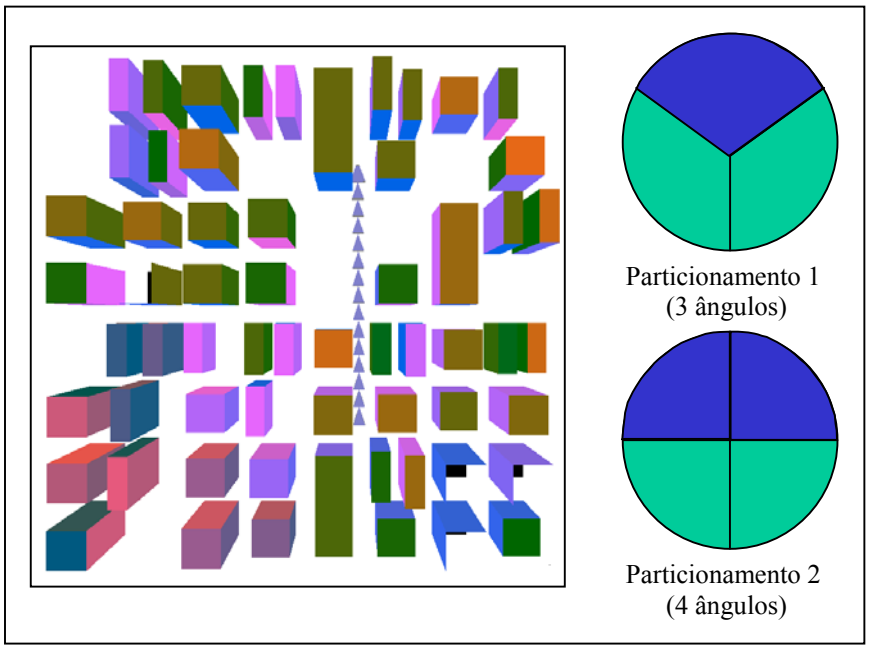


FIGURA 5.9 – Caminho favorecendo um tipo de particionamento (particionamento 1)

Outro fator que deve ser levado em consideração é a velocidade com que os dados são transmitidos ao usuário. Ao utilizarmos um SVS, a quantidade de dados que deverá ser transmitida ao usuário à medida que o caminho está sendo percorrido será menor do que ao usarmos um PVS. Na Tabela 5.7 podemos observar o número de regiões visíveis cumulativamente em diversas etapas do caminho;

TABELA 5.7 – Número total médio de regiões visíveis após ser percorrido um caminho

Andamento	Caminho 1				Caminho 2				Caminho 3			
	PVS	3A	4A	4Aa	PVS	3A	4A	4Aa	PVS	3A	4A	4Aa
0%	278	213	159	159	256	154	170	209	242	222	185	171
10%	320	233	168	177	256	154	170	209	297	274	234	228
20%	352	249	223	201	309	168	193	229	339	313	270	263
30%	360	286	277	222	343	254	222	261	349	315	272	265
40%	368	293	287	236	343	254	222	261	382	336	293	317
50%	385	311	305	242	376	283	251	290	385	339	296	321
60%	404	330	324	261	385	287	254	299	390	339	296	330
70%	404	330	324	261	385	287	254	299	396	339	332	351
80%	404	330	324	261	392	294	261	306	410	362	355	374
90%	410	332	246	269	397	294	301	306	410	362	355	374
100%	410	332	246	269	397	294	301	306	410	362	355	374

Na Figura 5.10 estão ilustrados o resultado cumulativo da visibilidade de cada caminho para cada um dos tipos de particionamento testados. Os polígonos em preto são aqueles que não foram visíveis em nenhum momento.

5.2 Limitações e Problemas Encontrados

Os principais problemas encontrados durante o desenvolvimento deste trabalho foram:

1. Existência de regiões que causavam uma má visibilidade, ou seja, regiões a partir das quais a maioria das outras regiões é visível
2. Impossibilidade de utilização de cenas mais complexas através do aplicativo QVIS3.
3. Dificuldades na implementação do algoritmo *Dual Ray*.

O problema das regiões com má visibilidade foi solucionado através da modificação do aplicativo QBSP3 que passou a particionar folhas muito grandes. Maiores detalhes podem ser encontrados na Seção 4.2.

Os tempos proibitivos no processamento de cenas grandes com o aplicativo de cálculo de visibilidade QVIS3 obrigaram à busca de outras soluções para se realizar esse cálculo. A solução encontrada foi a implementação do algoritmo *Dual Ray* [KOL 2001].

Apesar da implementação desse algoritmo parecer correta, foram necessários testes exaustivos para se garantir que a visibilidade calculada estava realmente correta e era, de fato, conservativa. Em particular, as estruturas chamadas de duplos-triângulos na

descrição do algoritmo podem ser também duplos trapézios. Maiores detalhes de como o bloqueio de visibilidade para cada obstáculo foi calculado podem ser vistos no Seção 4.4.

A utilização do SVS não possui nenhum tipo de limitação. Contudo, ela depende da existência de um cálculo prévio de visibilidade. No ambiente desenvolvido, o cálculo de visibilidade só pode ser realizado para ambientes com obstáculos do tipo 2.5D. Portanto, o SVS herda as limitações do algoritmo utilizado para o cálculo de visibilidade em um ambiente, mas não irá ampliar essas limitações.

Em outras palavras, o SVS provê um refinamento da técnica de PVS e, teoricamente, pode ser utilizado em qualquer aplicação em que o PVS se aplique. A implementação do SVS realizada para este trabalho faz uso do algoritmo *Dual Ray Space* e, portanto, limita-se ao uso em cenas 2.5D.

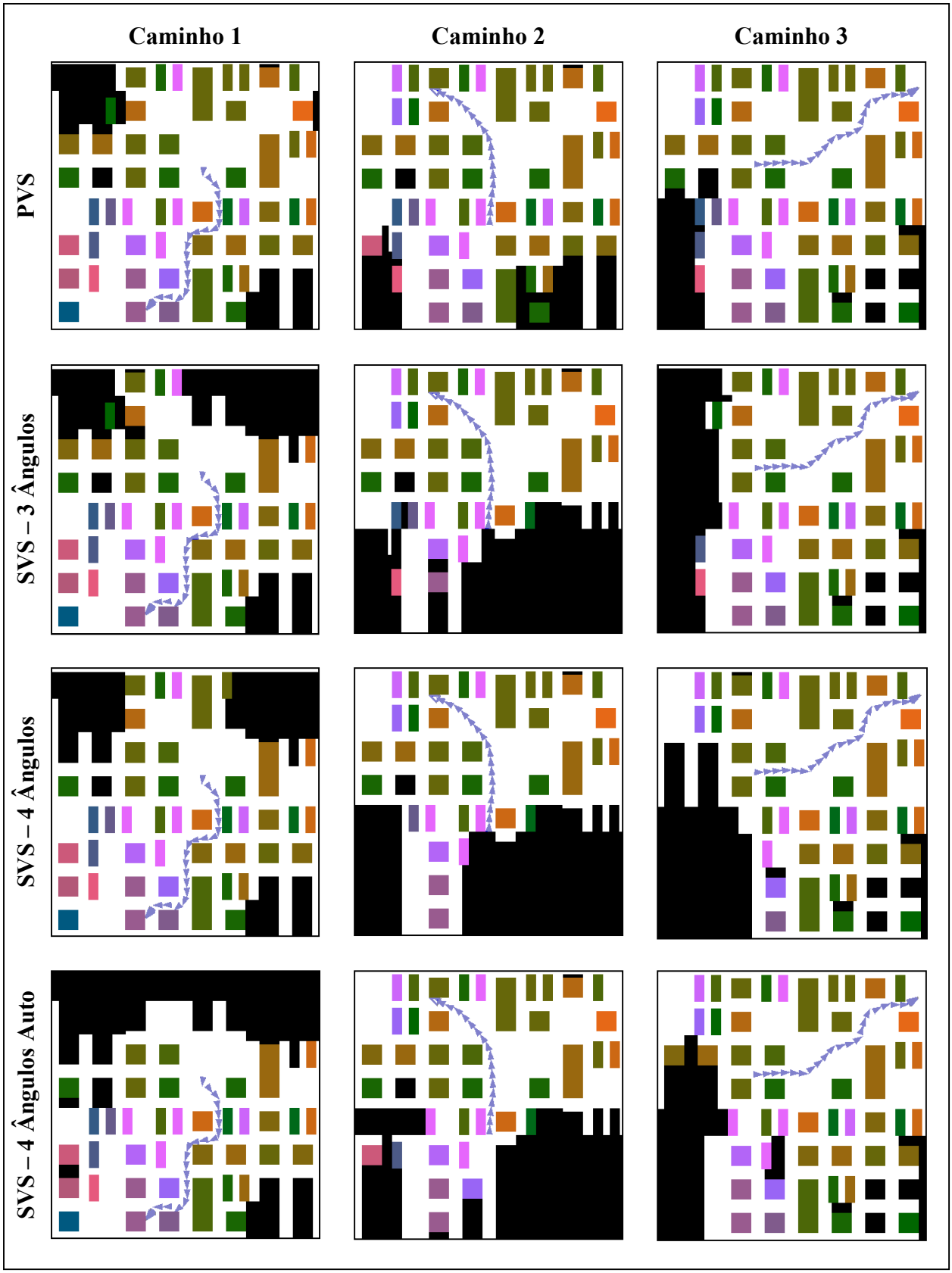


FIGURA 5.10 – Resultado final dos caminhos segundo diferentes particionamentos do SVS

6 Conclusões e Trabalhos Futuros

A crescente utilização de grandes cenários em aplicações do tipo passeio virtual e jogos *online* tem impulsionado o desenvolvimento do hardware gráfico atual. Contudo placas gráficas de grande capacidade não diminuem a quantidade de informação que deve ser trocada entre cliente e servidor. A largura de banda da rede tornou-se o fator limitante em muitas aplicações. Em uma cena onde estão interagindo centenas de clientes, muitas vezes o responsável pela perda de realismo da simulação é a rede, incapaz de transmitir a informação pertinente em tempo hábil, e não o hardware gráfico.

Esse é um problema muito freqüente. Uma solução bastante conhecida para amenizá-lo é a utilização de *Potentially Visible Sets* (PVS). Essa estrutura permite que apenas a informação capaz de alterar o campo de visão do usuário seja transmitida, representando uma grande economia no tráfego de rede. Contudo, certas aplicações ainda podem conter demasiada informação a ser transmitida em cada célula do PVS. Um passeio virtual onde o cenário é transmitido ao usuário pode facilmente extrapolar os limites da largura de banda da rede.

O presente trabalho apresentou uma abordagem para este problema. Os *Smart Visible Sets* (SVS) permitem a divisão da informação de visibilidade em subgrupos, ampliando o conceito dos PVS. Esses subgrupos são divididos através do particionamento do campo de visão e da distância entre as células. Assim, o conceito de "visível" ou "não-visível" encontrado nos PVS é estendido. A informação referente ao conjunto "visível" é especializada para "dentro do campo de visão" ou "fora do campo de visão" e "longe" ou "perto".

A informação referente ao conjunto "visível" é a mais importante. A subdivisão desse conjunto permite um detalhamento dessa informação. O armazenamento dos SVS como matrizes de bits permite ainda uma rápida operação entre diferentes SVS. Outros SVS podem ser adicionados ou subtraídos entre si com um custo computacional muito pequeno permitindo uma rápida alteração no resultado final.

Com uma melhor noção sobre a informação visível podem ser realizados cortes ou ajustes na informação a ser enviada aos clientes. Transmitir apenas a informação dentro de campo de visão do usuário ou não transmitir a informação muito distante do usuário são exemplos dos tipos de ajustes que podem ser realizados para se diminuir o volume de dados enviado. Um controle ainda mais preciso pode ser realizado através da construção de uma métrica que atribui valores aos diferentes conjuntos SVS.

Como o cálculo dos SVS depende da existência da informação de visibilidade entre as células, foi utilizada a aplicação QVIS3 [ID 2001]. Contudo, essa aplicação apresentou problemas de desempenho em cenários muito grandes e com pouca obstrução visual. Foi implementado, então, o algoritmo *dual ray space* [KOL 2001] que apresentou desempenho bastante superior à aplicação anterior.

Assim como o PVS, os SVS podem ser calculados em uma etapa de pré-processamento. Foi provado que o cálculo dos SVS não introduz uma grande sobrecarga na etapa de pré-processamento uma vez que ele representa uma pequena parcela do tempo necessário para o cálculo do PVS.

Para realizar o desenvolvimento e o teste da abordagem aqui construída foram implementadas diversas aplicações. Atenção especial foi devotada ao ambiente gráfico BSPViewer, capaz de ler *BSP-Trees* armazenadas em arquivos e testar o desempenho de

estruturas de dados como PVS, SVS-Angle e SVS-Distance através de diferentes percursos traçados no cenário.

Os testes realizados demonstram que os SVS são uma alternativa viável para a diminuição na quantidade de informação enviada aos clientes. Os maiores ganhos ocorreram nos maiores cenários, exatamente aqueles em que existe uma necessidade de diminuição dos dados enviados. Como os SVS podem ser combinados entre si, a forma como a informação a ser enviada é selecionada pode ser facilmente alterada. A largura de banda da rede do cliente ou o nível de utilização da rede do servidor são fatores que podem ser utilizados para controlar quais informações serão enviadas ao cliente e quais serão descartadas.

Os SVS permitem, também, o ordenamento das informações desde que exista uma métrica que estabeleça valores para os diferentes tipos de SVS. Dessa forma, a informação classificada como "mais importante" pode ser enviada antes. Isso equivale a dizer que a informação que tiver maior importância visual para o cliente estará disponível antes, o que resultará numa melhor experiência da simulação pelos usuários.

Como trabalhos futuros podemos citar formas alternativas de particionamento, numa extensão da abordagem aqui apresentada, e o desenvolvimento de uma aplicação para melhor análise dos resultados obtidos.

Estão sendo estudadas formas alternativas de particionamento do campo de visão 2D. Uma forma bastante interessante é a utilização de particionamento com ângulos não complementares. O objetivo dessa estratégia é reduzir o número de situações em que duas partições devem ser combinadas. Isso ocorre quando o ângulo de visão do observador contém um dos limites das partições. A Figura 6.1 mostra o campo de visão quebrado em ângulos não-complementares com um intervalo de 90^o. Se o usuário apresentasse um ângulo de visão de 60^o a 120^o seriam necessárias duas partições (partição 1 e partição 2) caso fosse utilizado apenas o primeiro particionamento. Com a adição de um segundo particionamento existe uma partição que engloba todo o campo de visão do usuário (partição 6).

Também está sendo estudada a utilização do campo de visão 3D como particionador para o SVS. Esse tipo de particionador será bastante útil em cenários onde os objetos não estão simplesmente dispostos ao longo de dois eixos.

O desenvolvimento de uma aplicação cliente-servidor para permitir uma melhor comparação dos ganhos obtidos com os SVS deve ser baseada na medida da quantidade de informação enviada através da rede com as soluções SVS e PVS. Com esta aplicação será possível medir as taxas de transferência de dados através da rede e os ganhos com diferentes estratégias.

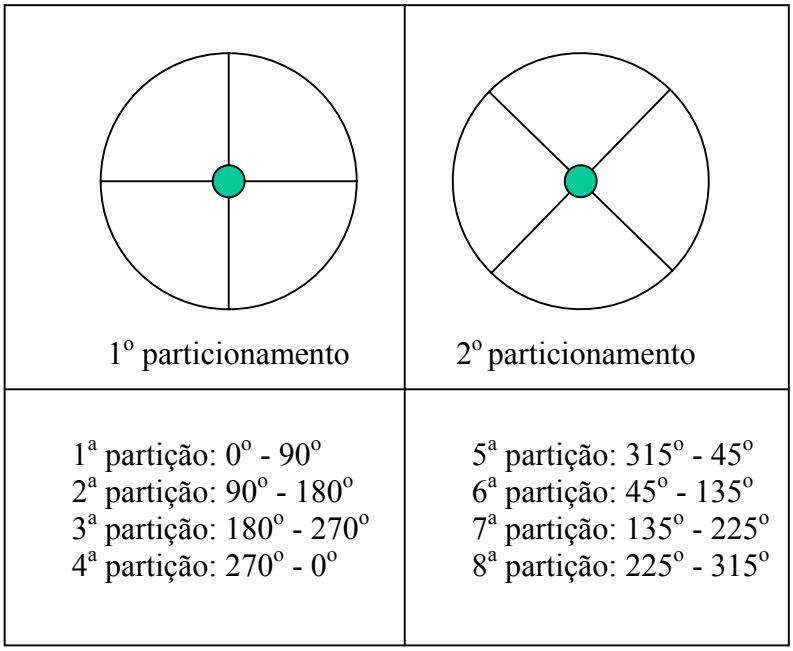


FIGURA 6.1 – Particionamento através de ângulos não-complementares

Anexo

Trabalho apresentado no SIBGRAPI - Brazilian Symposium on Computer Graphics and Image Processing, 2002, Forataleza – CE – Brasil. Submetido para Computer Graphics Forum (1/2003).

Smart Visible Sets for Networked Virtual Environments

Fábio O. Moreira,[1] João L. D. Comba[1] and Carla M. D. S. Freitas[1]

[1] Informatics Institute, Federal University of Rio Grande do Sul, Porto Alegre, RS, Brazil

Abstract

Real-time visualization of complex virtual environments across the network is a challenging problem in Computer Graphics. Pre-computed visibility associated to regions in space, such as in the Potentially Visible Sets (PVS) approach, may reduce the amount of data sent across the network. However, a PVS for a region may still be complex, and further partitions are still necessary. In this paper we introduce the concept of a Smart Visible Set (SVS), which corresponds to (1) a partition of PVS information into dynamic subsets that take into account client position, and (2) an ordering mechanism that enumerates these dynamic sets using a visual importance metric. Results comparing the SVS and the PVS approach are presented

.

1. Introduction

Recent advances in graphics hardware allow the creation of icher and complex scenes, with advanced texture and lightingeffects. Such effects are being extensively explored in omputer games, which continue to request more and more ffects, representing one of the most powerful forces driving gaphics development these days. A promising aspect of the game business is on-line games, where several players interact with each other in a virtual environment across the network. Unlike personal games, on-line games have the challenge of streaming information (environment, players positions, actions, etc) to remote clients, usually referred to as the latency problem. This is a hard problem, as streaming complex environments with lots of geometry and texture can be very expensive. Therefore, a trade-off on streaming speed against environment complexity needs to be made, leading most of the times to simpler environments (seen in on-line massive multiplayer games such as Everquest or Ultima Online).

If networking speed is a limiting factor, one way to improve the final result is to send game information ordered by visual importance to the client, i.e., information describing the part of the environment that is inside the field of view of a client. Solutions to this problem include visibility and occlusion-culling algorithms, multi-resolution and levelof- detail (LOD) representations, or image-based representations of geometry. Integrating these solutions to help solve the latency problem is the focus of our investigation.

The pre-computation of visibility information and storage in Potentially Visible Sets (PVS) have been used in many applications especially in games (e.g. Quake). A PVS consists of a list of objects (polygons or other regions), representing what can be seen from a region in the environment. The PVS is transmitted once for each region, and while the client stays in this region, the PVS of adjacent regions (hopefully predicting the client path) can be sent to maintain full use of network bandwidth, allowing spatial coherence to be explored. However, since a single PVS can still carry more information than the network can handle, further orderings or simplification of geometric detail should be applied.

In this paper we propose a new approach, called Smart Visible Sets (SVS), to allow pre-computed visibility information to be adapted to the needs of a client viewing parameters.

In the SVS, the PVS information is stored in such a way that data can be efficiently adapted to further processing.

One possibility is to organize PVS information into sets, obtained by dividing the original PVS by angles that span the hemisphere of viewing directions. This allows the sets located along the client viewing direction to be sent before the ones behind the viewer. Figure 1 shows a scenario, in which the PVS is divided into 4 sets corresponding to 4 angle intervals. It can be easily observed that the SVS obtained with that division encompasses less information that can be streamed in a convenient way depending on the viewing direction.

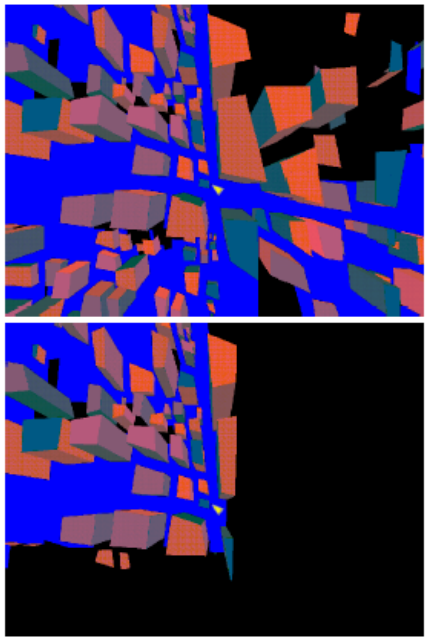


Figure 1: *The PVS and SVS(using 4 angle intervals) associated to the region that contains the viewer (represented by the yellow tetrahedron).*

2. Potentialy Visible Sets

Techniques for rendering complex scenes on networked virtual environments have been reported in the literature lately [9]. These techniques include fast rendering algorithms, efficient ways to stream graphics information, and scene simplification schemes (references [6] and [8] provide discussion on these topics, which will not be specifically surveyed here). In this work, we focus on the rendering aspect of the problem, more specifically in the computation of visibility information, a classical problem in Computer Graphics [5].

One approach to the visibility problem is based on Potentially Visible Sets (PVS) [11, 12, 13]. The PVS represents what is visible from a discrete position or region (cells) of space. During rendering, the PVS can quickly offer visibility information for each viewpoint that lies in a position (or region). Obviously, spatial coherence is better explored in a PVS computed for a cell rather than from a point, but at the expense of having larger sets (see Figure 2 for examples of region-based PVSs).

Although PVS storage can grow very large in complex scenes, the problem of reducing or simplifying PVS information has not been frequently addressed in the literature.
A technique to compress pre-computed visibility sets based on clustering of objects and cells was presented by Van de Panne and Stewart [7], while Gotsman et al. encode visibility information in a hierarchical scheme [2]. Cohen-Or et al. [3, 4] are concerned with the transmission of the visibility sets from servers to clients, or the selection of the best cell size depending on the size of the PVS [1, 4]. More recent results are reported by Koltun et al. [10]. Instead of using a PVS for each cell, these authors propose an intermediate representation that is used for generating the PVS itself during rendering. This intermediate representation is based on virtual occluders, which are a compact representation of the aggregate occlusion for a given cell.

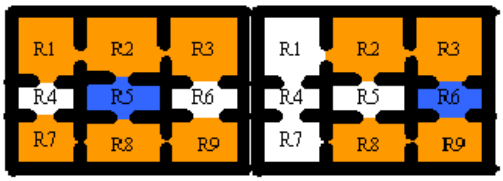


Figure 2: *Region-based PVS: PVS (R5)= R1, R2, R3, R5,R7, R; PVS (R6) = R2,R3,R6,R8,R9*

3. Smart Visible Sets

Smart Visible Sets (SVS) explore ways to break the PVS information into subsets by using an additional parameter, such as viewing angles, or distances from a region. Once the SVS is created, data streaming routines can search the SVS to obtain the most relevant data to be sent first, or to allow culling of non-crucial data. In this work we focused on citylike environments (2.5D), where the discussion of breaking viewing directions is simplified to a planar problem, but nevertheless has interesting aspects and allows us to explore the SVS concept.

The spatial data structure used in this work is the Binary Space Partitioning Tree (BSP-Tree), but other hierarchical data structures could be used as well. The PVS information is computed for each leaf cell of the BSP-tree, storing cell to-cell visibility information. The cells have an unique identification number, which can be encoded using O(log n) bits, where n is the number of cells. The PVS for each cell is stored as an array of n bits (with each ith bit turned on to represent that the ith cell is visible from the current cell). The SVS storage is similar, except that more than one set is created. Like the PVS, SVS is determined in a pre-processing step, causing no impact to the performance of the applications that use them.

3.1. Breaking Visibility by Angle

One way to split a PVS into different subsets is to break the hemisphere of viewing directions into sets according to the viewing frustums. Since our datasets are 2.5D, we use the azimuth to split the PVS, discarding frustum tilt or elevation. Deciding which are the best angles to split the PVS is the question to be answered

Constant Number of Angles and Orientations

This approach fixes a constant set of splitting frustums, breaking the PVS of all cells by the same angles (userdefined). To perform the split, the frustum volume for a given angle interval is computed. Because our datasets are 2.5D, the problem is simplified to a planar problem (Figure 3). For each angle interval, the splitting frustum limits is used to determine two supporting lines, which are

then used to verify the cells that lie inside this frustum. Cells that straddle frustums are included in all partitions they intersect. Further speed-up is obtained by using the original PVS (which corresponds to the visible sets for all angle directions), and only performing the frustum visibility checks on the cells already marked as visible in the PVS.

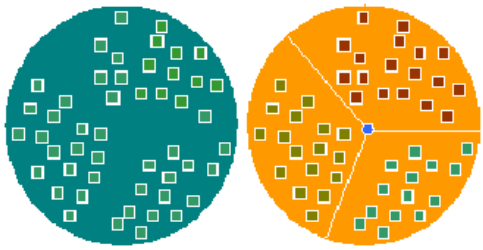


Figure 3: *The PVS (left) contains information about all viewing directions. In the SVS (right) the information of the PVS is divided into sets by a fixed angle (3 angles, in this example).*

The geometry of the cells associated with BSP leaves, necessary to the above computation, is computed using a depth-first traversal of the BSP. First, the bounding volume of the scene is computed, corresponding to the cell of the BSP root. For each visited node, we split the current bounding volume into two (using the partitioning plane stored at the node), using the result to update the bounding volumes of the left and right subtrees, and the computation proceeds recursively. For every visited leaf, the bounding volume is stored as the leaf cell.

For each leaf cell and angle interval, we define the frustum by two supporting lines, emanating from an anchor point along directions defined by the angle limits. A vertex on the bounding box of the leaf cell is selected as anchor point depending on (1) the angle used, and (2) if it corresponds to the frustum start or end angle (Figure 4).

Cell-to-cell visibility is computed by checking if the bounding volume of the target cell lies completely outside of both frustum limits using the two supporting lines (a nonvisible leaf), and stored as bit array, along with the viewing frustum limits. For each leaf, various sets of bit arrays are stored, instead of just one, like in the PVS. In the PVS approach, the visibility for a given leaf corresponds to check if the corresponding bits are set. In the SVS split by angles, Figure 5 shows how different viewing frustums generate different visibilities in a SVS split by three angles. Instead of storing a single bit array, three arrays are used for the three angle ranges respectively. The original PVS information is a bitwise OR of the three arrays.

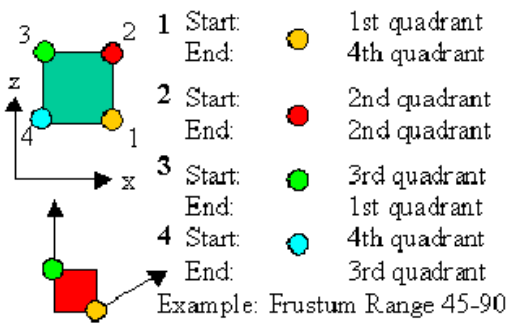


Figure 4: *Determining the supporting lines. Vertices marked as 1, 2, 3 and 4 are candidate for anchor points, depending on the starting and ending angle* a bitwise OR of the bit arrays whose angles intercept the user viewing frustum suffices.

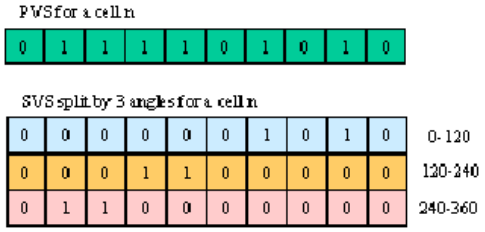


Figure 5: *SVS storage by 3 angles*

Adaptive Number of Angles and Orientations

The main problem with the constant angle approach is that it is common to a SVS set to store too many visible cells, while others store just a few. If the SVS sets become unbalanced, it will not be very efficient. The ideal solution would be the one that breaks the visible information associated with each angle into a size proportional to the number of angles used.

- In the adaptive approach, different splitting frustums are selected for each cell. A user-defined parameter indicates how many candidates will be tested as well as the maximum number of splitting frustums. The algorithm is described as follows:
- Starting with a 360 degree frustum, choose a set of splitting candidates (the number of candidates that are actually tested is picked by the user).
- Split the frustum using the candidates and choose the best split. The best split is the one that generates the lowest absolute value: eval_split = Number of visible cells in the 1st frustum - Number of visible cells in the 2nd frustum value.
- Recursively choose candidates and pick the best splits using the two halves of the frustum until the maximum number of splitting frustums has been reached. Once the splitting frustums are chosen, the cell-to-cell visibility calculation and the storage procedures are the same as in the constant approach (figure 6).

Once the splitting frustums are chosen, the cell-to-cell visibility calculation and the storage procedures are the same as in the constant angle approach.

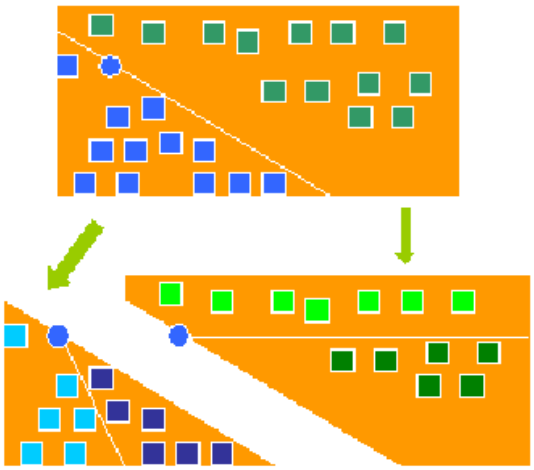


Figure 6: *SVS by angles using the adaptive approach.*

3.2. Breaking Visibility by Distance

Another way to split the PVS into different subsets is to use the minimum distance between cells, which estimates how far one cell is from another.We approximate this measure by computing the minimum distance among all pairs of faces of both cells. (Figure 7).

By setting a few distance intervals, we are able to build a new SVS that adds distance to the visibility information. Again, we store the distance-based SVS as bit arrays to makeb the bitwise operations easier. Each distance-SVS is stored in ascending order, along with the distance previously calculated and a list of the visible cells within that distance. The original PVS is used to select the cells to perform distance calculations. If we want to take into account the distance between the cells we can do a bitwise AND between the visibility result (bitwise OR of all SVS that intersects the viewing frustum) and the SVS that represents the desired distance range.

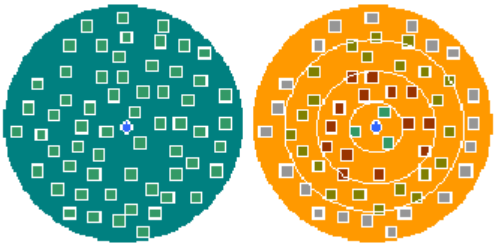


Figure 7: *SVS using distance breaks the sets using concentric circles, which can be used for multi-resolution or LOD.*

By changing the distance-SVS that is being ANDed, the user can quickly change the rendering quality of the scene.
We are also exploring an adaptive solution to generate the best possible rendering quality of the scene while still maintaining a desirable frame rate. Another possibility is combining distance-based SVS and level of detail (LOD) strategies. Checking the distance from objects to the user requires simple calculation of which cell the object is inside. Once we have that information, the SVS can be checked to see what level of detail should be used.

3.3. Breaking Visibility by Angle and Distance

Combining both angle and distance approaches into a single SVS is another possible way of splitting the PVS. In this case, a larger number of sets are created as can be observed in Figure 8.

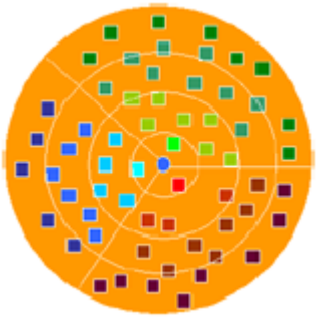


Figure 8: *SVS using angle and distance*

In this case, there are much possible ordering combinations since the system might need, for example, to associate different priorities for transmission depending on angles and distances (next section introduces visual importance metrics as a usage for this strategy).

3.4. Visual Importance Metrics

The main purpose of the SVS is to provide a fast ordering of visibility information. In a client-server environment, we can order the information that needs to be sent to the client using the SVS, thus sending more relevant data first. It is easy to see how data that lies inside the client's viewing frustum and closer to the client's position should have a higher priority over data that is far away and outside the viewing frustum.

Using the SVS we can determine which cells are closer or/and inside the client's viewing frustum with a small number of bitwise operations and then order the information that needs to be sent across the network according to its visual importance to the client.

Without a defined visual importance metric it is hard to answer questions such as: "Should data that is closer to the client but outside its frustum be sent before data that lies inside the viewing frustum but is farther away?". We are currently working on a

graph based metric that will allow quick reconfiguration and testing of the visual quality of the scene.

4. Results and Discussion

We are using the PVS generated by the QBSP3 and QVIS applications (developed by ID Software) as the support code for our work. In order to validate the proposed algorithm, we developed a 3D application that allows the following operations:

- Loading of scenes
- Visualization of the bounding boxes of each cell in the scene
- Visualization of the original PVS
- Visualization of SVS (viewing frustum and distance), individually or merged with any number of other SVS
- Cell culling based on the user viewing frustum (using the viewing frustum SVS)
- Cell culling based on the distance from the user cell to other cells (using the distance SVS)
- Creation, storage and loading of walkthrough paths (that are later used for benchmarking)
- A free camera mode and an user camera mode
- Benchmarks that take into account: raw number of cells rendered in each frame and total number of cells visible along a path

The two camera modes were created to make visualization of information easier. All culling and visualizations are done based on the user camera and position. The user himself is represented as tetrahedron in the scene. By using the free camera mode, we can walk around the scene without changing the current culling or visualization parameters, making it easier to check if the algorithms are working as intended.

Finally, the possibility of easily changing the rendering parameters and creating walkthrough paths allows the comparison between the SVS and PVS methods and among SVS built with different splitting parameters. The environment was implemented using C++ and OpenGL.

Our first benchmark (Table 1) shows the time needed to generate different SVSs for a scene with 600 buildings, represented by a BSP tree with 1257 leaves. Measurements were taken in an Athlon 1.5 computer with 256 MB RAM, Widows 2000. Table 1 shows also the PVS generation time using the source code provided in the Quake package by ID Software, based on portal calculations which are more appropriated to indoor scenes. We are currently investigating better PVS algorithms for city datasets [10]. The SVS calculation is much faster as it is done in 2D, and relies on the PVS to prune the queries.

In Table 2 we compare the performance of different SVSs and PVS. Results were obtained by running a set of different paths in an environment with 600 buildings. All paths were executed using different splitting options, which include: splitting in three angles, splitting in four angles and splitting in four angles using the adaptive algorithm previously described. During the execution of the path, the visible leaves on each frame were marked. After the path was completed, the total number of marked leaves allowed us to measure how much information would need to be sent to the client using that particular partitioning strategy. Path 1 starts in the center of the city and moves to the upper side; path 2 starts in the lower left corner and moves to the upper right corner of the city, and finally path 3 is a straight line from one side of the city to the other.

Function	Time
PVS using Quake	2h37m40s
SVS (3 angles)	14s
SVS (4 angles)	16s
SVS (4 angles, 10 splitters)	137s
SVS (4 distances)	12s

Table 1: *PVS and SVS computation times.*

In Figure 9 and 10 we show the results from an orthographic view. Darker buildings are the ones that are visible in the PVS but have been culled by the rendering engine because they lie in a SVS that do not intersect the user's viewing frustum (the user is represented by the tetrahedron located in the center of the image).

	Path 1	Path 2	Path 3
PVS	942 (100%)	1053 (100%)	1084 (100%)
SVS 3 Angles	809 (85.8%)	1012 (96.1%)	686 (63.2%)
SVS 4 Angles	776 (82.3%)	987 (93.7%)	937 (86.4%)
SVS 4 Angles 10 Splitters	752 (79.8%)	966 (91.7%)	884 (81.5%)

Table 2: *Number of visible cells along different paths with different partitioning strategies.*

.

Figure 11 illustrates the SVS for two different cells, taking into account a current viewing frustum. In order to understand how information is divided into angles and distances by the SVS, we use different color shades and display them side-by-side.

5. Conclusions

Client-server applications over the network often have to deal with the problem of having too much

information to send to its clients, or too many clients connected. They need fast and reliable ways to cull that information when needed.

In this paper we introduced the idea of Smart Visible Sets, and explained the algorithms used for its implementation. The SVS is a tradeoff between space (both memory and storage) and performance. It can be efficiently used to sort server data according to its visual importance to the clients. Then, the more important information for rendering each scene can be sent first, resulting in a smoother experience of the simulation by the user. It can also be used to cull part of the information (the least important part of it) if the network traffic exceeds a limit (either the server limit or the client limit). In a server with a big number of clients, or in slower network environments, the use of SVS can represent a big leap in performance.

We also performed a series of tests that show the advantages of the adaptive approach over the constant approach for choosing splitting angles. We have also shown that the SVS generation does not introduce a significant overhead in the pre-computation of visibility information since it is only a small fraction of the time needed for the PVS determination.

We have not yet executed tests in very large environments because the tools we used to generate the PVS perform poorly on large open areas. We are currently studying the use of different PVS generation algorithms (like the one described in [14]).

The next steps in this work are (a) developing a clientserver application and comparing the visual results of the SVS and PVS solutions; and (b) studying the possibility of using 3D-frustum as splitters for the SVS.

Acknowledgements

This work has been supported by CNPq and FAPERGS.

References

1. B. Nadler, G. Fibich, S. LevYehudi, and D. Cohen-Or, A qualitative and quantitative visibility analysis in urban scenes. *Computer & Graphics*, **23**(5):655-666, 1999.

2. C. Gostman, O. Sudarsky, and J. Fayman, Optimized occlusion culling. Computer & Graphics, 23(5):645- 654, 1999.

3. D. Cohen-Or and E. Zadicario, Visibility streaming for networked-based walkthroughs. Graphics Interface'98, pp. 1-7, 1998.

4. D. Cohen-Or, G. Fibich, D. Halperin, and E. Zadicario, Conservative visibility and strong occlusion for viewspace partitioning of densely occluded scenes. Computer Graphics Forum, 17(3):243-254, 1998.

5. D. Cohen-Or, Y. Chrysanthou and C. Silva. A survey of visibility fo walkthrough applications. In: SIGGRAPH 2000 Course Notes - Visibility: problems, techniques and applications, July 2000.

6. E. Teler and D. Lischinski, Streaming of complex scenes for remote walkthroughs. Computer Graphics Forum, 20(3): C18-C25, 2001.

7. M. van de Panne and J. Stewart, Efficient compression techniques for precomputed visibility. In Eurographics Workshop on Rendering, 1999.

8. P. Pires and J. Pereira, Dynamic algorithm binding for interactive walkthroughs. Proceedings of SIBGRAPI 2001, pp. 154-161, IEEE Computer Society Press, 2001.

9. S. Singhal and M. Zyda. Networked Virtual Environments. Addison-Wesley, 1999.

10. V. Koltun, Y. Chrysanthou and D. Cohen-Or, Virtual occluders: an efficient intermediate PVS representation, Eurographics Workshop on Rendering, pp. 59-70, Eurographics, 2000.

11. Airey, J. M., Rohlf, H. J, and Brooks, F. P. Towards Image Realism with Interactive Update Rates in Complex Virtual Environments. Computer Graphics (1990 Symposium on Interactive 3D Graphics), vol. 24, no. 2, pp. 41-50, March 1990.

12. Teller, S, and Sequin C. Visibility Preprocessing For Interactive Walkthroughs. Computer Graphics (Siggraph 91 Proceedings), pp 123-132, July 1998.

13. Teller, Seth J. Visibility Preprocessing For Interactive Walkthroughs. Ph.D. Thesis, Department of Computer Science, University of Berkeley, 1992.

14. Vladlen Koltun, Daniel Cohen-Or and Yiorgos Chrysanthou, Hardware-Accelerated From-Region Visibility Using a Dual Ray Space. 12th Eurographics Workshop on Rendering, pp. 204-214, 2001.

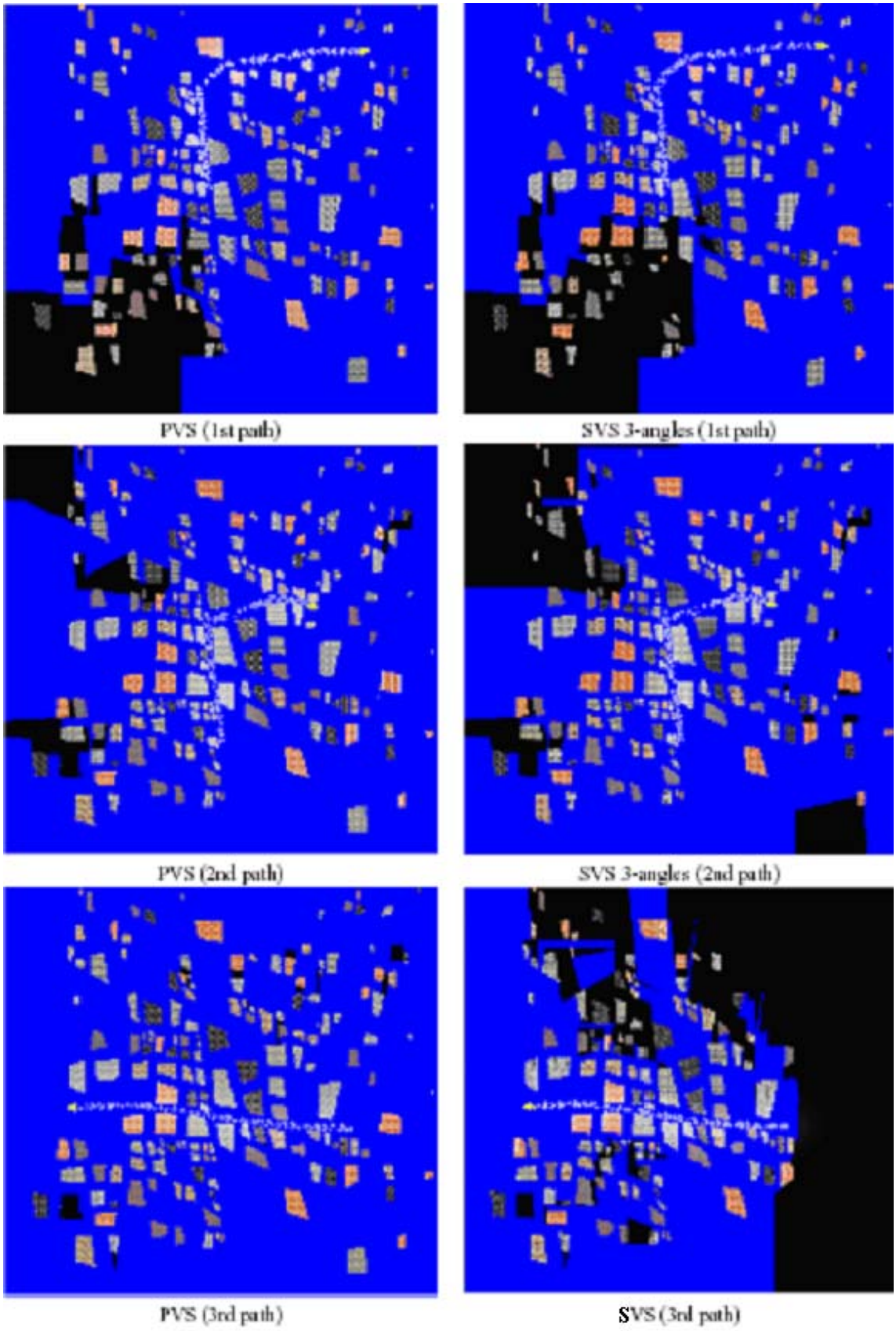


Figure 9: *Scene comparing PVS and SVS (3-angles) in 3 different paths. Visible cells are blue and culled information is black.*

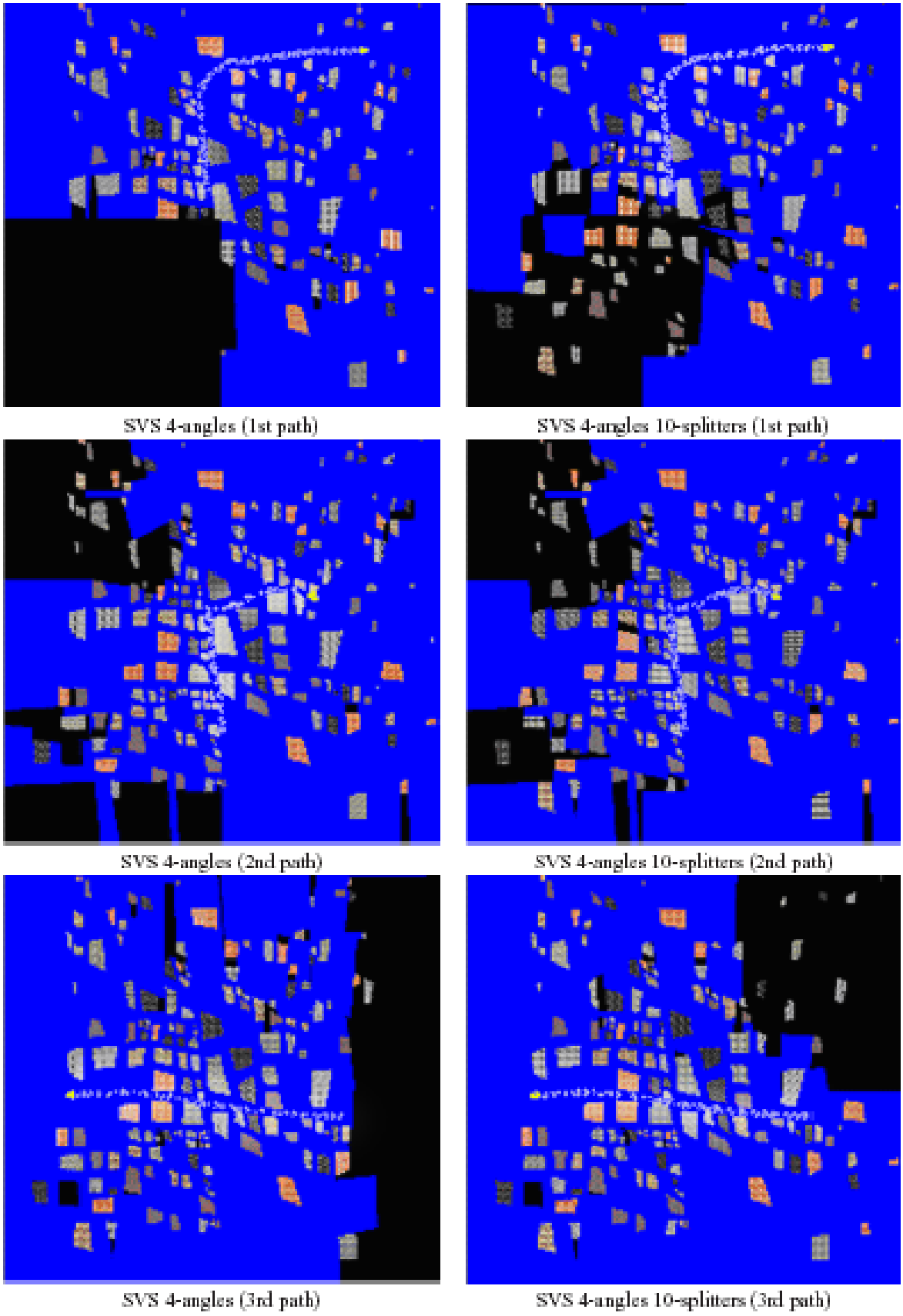


Figure 10: *Scene comparing different SVSs in 3 different paths. Visible cells are blue and culled information is black.*

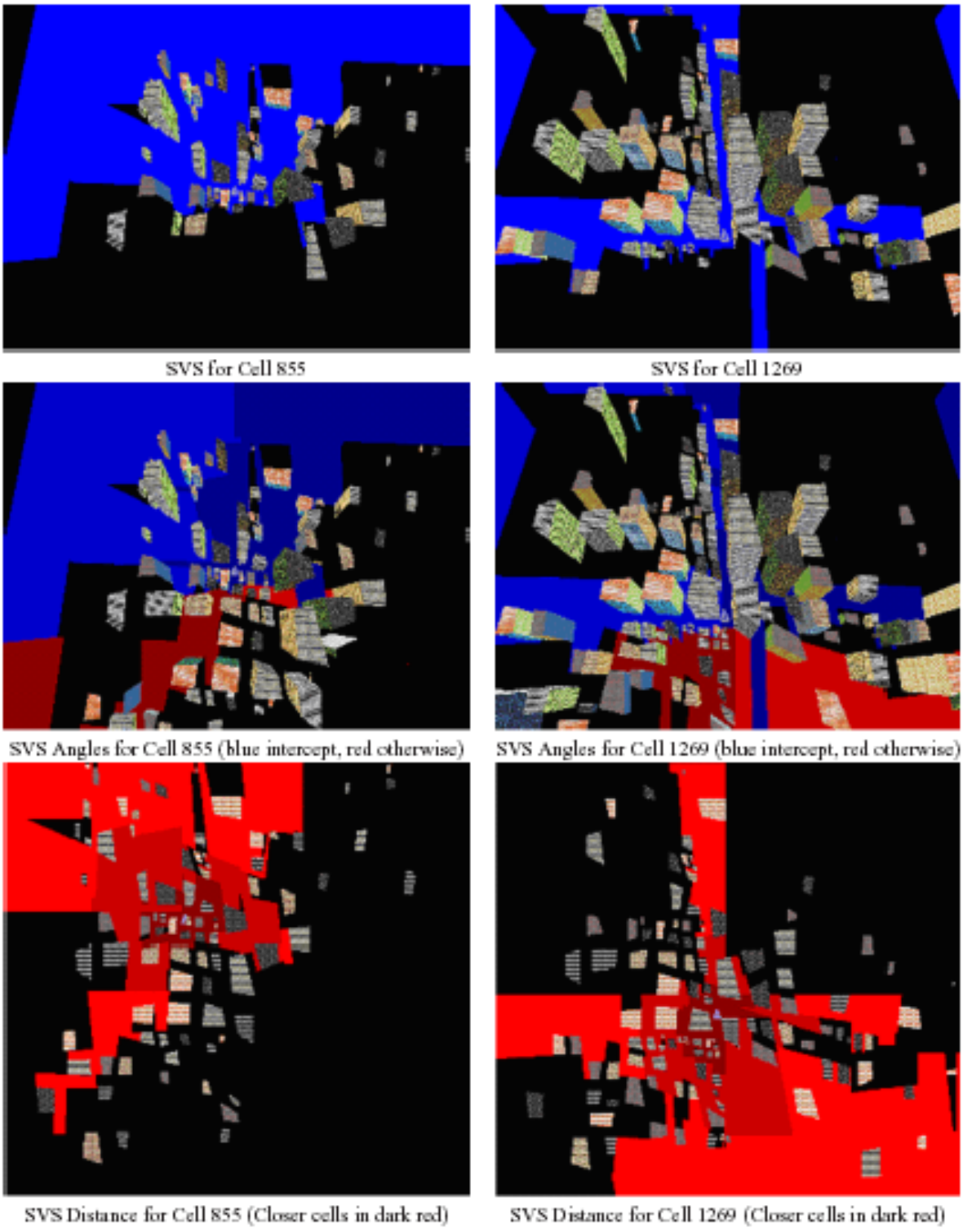


Figure 11: *SVS information (divided by angles and distances) for two different cells.*

Bibliografia

[AIR 90] AIREY, John M.; ROHLF, John H.; BROOKS JR., Frederick P. Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments. In: SYMPOSIUM ON INTERACTIVE 3D GRAPHICS, 1990. **Proceedings…[**S.l.:s.n], 1990. p.41–50.

[AIR 91] AIREY, John. **Increasing Update Rates in the Building Walkthrough System with Automatic Model - Space Subdivision and Potentially Visible Set Calculations**. 1991. PhD thesis - University of North Carolina, Chappel Hill.

[APP 68] APPEL, Arthur. Some Techniques for Shading Machine Renderings of Solids. In: JOINT COMPUTER CONFERENCE, 1968. **Proceedings…** Washington: Tompson Books, 1968. p.37–45.

[BAL 99a] BALA, Kavita; DORSEY, Julie; TELLER Seth. Radiance Interpolants for Accelerated Bounded-error Ray Tracing. **ACM Transactions on Graphics**, New York, v.18 n. 3 p.213–256, July 1999.

[BAL 99b] BALA Kavita; DORSEY, Julie; TELLER, Seth. Ray-traced Interactive Scene Editing Using Ray Segment Trees. In: EUROGRAPHICS RENDERING WORKSHOP, 1999. **Proceedings…** [S.l.: s.n.], 1999.

[BAR 2001] BARTZ, D.; KLOSOWSKI, J.; STANEKER D. K-dops as Tighter Bounding Volumes for Better Occlusion Performance. In: ACM SIGGRAPH VISUAL, 2001. **Proceedings…** [S.l.: s.n.], 2001.

[BAR 98] BARTZ, Dirk; MESSNER, Michael; and HTTNER,Tobias. Extending Graphics Hardware for Occlusion Queries in OpenGL. In: WORKSHOP ON GRAPHICS HARDWARE, 1998. **Proceedings…** [S.l.: s.n.], 1998. p.97–104.

[BER 2000] BERNARDINI, F.; KLOSOWSKI, J. T.; V J. Directional Discretized Occluders for Accelerated Occlusion Culling. **Computer Graphics Forum**, Amsterdam, v.19, n.3, p.507–516, 2000.

[BIT 98] BITTNER, J.; HAVRAN, V.; STAVIK, P. Hierarchical Visibility Culling with Occlusion Trees. In: CONFERENCE ON COMPUTER GRAPHICS INTERNATIONAL, 1998. **Proceedings…** [S.l.: s.n.], 1998. p.207-219.

[CAT 74] CATMULL, E. **A Subdivision Algorithm for Computer Display of Curved Surfaces.** 1974.Ph.D. thesis - University of Utah, Utah.

[CHI 89] CHIN, N.; FEINER, S. Near Real-time Shadow Generation Using BSP Trees. **ACM Computer Graphics**, New York, v.23, n.3, p.99–106, 1989.

[COH 95] COHEN-OR, Daniel; SHAKED, Amit. Visibility and Dead-zones in Digital Terrain Maps**. Computer Graphics Forum**, Amsterdam, v.14, n.3, p.171–180, Aug. 1995.

[COH 96] COHEN-OR, Daniel, RICH Eran; LERNER, Uri; SHENKAR, Victor. A Real-time Photo-realistic Visual Flythrough. **IEEE Transactions on Visualization and Computer Graphics**, Los Alamitos, v.2, n.3, p.255–264, Sept. 1996.

[COH 98] COHEN-OR, Daniel; FIBISH, Gadi; HALPERIN, Dan; ZADICARIO, Eyal. Conservative Visibility and Strong Occlusion for Viewspace Partitioning of Densely Occluded Scenes. **Computer Graphics Forum**, Amsterdam, v.17, n.3, p.243-254, 1998.

[COH 2003] COHEN-OR, Daniel; CHRYSANTHOU, Yiorgos; SILVA, Claudio; DURAND, Fredo. A Survey of Visibility for Walkthrough Applications. To appear in IEEE TVCG(TVCG03).

[COO 96] COORG, Satyan; TELLER, Seth. Temporally Coherent Conservative Visibility. In: ANNU. ACM SYMPOSIUN COMPUT. GEOM., 12., 1996. **Proceedings…** [S.l.: s.n.], 1996. p.78–87.

[COO 97] COORG, Satyan; TELLER, Seth. Real-time Occlusion Culling for Models with Large Occluders. In: SYMPOSIUM OF INTERACTIVE 3D GRAPHICS, 1997. **Proceedings…** [S.l.: s.n.], 1997. p.83-90.

[DEC 99] DECORET, Xavier; SCHAUFLER, Gernot. Multi-layered Impostors for Accelerated Rendering. **Computer Graphics Forum**, Amsterdam, v.18, n.3, Sept. 1999.

[DUR 97] DURAND, Frédo; DETTAKIS, George; PUECH, Claude. The Visibility Skeleton: A Powerful and Efficient Multi-Purpose Global Visibility Tool. **Computer Graphics**, LosAngeles, p.89-100, Aug. 1997. Trabalho apresentado em SIGGRAPH 1997.

[DUR 2000] DURAND, Frédo et al. Conservative Visibility Preprocessing Using Extending Projections. In**:** SIGGRAPH CONFERENCE, 2000. **Computer Graphics Proccedings**. New York:ACM, 2000. p.239-248.

[FEI 76] FEINER, J. H. X. Hierarchical Geometric Models for Visible Surface Algorithms. **Communications of the ACM**, New York, v.19, n.10, p.547–554, Oct. 1976.

[FOL 90] FOLEY, James D et all. **Computer Graphics:** principles and practice*,* 2nd ed. Reading, Massachusetts: Addison-Wesley, 1990.

[FUC 80] FUCHS, H.; KEDEM, Z.; NAYLOR, B. On Visible Surface Generation by A Priori Tree Structures. **Computer Graphics**, New York, v.14, n. 3, p.124-133, July 1980.

[GRE 93] GREENE, Ned; KASS, M. Hierarchical Z-buffer visibility. In: SIGGRAPH**,** 1993. **Proceedings…** [S.l.: s.n.], 1993. p.231-240.

[GRE 94] GREENE, Ned; KASS, Michael . Error-bounded Antialiased Rendering of Complex Environments. In: SIGGRAPH, 1994. **Proceedings…** [S.l.: s.n.], 1994. p.59–66.

[GRE 2001] GREENE, N.. **Occlusion Culling with Optimized Hierarchical Z-buffering.** Apresentado no Course Notes #30 da ACM SIGGRAPH, 2001.

[HUD 97] HUDSON, T.; MANOCHA, D.; COHEN, J.; LIN, M.; HOFF, K.; V H. X. Accelerated Occlusion Culling Using Shadow Frustra. In: ACM SYMPOSIUM COMPUT. GEOM., 13., 1997. **Proceedings…** [S.l.: s.n.], 1997. p.1–10.

[ID 2001] ID SOFTWARE. **QBSP3, QVIS3, QRAD3 Utilities**. Disponível em: <http://www.joz3d.net/files/Games/Quake/2/q2utils.exe>. Acesso em: 15 abr. 2001.

[JIM 2000] JIMÉNEZ, W. F. H.; ESPERANÇA, C.; OLIVEIRA A. A. F. Efficient Algorithms for Computing Conservative Portal Visibility Information. **Computer Graphics Forum**, Amsterdam, v.19, n.3, p.489–498, Aug. 2000.

[JON 71] JONES, C. B. A New Approach to the 'Hidden Line' Problem. **Computer Journal,** Cambridge, v.14, n.3, p.232–237, Aug. 1971.

[KLO 99] KLOSOWSKI, James T.; SILVA, Cláudio T. Rendering on a Budget: A Framework for Time-critical Rendering. In: IEEE VISUALIZATION, 10., 1999. **Visualisation 99 :** proceedings. New York, 1999. p.115–122.

[KLO 2000] KLOSOWSKI, James T.; SILVA Cláudio T. The Prioritized-layered Projection Algorithm for Visible Set Estimation. **IEEE Transactions on Visualization and Computer Graphics**, Los Alamitos, v.6, n.2, p.108–123, Apr./June 2000.

[KLO 2001] KLOSOWSKI, James T.; SILVA, Cláudio T. Efficient Conservative Visibility Culling Using the Prioritized Layered Projection Algorithm. **IEEE Transactions on Visualization and Computer Graphics**, Los Alamitos, v.7, n.4, p.365 – 379, Oct./Nov. 2001.

[KOL 2000] KOLTUN, Vladlen; CHRYSANTHOU, Yiorgos; COHEN-OR Daniel. Virtual Occluders: An Efficient Intermediate PVS Representation In: EUROGRAPHICS WORKSHOP ON RENDERING, 11., 2000. **Proceedings…** [S.l.: s.n.], 2000. p.59–70.

[KOL 2001] KOLTUN, Vladlen; CHRYSANTHOU, Yiorgos; COHEN-OR Daniel.

Hardware-accelerated From-region Visibility Using a Dual Ray Space. In: EUROGRAPHICS WORKSHOP ON RENDERING, 12., 2001. **Proceedings…** [S.l.: s.n.], 2001. p.205–216.

[LU 2002] LU, L.; WANGAND, Z.; BOVIK, A. C. Scalable Foveated Visual Information Coding and Communications. Invited Paper, International Conference of Communications, Circuits and Systems. June/July 2002.

[LUE 95] LUEBKE, David; GEORGES Chris. Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets. In: ACM SIGGRAPH, 1995. **Symposium on Interactive 3D Graphics:** proceedings.[S.l.] : Pat Hanrahan and Jim Winget], 1995. p.105–106.

[LUE 2002] LUEBKE, David et al. **Level of Detail for 3D Graphics**. [S.l.] : Morgan Kaufmann, 2002.

[MAC 95] MACIEL, Paulo W. C.; SHIRLEY, Peter. Visual Navigation of Large Environments Using Textured Clusters. In: ACM SIGGRAPH, 1995. **Symposium on Interactive 3D Graphics:** proceedings.[S.l.] : Pat Hanrahan and Jim Winget], 1995. p.95-102.

[MEA 82] MEAGHER, D. Efficient Synthetic Image Generation of Arbitrary 3D Objects. In: IEEE COMPUTER SOCIETY CONFERENCE ON PATTERN RECOGNITION AND IMAGE PROCESSING, 1982. **Proceedings…** [S.l.: s.n.], 1982. p.473–47.

[NAY 92] NAYLOR, Bruce F. Partitioning Tree Image Representation and Generation from 3D Geometric Models. In: GRAPHICS INTERFACE, 1992. **Proceedings…** [S.l.: s.n.], 1992. p.201–212.

[PAR 99] PARKER, Steven et al. Interactive Ray Tracing. In: ACM SYMPOSIUM ON INTERACTIVE 3D GRAPHICS, 1999. **Proceedings…** [S.l.: s.n.], 1999. p.119–126.

[SCH 2000] SCHAUFLER, Gernot; DORSEY, Julie; DECORET, Xavier; SILLION, François. Conservative Volumetric Visibility with Occluder Fusion. In: SIGGRAPH, 2000. **Proceedings…** [S.l.: s.n.], 2000. p.229-238.

[SIL 95a] SILLION, François X. A Unified Hierarchical Algorithm for Global Illumination with Scattering Volumes and Object Clusters. **IEEE Transactions on Visualization and Computer Graphics**, Los Alamitos, v.1, n.3, p.240–254, Sept. 1995.

[SIL 95b] SILLION, François X.; DRETTAKIS, George. Feature-based Control of Visibility Error: A Multi-resolution Clustering Algorithm for Global Illumination. In: SIGGRAPH, 1995. [S.l.: s.n.], 1995. p.145–152.

[STA 99] STANEKER, Dirk; MEINER, Michael; HÜTTNER Tobias. OpenGL-assisted Occlusion Culling for Large Polygonal Models. **Computer &**

Graphics, New York, v.23, n.5, p.667–679, 1999.

[SUD 99] SUDARSKY; Oded; GOTSMAN, Craig. Dynamic scene occlusion culling. **IEEE Transactions on Visualizationand Computer Graphics**, Los Alamitos, v.5, n.1, p.13–29, Jan./Mar.1999.

[SUT 74] SUTHERLAND, I. E.; SPROULL, R. F.; V R. A.. A Characterization of Ten Hidden Surface Algorithms. **ACM Computer Surveys**, New York, v**.**6, n.1, p.1–55, Mar. 1974.

[SUT 99] SUTER, Jaap. **Introduction To Octrees**. Disponível em: <http://www.flipcode.com/tutorials/tut_octrees.shtml>, Acesso em: 20 de abr. 1999.

[TEL 91] TELLER, Seth; SÉQUIN, Carlo. Visibility Preprocessing for Interactive Walkthroughs. **Computer Graphics**, New York, v.25, p.61-69, July 1991.

[TEL 92] TELLER, Seth J. **Visibility Computations in Densely Occluded Environments**. 1992. PhD thesis, University of California, Berkeley.

[WOL 2000] WOLTERBEEK, Lambert. **Multiplayer issues in game-engine design**. Disponível em: <http://www2.davilex.nl/devweb/pages/articles/ multiplayerarticle/text.shtml>. Acesso em: 15 de set. de 2000.

[WON 99] WONKA, Peter; SCHMALSTEIG, Dieter. Occluder Shadows for Fast Walkthroughs of Urban Environments. **Computer Graphics Forum**, Amsterdam, v.18, n.3, p.51-60, 1999.

[WON 2000] WONKA, Peter; WIMMER, Michael; SCHMALSTIEG, Dieter. Visibility Preprocessing with Occluder Fusion for Urban Walkthroughs. Rendering Techniques 2000**.** In: EUROGRAPHICS WORKSHOP ON RENDERING, 11., 2000. **Proceedings…** [S.l.: s.n.], 2000. p.71–82.

[WON 2001] WONKA, P.; WIMMER, M.; SILLION, F. X. Instant Visibility. In: EUROGRAPHICS WORKSHOP ON RENDERING, 12., 2001. **Proceedings…** [S.l.: s.n.], 2001. p.411-420.

[ZHA 97] ZHANG, Hansong; MANOCHA, Dinesh; HUDSON, Thomas; HOFF III, Kenneth E. Visibility Culling Using Hierarchical Occlusion Maps. In: SIGGRAPH, 1997. **Proceedings…** [S.l.]: Turner Whitted, 1997. p.77-88.

[ZHA 98] ZHANG, Hansong. **Effective Occlusion Culling for the Interactive Display of Arbitrary Models**. 1998. Ph.D. thesis, Department of Computer Science, UNC-Chapel Hill.