FUNDAÇÃO EDSON QUEIROZ

UNIVERSIDADE DE FORTALEZA – UNIFOR

Renderização Interativa em Dispositivos Móveis Utilizando Algoritmos de Visibilidade e Estruturas de Particionamento Espacial

Wendel Bezerra Silva

Fortaleza, CE - Brasil

2 de outubro de 2008

Wendel Bezerra Silva

Renderização Interativa em Dispositivos Móveis Utilizando Algoritmos de Visibilidade e Estruturas de Particionamento Espacial

Dissertação apresentada ao Curso de Mestrado em Informática Aplicada da Universidade de Fortaleza como requisito parcial para obtenção do Título de Mestre em Informática Aplicada.

Orientadora:
Maria Andréia Formico Rodrigues, Ph.D.

MESTRADO EM INFORMÁTICA APLICADA
UNIVERSIDADE DE FORTALEZA - UNIFOR

Fortaleza, CE - Brasil

2 de outubro de 2008

S586r Silva, Wendel Bezerra.
Renderização interativa em dispositivos móveis utilizando algoritmos de visibilidade e estruturas de particionamento espacial / Wendel Bezerra Silva. - 2008.
109 f.

Cópia de computador.
Dissertação (mestrado) – Universidade de Fortaleza, 2008.
"Orientação : Profa. Maria Andréia Formico Rodrigues, PhD."

1. Algoritmo. 2. Computação gráfica. 3. Estrutura de dados. I. Título.

CDU 681.3.06:510.5

Renderização Interativa em Dispositivos Móveis Utilizando Algoritmos de Visibilidade e Estruturas de Particionamento Espacial

Wendel Bezerra Silva

PARECER: _____

DATA: ____/____/_____

BANCA EXAMINADORA:

_____

Profª Drª Maria Andréia Formico Rodrigues
(Orientadora)
Mestrado em Informática Aplicada
Universidade de Fortaleza - UNIFOR

_____

Prof. Dr. Francisco Nivando Bezerra
(Membro da Banca)
Mestrado em Informática Aplicada
Universidade de Fortaleza - UNIFOR

_____

Profª Drª Maria Cristina Ferreira de Oliveira
(Membro da Banca)
ICMC-USP, São Carlos

Agradecimentos

Meus sinceros agradecimentos:

– Aos meus pais, por todo o apoio emocional e financeiro que me deram e aos meus irmãos, por estarem presentes na minha vida;

– À professora Andréia, pela orientação profissional e acadêmica, pelo incentivo, por ter acreditado e apostado em mim e, por toda a ajuda no desenvolvimento dessa dissertação;

– Ao Rafael Rocha pelo auxílio nos tópicos referentes à Matemática;

– A todos os professores do Projeto Fatura Imediata, que disponibilizaram os dispositivos móveis para a realização dos testes neste trabalho; e

– A todos os colegas do Mestrado em Informática Aplicada da Unifor.

It matters not how strait the gate,
How charged with punishments the scroll,
I am the master of my fate:
I am the captain of my soul.
(William Ernest Henley - Invictus).

Resumo

Este trabalho apresenta VisMobile, um sistema para renderização interativa em dispositivos móveis utilizando a API OpenGL ES (Biblioteca Gráfica para Sistemas Embarcados), o qual corresponde a uma biblioteca apropriada para desenvolvedores de aplicações gráficas 3D. Algoritmos de visibilidade e estruturas de particionamento espacial são apresentados e implementados, de modo a propiciar o processamento em tempo real de aplicações gráficas 3D em situações nas quais a câmera se locomove pelo ambiente. Diferentes combinações de algoritmos de visibilidade e estruturas de dados espaciais foram testadas no VisMobile. Variando-se os ambientes gráficos 3D, os dispositivos móveis e as plataformas de execução, diferentes combinações foram testadas sistematicamente. Os resultados mostram que taxas interativas em torno de 30q/s podem ser obtidas com sucesso utilizando-se o celular N82.

Abstract

This work presents VisMobile, an interactive rendering system for mobile devices, using the OpenGL ES API (Open Graphics Library for Embedded Systems), that corresponds to a library suitable for 3D graphical application developers. Visibility algorithms are presented and implemented to provide a real time processing of a 3D application, in situations where the camera walks through the environment. Different combinations of visibility algorithms and spatial data structures were tested on VisMobile. By varying the 3D graphical environments, the mobile devices and the execution platforms, those differents combinations were tested systematically. The results show that interactive frames per second around 30fps can be obtained with success using the mobile phone N82.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1
Introdução

Normalmente, em um ambiente tridimensional (3D) não é possível visualizar todas as superfícies de todos os objetos simultaneamente, a partir de um mesmo observador. Conseqüentemente, objetos ou partes de objetos que não estão visíveis ao observador devem ser removidos do *pipeline* de renderização. Este procedimento é importante, pois diminui o número de polígonos a serem preenchidos em uma dada cena.

Visibilidade é, portanto, um problema complexo o qual não tem uma solução ótima. Vários são os fatores que influenciam neste problema, tais como: número de *pixels* a serem pintados, número de objetos na cena, complexidade geométrica dos objetos, distribuição dos objetos, dinâmica dos objetos (ou seja, se estão em movimento ou parados), nível de realismo dos objetos (presença de texturas, transparências), etc. Estruturas de dados que representem partições recursivas do espaço são muito utilizadas em aplicações em Computação Gráfica e, particularmente, podem ser associadas a algoritmos de visibilidade na tentativa de obter resultados com melhor desempenho.

1.1 Motivação

Os desafios de otimização dos algoritmos de visibilidade para a renderização interativa de polígonos tornam-se mais complexos ainda quando a plataforma de execução é um pequeno dispositivo móvel, particularmente, telefone celular, *Personal Digital Assistant*[1] (PDA)

[1]PDA (ou *handheld*) é um computador de dimensão reduzida dotado de grande capacidade computacional, com a possibilidade de acesso a PCs, via rede sem fio. Existem duas famílias principais de PDAs no mercado atual: os PalmOne e os Pocket PCs.

ou *Smartphone*[2]. O aumento progressivo da capacidade de processamento gráfico e de armazenamento desses dispositivos móveis aponta para um novo cenário de interação em que usuários poderão fazer uso de aplicações gráficas realistas. Jogos 3D e navegação em ambientes virtuais são apenas alguns dos exemplos de aplicações gráficas que poderão se beneficiar desse novo cenário. Entretanto, apesar dos avanços tecnológicos significativos presenciados nos últimos anos, os dispositivos móveis, em sua maioria, ainda apresentam limitações importantes quando comparados aos computadores pessoais tradicionais: baixo poder de processamento; pouca memória de armazenamento; tamanho restrito e baixa resolução da tela; formas limitadas de interação com o usuário; e ausência de *hardware* gráfico (GPU). Em particular, um problema crítico enfrentado durante o desenvolvimento de aplicações gráficas 3D para dispositivos móveis é conseguir oferecer recursos de visualização e locomoção realistas, levando-se em conta os limites de memória e processamento de cada dispositivo.

O desenvolvimento de aplicações gráficas 3D para dispositivos móveis, que leve em conta as restrições discutidas anteriormente, é uma área de pesquisa recente e ainda pouco explorada por pesquisadores e desenvolvedores de *software*. Nesse sentido, há uma evidente demanda por propostas de otimizações que propiciem o uso eficiente dos diferentes recursos tecnológicos disponíveis em cada tipo de dispositivo, de tal forma a garantir a geração de implementações compactas e, ao mesmo tempo, realistas.

1.2 Objetivos

Os objetivos a serem alcançados neste trabalho, resultando em suas principais contribuições, são:

- Implementar VisMobile, um sistema para a renderização interativa de ambientes 3D em pequenos dispositivos móveis com recursos computacionais limitados, utilizando a API gráfica OpenGL ES;
- Implementar uma biblioteca para desenvolvedores de aplicações gráficas 3D;
- Implementar algoritmos de visibilidade no VisMobile;
- Realizar um estudo comparativo detalhado sobre a API gráfica OpenGL ES;

[2] *Smartphone* é um telefone celular com funcionalidades estendidas por meio de programas executados no seu sistema operacional.

- Fazer uso de estruturas de dados espaciais e combinações de algoritmos de visibilidade, de forma a produzir otimizações no desempenho do VisMobile, gerando taxas de quadros por segundo interativas;
- Definir e gerar os ambientes 3D utilizados para teste no VisMobile;
- Propor e aplicar uma metodologia para a realização dos testes baseada na locomoção da câmera (observador) pelo ambiente e na variação do nível de complexidade das cenas geradas, a qual possa ser sistematicamente aplicada;
- Analisar e comparar o desempenho das diferentes combinações de algoritmos de visibilidade e estruturas de particionamento espacial implementadas, executadas no Pocket PC iPaq hx2490b e no telefone celular Nokia N82, baseando-se nos resultados obtidos nos casos de estudo; e
- Identificar as combinações de algoritmos de visibilidade e estruturas de particionamento espacial com melhor desempenho, para cada um dos ambientes 3D, dispositivos e plataformas de execução testados.

1.3 Trabalhos Existentes

A capacidade computacional dos computadores tem aumentado, assim como a expectativa do usuário na obtenção de uma visualização de qualidade cada vez melhor, mais próxima à realidade. Em paralelo, pequenos dispositivos móveis têm se tornado parte integrante das atividades diárias da população. O poder de processamento e a capacidade de memória desses dispositivos têm aumentado consideravelmente nos últimos anos, motivando cada vez mais o desenvolvimento de aplicações 3D realistas e serviços mais especializados, direcionados para estas plataformas [7, 8].

Muitas das aplicações gráficas voltadas para a navegação, locomoção, visualização e simulação, entre outras, vêm utilizando ambientes 3D amplos, com uma quantidade massiva de dados. Assim, implementar otimizações no processo de renderização destes ambientes é importante para a geração de animações com taxas de quadros por segundo (q/s) que garantam interatividade.

Sendo assim, para a geração de aplicações gráficas que executem em tempo real, é importante o uso otimizado de recursos computacionais, estruturas de dados e de algoritmos de visualização. Otimizações se tornam ainda mais essenciais quando as plataformas de execução são pequenos dispositivos móveis, tais como: PDAs, *Smartphones*, telefones celulares, etc.

Muitas técnicas de aceleração têm sido desenvolvidas para aumentar a velocidade de renderização em ambientes gráficos complexos e compostos por dados massivos [9], dentre essas técnicas, estão os algoritmos de visibilidade [3]. Estes visam a remoção eficiente de partes não-visíveis que compõem uma cena, para que não sejam processadas pelo *pipeline* de renderização.

Dentre os algoritmos de visibilidade mais conhecidos, há os métodos que são executados em fase de pré-processamento (*offline*) e os que são executados em tempo de execução da aplicação (*online*). Os algoritmos de visibilidade também podem ser classificados quanto ao espaço no qual trabalham. Há algoritmos que trabalham no espaço do objeto, utilizando informações 3D do ambiente; e os que operam no espaço da imagem, utilizando uma representação 2D [10, 11, 12] do espaço 3D [4, 13, 14, 15, 16, 17, 2, 18].

Nas próximas seções, serão introduzidos e comparados os trabalhos existentes na área de visibilidade. Alguns destes trabalhos fazem uso de estruturas de particionamento espacial para sintetizar ambientes 3D, a um custo computacional viável, em diferentes plataformas. Sempre que pertinentes, comparações sucintas com o trabalho descrito nesta dissertação serão feitas, com o objetivo de melhor contextualizá-lo com os existentes.

1.3.1 Algoritmos de Visibilidade Baseados no Espaço do Objeto

Luebke e Georges propuseram uma estratégia de visibilidade baseada no espaço do objeto que utiliza células e portais [4]. Desenvolvido para ambientes internos, este método é baseado no espaço do objeto. Cada sala da cena é chamada de célula e as conexões entre as salas (portas e janelas) são chamadas de portais. Os autores utilizam a projeção dos portais para calcular a parte visível da cena em cada célula. Em particular, nesta dissertação, foi utilizada a estrutura *Grid* irregular no qual cada *voxel* desse *Grid* corresponde a uma célula e cada conexão de visibilidade entre os *voxels* corresponde aos portais. Ao contrário do *Grid* regular, no *Grid* irregular a cena é particionada em volumes de tamanhos diferentes. Essa estratégia é vantajosa em situações nas quais há muitos objetos aglomerados em regiões específicas do mundo a ser particionado, por exemplo, facilitando o uso da coerência espacial.

Já Coorg e Teller identificam os objetos ocultos utilizando um subconjunto de grandes objetos convexos como obstáculo [13]. Os autores utilizam coerência temporal para encontrar os objetos ocultos pelos obstáculos, quando o observador sofre alguma alteração em sua posição. Nesta dissertação, além da posição do observador, também é considerada a sua direção de visão. Adicionalmente, diferentes abordagens foram usadas para a escolha dos obstáculos. Dentre elas, foram escolhidos como obstáculos polígonos particularmente com grandes áreas

e localizados próximos ao observador. Similarmente ao trabalho de Coorg e Teller, também é utilizada a coerência temporal, porém, com uma pequena diferença: os objetos que estavam visíveis no último quadro renderizado são considerados como possível obstáculo, já que esses objetos são fortes candidatos para serem renderizados no próximo quadro.

Hudson *et al.* consideram que os objetos posicionados na sombra gerada pelos obstáculos não estão visíveis pelo observador [14]. Eles descrevem uma abordagem baseada na escolha dinâmica de um conjunto de obstáculos e no cálculo de suas sombras. Esta abordagem difere da implementada nesta dissertação, pois não utiliza métodos de particionamento espacial para fazer uso da coerência espacial.

Para gerar a remoção eficiente de partes não-visíveis de objetos, Bittner *et al.* realizam a fusão das sombras dos obstáculos e geram uma árvore de oclusão [15]. Os autores afirmam que as consultas à árvore de oclusão são rápidas o bastante (em tempo de execução), de tal forma a garantir a identificação dos objetos ocultos. Na versão corrente apresentada nesta dissertação, os obstáculos ainda não são agrupados.

1.3.2 Algoritmos de Visibilidade Baseados no Espaço da Imagem

Algoritmos que pertencem à essa classe são utilizados para identificar os objetos a serem removidos, representando o ambiente 3D por uma imagem 2D. Um exemplo bastante conhecido de uma técnica que trabalha no espaço da imagem é o algoritmo de Z-buffer [10]. Para identificar os objetos que não estão visíveis em uma cena, compara-se o valor do Z-buffer do objeto a ser testado com o do Z-buffer da cena. Serão visíveis os *pixels* do objeto que tiverem valores numéricos inferiores aos valores do Z-buffer da cena. Hoje em dia, métodos baseados no algoritmo de Z-buffer para a consulta da visibilidade de um objeto podem ser encontrados em muitos *hardwares* gráficos [11]. O processo de renderização utilizando *ray casting*[3], por exemplo, faz uso dessa técnica para determinar os objetos visíveis [12].

Inicialmente, foram introduzidos os algoritmos de remoção de superfícies ocultas (*Hidden Surface Removal* - HSR) para resolver o problema da determinação das partes visíveis de uma cena [10]. Atualmente, o Z-buffer é o método padrão de remoção de superfícies visíveis. O Z-buffer hierárquico foi introduzido como uma extensão ao método HSR [16], utilizando duas hierarquias: uma estrutura de dados espacial e uma representação hierárquica de Z-buffer. Bartz *et al.* desenvolveram um método que utiliza uma representação hierárquica de uma cena, a qual

[3] *Ray casting* é uma técnica que utiliza raios de interseção para resolver vários problemas em Computação Gráfica como por exemplo, a visualização volumétrica.

é testada para a determinação de sua parte oculta [17]. Esse método faz uso do *stencil buffer*[4] para identificar as áreas não visíveis de uma dada cena. Klosowski *et al.* criaram uma ordenação para a geometria de uma cena de acordo com as suas propriedades geométricas, gerando uma lista de objetos prioritários para serem renderizados seqüencialmente [2, 18]. Nesta dissertação, não são priorizados os objetos a serem renderizados.

1.3.3 Algoritmos de Visibilidade Baseados em Cenas Arbitrárias

Schaufler *et al.* introduziram uma técnica conservativa que utiliza um esquema discretizado do espaço e do interior dos objetos opacos para a representação de obstáculos [19]. Durang *et al.*, a partir de um observador, identificam a área visível utilizando a visibilidade volumétrica das células de visualização [20]. Tanto os obstáculos quanto os objetos a serem testados são então projetados em um plano. Os objetos serão declarados ocultos se suas projeções estiverem cobertas pela projeção cumulativa dos obstáculos. Koltun *et al.* definiram a noção de obstáculos virtuais [6]. Em uma implementação *online* 2.5D, demonstraram que um número pequeno de obstáculos virtuais é suficiente para identificar, de forma eficiente, uma área potencialmente visível da cena. Wonka *et al.* usaram a penumbra das sombras dos objetos para identificar partes ocultas do ambiente 3D [21]. Koltun *et al.* propuseram um método capaz de melhorar as técnicas de determinação da área visível de uma cena a partir de uma região [22]. Trabalhando em 2.5D, o método utiliza uma transformação de dualidade que permite a representação do problema de visibilidade em um espaço 2D, possibilitando o uso de *hardware* gráfico para acelerar os cálculos envolvidos. Nesta dissertação, é utilizada uma abordagem *online* e 3D para identificar os objetos ocultos durante a locomoção do observador em diferentes ambientes (internos e externos).

1.3.4 Tabela Comparativa

Cohen-Or *et al.* [23, 3] e Moreira *et al.* [24] realizaram estudos comparativos detalhados sobre diversos algoritmos de visibilidade. Na Tabela 1.1, os métodos mais conhecidos são comparados de acordo com as suas principais características e são organizados segundo a classificação dos métodos de visibilidade proposta por aqueles autores. Em particular, neste trabalho, essa tabela foi modificada e estendida, com referências mais recentes.

[4] *Stencil buffer* corresponde a uma matriz 2D do mesmo tamanho da janela de visualização. Utiliza-se essa matriz como máscara para o Z-buffer.

1.4 Organização da Dissertação

Esta dissertação é organizada da seguinte maneira. Primeiramente, no Capítulo 1 são apresentados os objetivos, a motivação e os trabalhos existentes na área. No Capítulo 2, é introduzida a fundamentação teórica dos algoritmos de visibilidade e das estruturas de particionamento espacial implementadas neste trabalho, bem como outros conceitos teóricos que direta ou indiretamente colaboraram para agregar valor a esta dissertação. No mesmo capítulo, é proposta uma taxonomia para os métodos de visibilidade, utilizada para melhor organizá-los segundo essa classificação. No Capítulo 3, é introduzida e detalhada a API gráfica OpenGL ES, descrevendo os seus aspectos evolutivos, as suas funcionalidades básicas e o seu *status* atual. No Capítulo 4, é descrito VisMobile, um sistema gráfico 3D para renderização interativa em dispositivos móveis utilizando algoritmos de visibilidade e estruturas de particionamento espacial como produto deste trabalho de pesquisa. No Capítulo 5, são apresentados e detalhados os testes realizados no VisMobile, bem como a metodologia definida e utilizada para a realização dos mesmos. Finalmente, no Capítulo 6, são especificadas as conclusões, bem como algumas possibilidades de trabalhos futuros.

Métodos de *occlusion culling*	**Espaço 2D/3D**	**Precisão do Método**	**Tipos de Obstáculos**	**Realiza Pré-processamento**	**Utiliza a GPU**	**Ambiente Dinâmico**
Métodos baseados em regiões que exploram estruturas de células e portais						
Airey'90 [25]	2D/3D	pode ser conservativo	portais	PVS	não	não
Teller'91 [5]	2D/3D	conservativo	portais	PVS	não	não
Métodos baseados em pontos que trabalham no espaço do objeto						
Luebke e Georges'95 [4]	3D	conservativo	portais	conectividade de células	não	atualiza a estrutura
Coorg e Teller'96 [13]	3D	conservativo	grandes, convexos e agrupados	seleção de obstáculos	não	atualiza a estrutura
Hudson *et al.*'97 [14]	3D	conservativo	grandes, convexos e não-agrupados	seleção de obstáculos	não	atualiza a estrutura
Brittner *et al.*'98 [15]	3D	conservativo	grandes e agrupados	seleção de obstáculos	não	atualiza a estrutura
Klosowski e Silva'00 [2]	3D	agressivo	todos os objetos e agrupados	volumétrico	não	atualiza a estrutura
Métodos baseados em pontos que trabalham no espaço da imagem						
Greene *et al.*'93 [26]	3D	conservativo	todos os objetos e agrupados	não	*Z-Buffer*	sim
Zhang *et al.*'97 [27]	3D	conservativo ou agressivo	grande subconjunto e agrupados	banco de dados de obstáculos	*Z-Buffer* e *Texture-Mapping*	atualiza a estrutura
Bartz *et al.*'98 [17]	3D	aproximado	todos os objetos e agrupados	não	*Buffer* secundário	atualiza a estrutura
Wonka et al.'99 [28]	2.5D	conservativo	grande subconjunto e agrupados	não	*Z-Buffer*	sim
Bernardini *et al.*'00 [29]	3D	conservativo	pré-processado e agrupados	obstáculos	não	não
Klosowski e Silva'01 [18]	3D	conservativo	todos os objetos e agrupados	volumétrico	sim	atualiza a estrutura
Métodos baseados em regiões						
Schaufler *et al.*'00 [19]	2D e 3D	conservativo	todos os objetos e agrupados	PVS	não	atualiza a estrutura
Durand *et al.*'00 [20]	3D	conservativo	grande subconjunto e agrupados	PVS	sim	atualiza a estrutura
Koltun *et al.*'00 [6]	2D e 2.5D	conservativo	grande subconjunto e agrupados	obstáculos virtuais	não	atualiza a estrutura
Wonka *et al.*'00 [21]	2D	conservativo	grande subconjunto e agrupados	PVS	sim	atualiza a estrutura
Koltun *et al.*'01 [22]	2.5D	conservativo	todos os objetos e agrupados	não	sim	atualiza a estrutura
Nirenstein *et al.*'03 [30]	3D	conservativo	todos os objetos	PVS	não	não
Moreira *et al.*'03 [24]	3D	conservativo	todos os objetos	PVS	não	atualiza a estrutura
Leyvand *et al.*'03 [31]	2.5D	conservativo	todos os objetos	PVS	sim	atualiza a estrutura
Mora *et al.*'05 [32]	3D	exato	todos os objetos	PVS	não	atualiza a estrutura
Haumont *et al.*'03 [33]	2D e 3D	conservativo	todos os objetos	não	não	não
Laine *et al.*'05 [34]	2D e 3D	agressivo e conservativo	todos os objetos	PVS	não	não

Tabela 1.1: Tabela comparativa de alguns algoritmos de visibilidade (embora não seja completa, esta tabela inclui os algoritmos mais conhecidos na literatura). PVS (*Potentially Visible Set*) identifica a área potencialmente visível a partir de um referencial. Maiores detalhes sobre os elementos utilizados na classificação serão apresentados no Capítulo 2.

Capítulo 2
Conceitos Principais: Algoritmos de Visibilidade e Particionamento Espacial

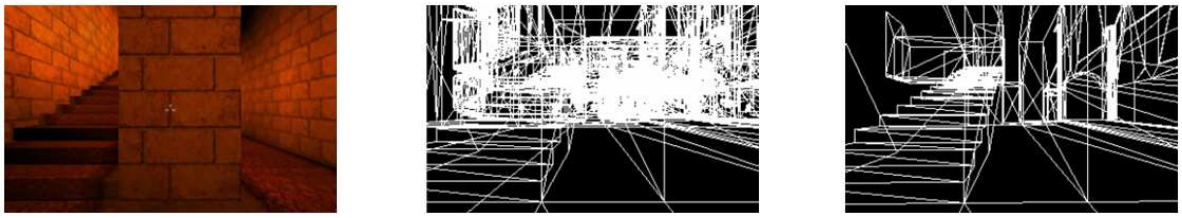
2.1 Introdução

A complexidade computacional de uma cena sintetizada por computador aumenta consideravelmente à medida que objetiva-se a geração de aplicações gráficas mais realistas, interativas e em tempo real. Isso se torna ainda mais complexo quando tais aplicações são executadas em dispositivos com recursos computacionais limitados, tais como os dispositivos móveis. Nesse caso, há a necessidade da implementação de algoritmos otimizados de visibilidade e estruturas de particionamento espacial que possibilitem o aumento da velocidade da renderização das cenas, melhorando o desempenho dessas aplicações.

Dentre as otimizações possíveis de serem implementadas em aplicações gráficas 3D, existem algoritmos que têm como finalidade reduzir a quantidade de vértices que serão enviados ao *pipeline* de renderização, diminuindo o tempo de processamento de cada quadro sintetizado. Esses algoritmos são conhecidos como algoritmos de visibilidade (Figura 2.1).

2.2 Algoritmos de Visibilidade

Os algoritmos de visibilidade podem ser divididos basicamente em 3 tipos: *view-frustum culling*, *backface culling* e *occlusion culling* [35]. Esses algoritmos podem, inclusive,



(a) (b) (c)

Figura 2.1: Em (a) observa-se uma cena de um ambiente interno renderizado. Em (b) e em (c) observa-se a mesma cena em formato *wireframe*, com e sem a aplicação de um algoritmo de visibilidade, respectivamente (imagens extraídas de [1]).

ser usados simultaneamente, de forma combinada (Figura 2.2).

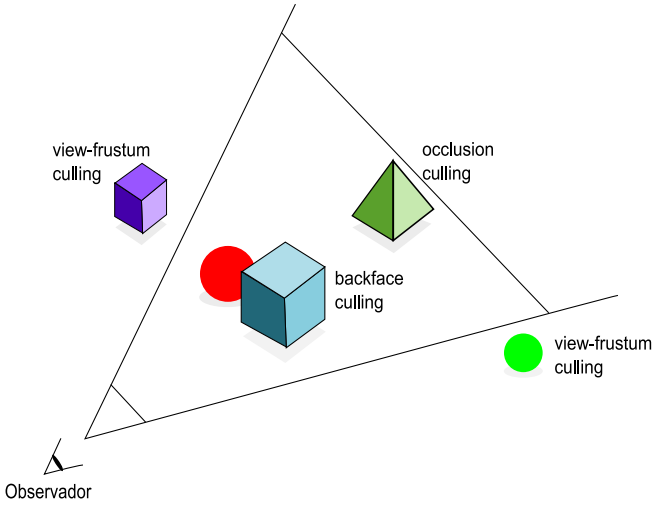


Figura 2.2: A esfera e o cubo que estão fora do volume de visualização são descartados pelo algoritmo de *view-frustum culling*, a pirâmide que está oculta pela esfera é descartada pelo *occlusion culling* e a face de trás do cubo é descartada pelo algoritmo de *backface culling*.

2.2.1 *View-frustum Culling*

O *frustum* de visualização (*view-frustum*) corresponde ao volume definido pela câmera a partir do observador, o qual define o campo de visão do observador (Figura 2.3). No modelo de projeção perspectiva, por exemplo, o *frustum* tem o formato de um tronco de pirâmide. O algoritmo de *view-frustum culling* descarta todos os objetos que estão fora da região do *frustum*, evitando que primitivas geométricas localizadas fora do volume de visualização sejam renderizadas [36].

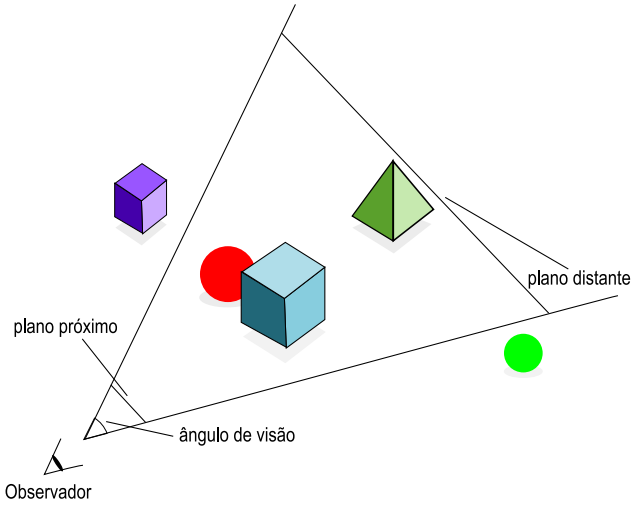


Figura 2.3: *Frustum* de visualização em uma projeção perspectiva definida pelo ângulo de visão, razão de aspecto e os planos próximo e distante.

Para identificar se um polígono está fora do *frustum* de visualização, primeiramente os 6 planos que compõem o *frustum* são calculados. Em seguida, verifica-se se há algum polígono contido nessa região. Um ponto $p_{(px,py,pz)}$ no espaço, está localizado no volume de visualização se, para todo plano do *frustum* ($Ax+By+Cz=D$), o ponto satisfaça a equação $h=\{p \in R^3 : A*px+B*py+C*pz \leq 0\}$, sendo h o semi-espaço definido pelo plano. Uma esfera (de centro c e raio r) intercepta ou está contida no volume de visualização se, para todo plano do *frustum* ($Ax+By+Cz=D$), satisfazer a equação $h=\{p \in R^3 : A*px+B*py+C*pz \leq -r\}$, sendo h o semi-espaço definido pelo plano. Apesar de bastante difundido, este teste de interseção entre a esfera e o volume de visualização pode gerar um falso positivo (Figura 2.4).

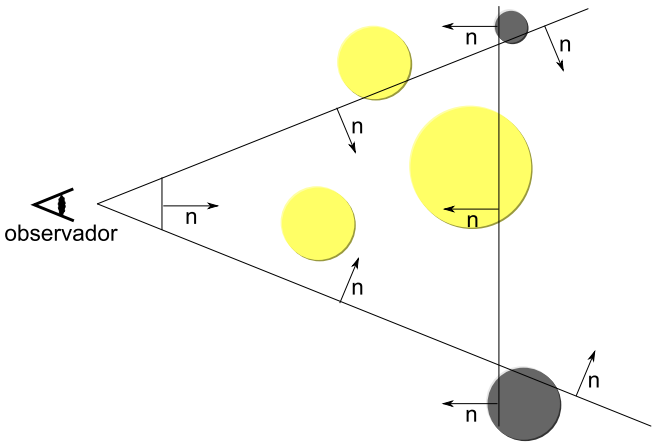


Figura 2.4: Interseção entre esferas e o *frustum* de visualização. Em particular, as duas esferas mais escuras são erroneamente identificadas como contidas no volume de visualização.

2.2.2 *Backface Culling*

O algoritmo de *backface culling* é utilizado para descartar as faces dos objetos que não estão voltadas para o observador e, portanto, que não estão visíveis. Por exemplo, as faces que estão localizadas atrás dos objetos, reduzindo assim, a quantidade de vértices que serão enviados ao *pipeline* de renderização. Esse algoritmo é bastante simples, já que essas faces podem ser facilmente identificadas utilizando-se, por exemplo, a equação $\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}||\mathbf{b}|\ cos\theta$ que corresponde ao produto escalar entre o vetor direção do observador **a** e a normal da face a ser testada **b**, sendo θ o ângulo formado entre estes dois vetores. Nesta equação, o $cos\theta$ define o sinal do produto escalar. Se o produto escalar for positivo, $-90^\circ \leq \theta \leq +90^\circ$, ou seja, a face não está virada para o observador e, consequentemente, deverá ser removida (Figura 2.5).

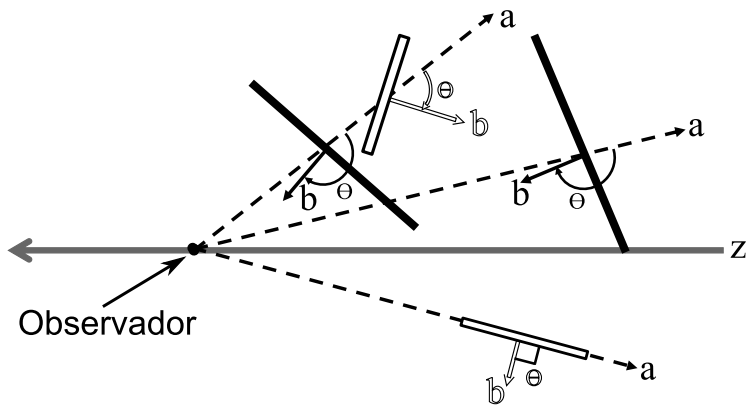


Figura 2.5: As faces escuras são renderizadas pois são visíveis pelo observador, enquanto as claras não são.

2.2.3 *Occlusion Culling*

Outro algoritmo de visibilidade existente é o utilizado para descartar objetos ocultos por outros objetos (*occlusion culling*) a partir de um observador [3]. Corresponde, na verdade, a uma família de variações de métodos bastante interessantes devido à capacidade para descartar uma grande quantidade de objetos não visíveis pelo observador, principalmente, em ambientes 3D internos e complexos. Porém, há várias abordagens nesta categoria de algoritmos, cada uma com suas especificidades. Nos próximos parágrafos, serão apresentadas as principais categorias existentes.

Quanto à taxonomia, os algoritmos de *occlusion culling* podem ser classificados da seguinte maneira [3]:

- **Ponto x Região**: a área visível é identificada pelo algoritmo de oclusão a partir de um ponto (observador) ou a partir de uma região;

- **Espaço da Imagem x do Objeto**: o espaço no qual o algoritmo executa pode utilizar uma representação 3D ou 2D da geometria do objeto;
- **Ambientes Internos x Externos x Ambos**: tipo de ambiente 3D utilizado pelo algoritmo de oclusão, particularmente: externos, ambientes internos ou fechados, ou os que apresentam ambientes externos e internos simultaneamente;
- **Obstáculos Individuais x Agrupados**: obstáculos podem ser tratados pelo algoritmo de oclusão de tal forma a agrupá-los, com o objetivo de gerar um único obstáculo, maior que os anteriores; e
- **Cenas Estáticas x Dinâmicas**: os objetos geométricos podem compor cenas nas quais inexiste movimento ou podem compor cenas nas quais existe movimento.

Os algoritmos de *occlusion culling* por ponto, também conhecidos como *from-point* ou *viewpoint*, determinam a área visível a partir de um observador [4, 13, 15]. A principal vantagem destes algoritmos é o baixo custo computacional, pois podem ser executados antes da renderização de cada quadro. Ou seja, estes algoritmos são indicados para o tratamento de ambientes que possuem objetos dinâmicos.

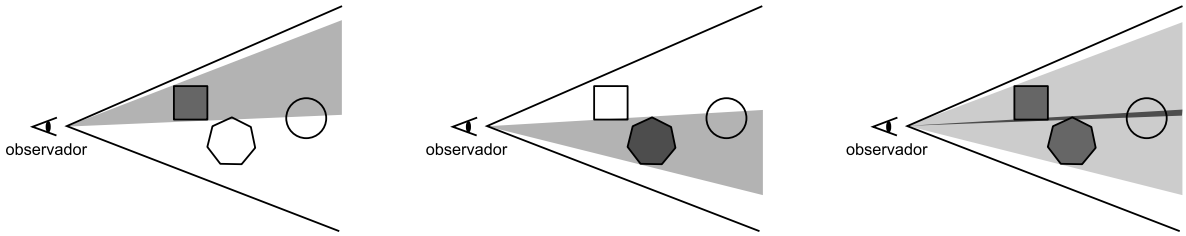
Já os algoritmos que definem a área visível a partir de regiões são conhecidos como *from-region* ou *viewcell* [37]. Essas regiões são usualmente chamadas de células. Os algoritmos mais conhecidos que utilizam essa abordagem são os algoritmos de células e portais [4]. A visibilidade é calculada para uma determinada região e enquanto o observador estiver posicionado naquela determinada área não será necessário identificar novamente a área visível. A principal vantagem desses algoritmos é também a diminuição do custo computacional, pois a identificação da parte visível da cena para cada região pode ser calculada em uma fase de pré-processamento. Os algoritmos do tipo *from-region* são muito utilizados em ambientes com objetos estáticos, ou em conjunto com outro algoritmo de oclusão para tratar ambientes com objetos dinâmicos [5].

Algumas implementações de algoritmos de visibilidade tratam o ambiente utilizando suas informações 3D [4, 22]. Ou seja, trabalham no espaço do objeto. Porém, existem formas mais simplificadas para a representação de ambientes 3D, como por exemplo, utilizando imagens 2D(representação da cena rasterizada). As abordagens que utilizam imagens 2D consideram somente o espaço da imagem [26, 17].

Existem ainda soluções que utilizam características específicas do tipo de ambiente, como por exemplo, o método definido por Luebke *et al.* [4]. Os autores introduziram a idéia de

células e portais, uma abordagem específica para ambientes internos. Há também soluções para ambientes externos, como a solução proposta por Wonka *et al.* para resolução de problemas de oclusão em tempo real para ambientes urbanos [21].

A escolha dos obstáculos no método de *occlusion culling* pode definir o desempenho do algoritmo. Dentre as soluções propostas mais conhecidas, há aquelas que utilizam os obstáculos individualmente e as que agrupam os obstáculos [21]. Alguns autores optam por agrupar obstáculos tentando diminuir o número de testes de visibilidade, de tal forma a obter um resultado mais exato (às vezes, é necessário o uso de mais de um obstáculo para garantir que um objeto fique totalmente oculto), como mostra a Figura 2.6.



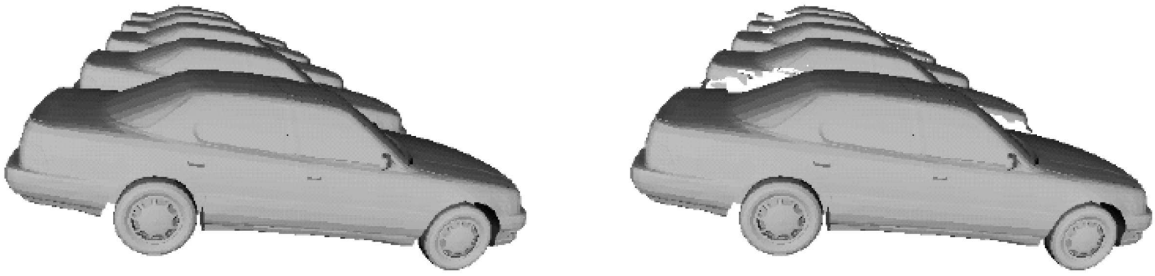
(a) (b) (c)

Figura 2.6: Em (a) e (b) imagens ilustrativas de situações nas quais os obstáculos individualmente não ocultam um objeto por completo. Porém em (c), pode-se observar que os obstáculos agrupados ocultam o objeto.

Os algoritmos de *occlusion culling* também podem ser classificados como os desenvolvidos para ambientes estáticos e os para ambientes dinâmicos [29, 28]. Os algoritmos voltados para ambientes estáticos podem não executar em ambientes dinâmicos quando pré-processam informações sobre o ambiente. No caso dos ambientes dinâmicos, esses cálculos teriam que ser realizados em tempo de execução. Algumas soluções, por exemplo, identificam a área visível para cada região em uma fase de pré-processamento [30].

Ainda, quanto à precisão da área visível identificada, os algoritmos de *occlusion culling* podem ser classificados da seguinte forma: conservativos, agressivos, aproximativos e exatos [35]. Os algoritmos conservativos superestimam a área visível, podendo selecionar objetos que não estão realmente visíveis, além de todos os objetos visíveis. Podem executar rapidamente, porém, polígonos desnecessários são enviados ao *pipeline* de renderização. Já os algoritmos agressivos subestimam a área visível e, com isso, alguns objetos visíveis não são selecionados. Também podem executar rapidamente, porém, há a possibilidade de apresentarem como resultado final uma cena incompleta, com ausência de certos objetos que deveriam estar visíveis (Figura 2.7). Os algoritmos aproximativos selecionam a área visível da cena de forma aproximada, contudo, podem selecionar objetos que não estão visíveis na cena e/ou não selecionar objetos que deveriam ser selecionados. Executam rapidamente, porém, alguns polígonos

podem ser desnecessariamente enviados ao *pipeline* de renderização gerando redundância, enquanto outros podem ser erroneamente descartados, gerando uma cena incompleta. Finalmente, os algoritmos exatos selecionam somente os objetos visíveis na cena. São métodos não-triviais de serem implementados e bastante custosos computacionalmente. Enviam ao *pipeline* de renderização apenas os objetos visíveis.



(a) (b)

Figura 2.7: Em (a), uma cena renderizada utilizando um algoritmo conservativo e, em (b), a mesma cena renderizada utilizando um algoritmo agressivo (imagem retirada de [2]).

2.2.4 Outros Algoritmos

Existem ainda algoritmos que trabalham em conjunto com os métodos de visibilidade para determinar a parte visível de uma cena. Dentre eles, podem ser citados: Células e Portais, *Potentially Visible Set* (PVS) ou Conjunto Potencialmente Visível, *Aspect Graph* ou Grafo de Aspecto, Obstáculos Virtuais e Projeções Estendidas. Estes métodos serão detalhados nas subseções seguintes.

Células e Portais

Células e portais têm como idéia principal dividir a cena 3D em regiões (células), identificando a conectividade (portais) entre essas regiões e representando-as em um grafo. Neste grafo, as células correspondem aos nós e os portais correspondem às conexões. A partir desse grafo inicial é criado um grafo de visibilidade. Percorre-se o grafo inicial e para cada célula, identificam-se as outras regiões visíveis, a partir da região de origem (Figura 2.8).

Células e portais são principalmente utilizados em ambientes internos, como mostra a Figura 2.9. Nesse exemplo, cada quarto é identificado como uma célula, sendo as janelas e as portas definidas como portais.

No algoritmo proposto por Teller *et al.*, o ambiente interno é subdividido em células,

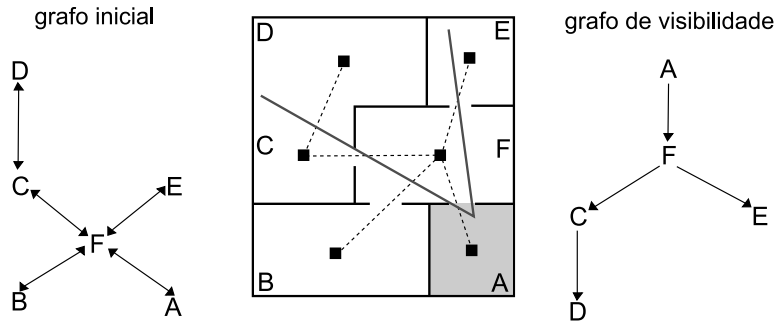


Figura 2.8: Representação de uma cena por grafos. À esquerda, o grafo de células e portais e à direita, o grafo de visibilidade entre as células. Pode ser observado que, como a região B não está visível pelo *frustum*, seu respectivo nó não está presente no grafo de visibilidade (imagem retirada de [3]).

Figura 2.9: Um ambiente interno típico utilizando células e portais. À esquerda, podem ser observados em destaque os portais (janelas e portas) e, à direita, as áreas visíveis a partir de um observador (imagem retirada de [4]).

as quais não correspondem, obrigatoriamente, cada uma a uma sala do ambiente [5]. Esse algoritmo apresenta duas etapas: a visibilidade célula-a-célula e a visibilidade observador-a-célula. Na primeira etapa, são identificadas as células visíveis e, na segunda etapa, as partes visíveis do ambiente a partir de um referencial (Figura 2.10). As linhas brancas definem as células. A região em cinza escuro corresponde à região onde o observador está localizado, em cinza as células selecionadas pelo algoritmo de visibilidade célula-a-célula e em preto as células selecionadas após a aplicação do algoritmo observador-a-célula.

Conjunto Potencialmente Visível

A primeira implementação de um Conjunto Potencialmente Visível ou *Potentially Visible Set* (PVS) foi descrita por Airey *et al.* para determinar o conjunto de objetos geométricos potencialmente visíveis [25]. O PVS identifica a área visível a partir de um referencial. Este pode ser calculado em fase de pré-processamento ou em tempo de execução. Conhecendo o PVS, rapidamente descarta-se a geometria do ambiente que não está visível. Algumas vezes,

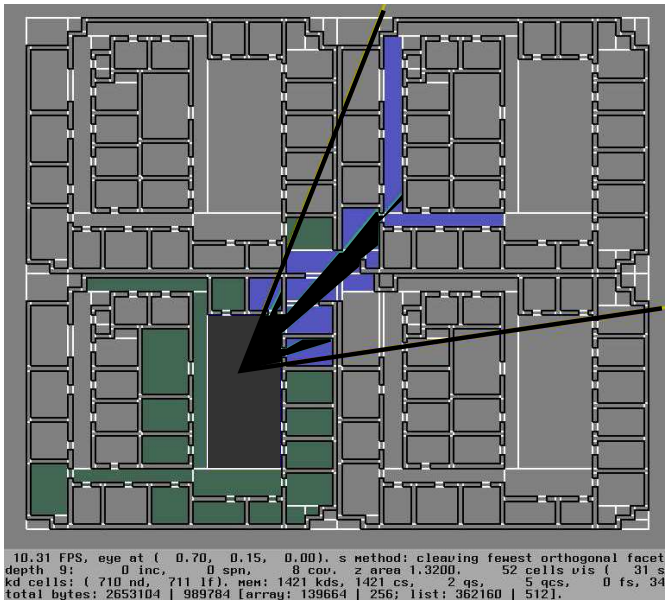


Figura 2.10: Áreas visíveis a partir de um observador (extraída de [5]).

os algoritmos de *occlusion culling* por região são denominados de PVS. Porém, o termo PVS está difundido e vem sendo utilizado também para identificar um tipo específico de *occlusion culling* e não apenas o *occlusion culling* por região [38].

Grafo de Aspecto

Grafo de Aspecto (ou *Aspect Graph*) corresponde à uma representação de objetos 3D que utiliza um conjunto de visualizações 3D, também conhecidas como Partição Espacial por Portas de Visão [37]. De forma mais genérica, são estruturas de dados que incorporam informações sobre a visualização de um objeto ou sobre uma coleção de objetos em uma dada cena. Este tipo de estrutura é utilizada para classificar os objetos de acordo com os seus aspectos visuais para serem utilizados, por exemplo, em uma pesquisa por objetos semelhantes [39]. Teller demostrou como utilizar uma estrutura similar ao grafo de aspecto para identificar a parte oculta de um ambiente 3D através de um conjunto de portais [40].

Obstáculos Virtuais

Como já mencionado anteriormente, os métodos de *occlusion culling* são utilizados para remover os objetos que estão ocultos a partir de um observador [3]. Os obstáculos (nesse caso, objetos que estão impedindo outros objetos de serem visualizados) podem ter diferentes formas e tamanhos. Em alguns casos, um obstáculo sozinho pode não ser capaz de ocultar um determinado objeto, porém, quando associado a outros obstáculos, pode ocultá-lo. Ou seja,

com os obstáculos virtuais é possível agrupar obstáculos, obtendo-se um obstáculo de tamanho maior. Esta estratégia possibilita a diminuição do tempo de processamento necessário para a determinação da área oculta, pois diminui o número dos possíveis obstáculos a serem testados. Existem na literatura vários métodos que implementam obstáculos virtuais, trabalhando em 2.5D [6, 21] e 3D [19].

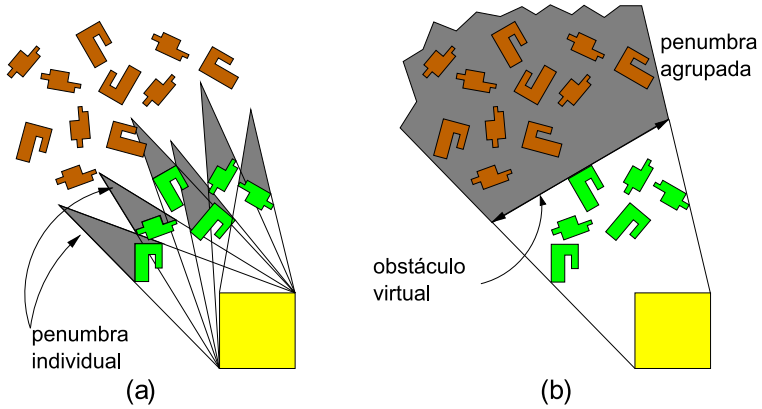


Figura 2.11: Obstáculos virtuais gerados para uma determinada região (cubo). São exibidas as áreas ocultas (penumbras), as geradas pelos obstáculos separados e as regiões ocultas pelo obstáculo virtual (extraído de [6]).

Projeções Estendidas

Durand *et al.* apresentaram uma abordagem para identificar os objetos ocultos, utilizando a visibilidade volumétrica de uma determinada célula na fase de pré-processamento do PVS [20]. O método divide os objetos em duas categorias (obstáculos e ocultos). Em seguida, os obstáculos e os objetos ocultos são projetados em um plano. Os objetos são considerados ocultos caso estejam completamente cobertos pela área comum das projeções dos obstáculos. Para garantir que são conservativos, a projeção estendida de um obstáculo subestima a projeção de um ponto da região de visualização, enquanto que a projeção do objeto é superestimada.

Além dos algoritmos de visibilidade, estruturas de particionamento espacial são utilizadas para obter um melhor desempenho da aplicação gráfica. A seguir, serão detalhadas algumas das estruturas de dados espaciais mais conhecidas na literatura.

2.3 Estruturas de Particionamento Espacial

Algoritmos de Computação Gráfica que fazem uso de dados espaciais podem ser otimizados aplicando-se técnicas de particionamento espacial, de tal forma a subdividir um ambiente 3D em pequenas sub-regiões, as quais são armazenadas nas estruturas de dados espaciais.

Utilizando-se estruturas de particionamento espacial, certos algoritmos podem fazer uso da coerência espacial para selecionar partes específicas do ambiente. Alguns exemplos de estruturas de dados espaciais são: *Grids* Irregulares, *Octrees*, Portal *Octrees* e *BSP-Trees*.

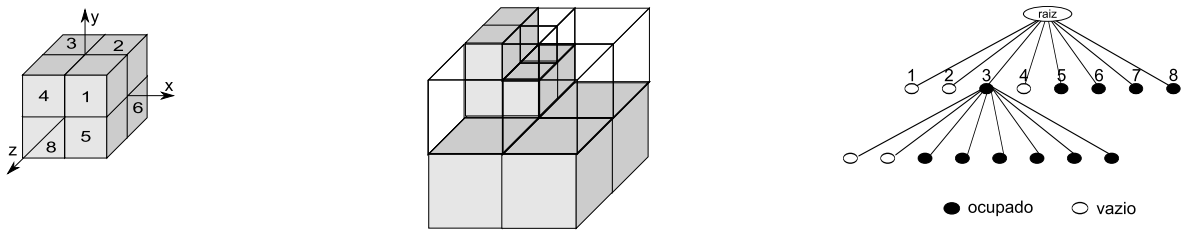
2.3.1 *Grid* Irregular

O *Grid* uniforme é uma partição espacial básica. O espaço 3D, considerado como sendo um cubo, é particionado em $N \times N \times N$ células cúbicas de mesmo volume. Cada *voxel* contém uma lista de objetos que ocupam essa determinada região. Cada objeto 3D da cena pode ocupar uma ou mais células do *Grid*. Quando se particiona um ambiente utilizando *Grids* é necessário levar em consideração o número de divisões N. Este valor é importante pois define o desempenho da busca espacial na estrutura.

Ao contrário do *Grid* uniforme, o *Grid* Irregular particiona o espaço em um número distinto de divisões para cada eixo de coordenadas ($N_x \times N_y \times N_z$). Assim como o *Grid* uniforme, o desempenho do *Grid* Irregular depende da quantidade e do tamanho de suas partições.

2.3.2 *Octree*

A estrutura *Octree* representa objetos sólidos 3D de forma hierárquica, subdividindo recursivamente o espaço [41]. O volume ocupado pelos objetos no cenário é particionado a cada nível da hierarquia, de tal forma a dividir o volume original em 8 subvolumes cúbicos alinhados aos eixos ou octantes (Figura 2.12). Na *Octree*, cada nó representa um subvolume cúbico, que pode ser marcado como vazio ou ocupado (Figura 2.12(c)). A estrutura resultante da subdivisão recursiva é uma árvore de grau 8, na qual cada nó não-folha tem 8 filhos. Existe também uma representação alternativa na qual esferas são usadas como volumes octantes [42].



(a) (b) (c)

Figura 2.12: Em (a), numeração dos octantes; em (b) uma partição em uma *Octree* e, em (c), a estrutura hierárquica da *Octree* em (b).

Se o objeto está parcialmente contido no octante, este é decomposto em oito sub-octantes onde, para cada um dos sub-octantes, verificam-se os objetos completamente contidos

nestes sub-octantes. Esse procedimento recursivo de construção da *Octree* é executado até que nenhum nó apresente objetos parcialmente contidos ou até que a estrutura da *Octree* alcance um determinado nível de profundidade limite, definido na implementação. Nesse último caso, os octantes que possuem objetos parcialmente contidos são aproximados como ocupados ou como não ocupados, usando algum critério definido préviamente.

O uso de *Octrees* propicia o descarte rápido de partes do espaço, de tal forma a identificar a área ocupada por um nó da árvore que não satisfaz às necessidades da busca, conseqüentemente, que seus nós filhos também não satisfazem. Entretanto, o uso de *Octree* apresenta algumas limitações. A posição dos octantes é restrita ao alinhamento dos eixos de coordenadas e sua dimensão tem tamanho fixo.

2.3.3 Portal *Octree*

Portal *Octree* é uma estrutura híbrida que combina a partição do ambiente em células e estruturas do tipo *Octree*. Inicialmente, o ambiente 3D é particionado em regiões, e cada região é subdividida utilizando-se *Octrees* (Figura 2.13). A partição inicial em células se assemelha à abordagem de portais dos métodos de visibilidade (as regiões representam as células e as conexões entre essas regiões, os portais).

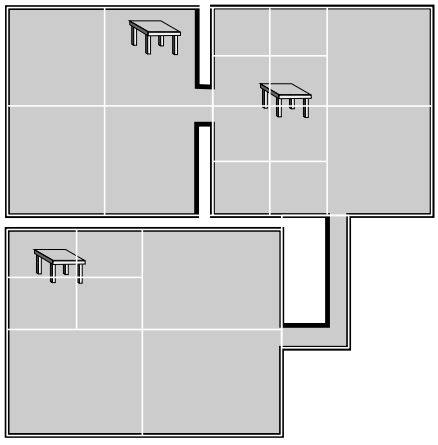


Figura 2.13: Visão bidimensional de um ambiente particionado em Portal *Octree*. Cada sala é identificada como célula e particionada utilizando-se *Octree*.

Uma vantagem é a possibilidade de utilização da técnica de células e portais em conjunto com a Portal *Octree*. Além disso, esta estrutura oferece um maior nível de detalhamento do ambiente, em comparação à técnica de portal. Para evitar a alocação redundante de objetos, a *Octree* pode armazenar apenas as referências a estes objetos. Esta estrutura também pode ser utilizada em ambientes dinâmicos, necessitando, porém, que a *Octree* seja atualizada.

2.3.4 *BSP-Tree*

BSP-Tree (*Binary Space Partitioning*) é um tipo de árvore hierárquica, binária, proposta inicialmente para resolver o problema de determinação de superfícies visíveis. Representa recursivamente a subdivisão de um universo de dimensão n, usando hiperplanos de dimensão $n-1$ [43]. O particionamento do espaço é definido por hiperplanos os quais não necessitam ser alinhados aos eixos de coordenadas. Essas divisões do espaço são realizadas recursivamente, até que as subregiões alcancem um determinado limiar τ definido na implementação.

Em um espaço 3D, o hiperplano de partição h é definido pela equação do plano $ax+by+cz=d$. Este hiperplano particiona o mundo R^3 em dois semi-espaços. Os pontos do espaço definidos por $h^+ = \{\mathbf{p} \in R^3 : ax+by+cz > d\}$ fazem parte da sub-região positiva da partição (ou frente), enquanto que os pontos da sub-região negativa h^- satisfazem a condição representada como $h^- = \{\mathbf{p} \in R^3 : ax+by+cz < d\}$.

A raiz da BSP-Tree (nó t_0) corresponde a toda região do ambiente 3D, $r(t_0) = R^3$. Na realidade, a região coberta pelo nó raiz é ilimitada mas, na prática, é considerada apenas a região definida pelos limites do modelo 3D. Para essa região, é escolhido um hiperplano de partição h_0, dividindo o espaço em 2: sub-espaço negativo h_0^- e o positivo h_0^+. Sendo o sub-espaço negativo definido por $r^-(t_1) = r(t_0) \cap h_0^-$ e o sub-espaço positivo, por $r^+(t_1) = r(t_0) \cap h_0^+$. Estes sub-espaços são armazenados na estrutura hierárquica como $(t_0, esquerda)$ e $(t_0, direita)$, filhos do nó t_0 da árvore.

Esse procedimento de divisão do espaço pode ser repetido recursivamente para cada uma dessas sub-regiões construindo, assim, a árvore binária. A construção da estrutura inicia usando-se como região todo o universo do ambiente 3D. Cada nó t_n da arvore binária armazena o hiperplano h_{t_n}, responsável por particionar o espaço em 2: o nó da esquerda representa o semi-espaço negativo e o da direita representa o semi-espaço positivo. A região de um nó $r(t)$ é definida pela interseção dos semi-espaços determinado pela associação dos hiperplanos dos nós encontrados no caminho percorrido na árvore, até o nó t. Isto é, dado um nó t_i em uma profundidade i, seu caminho na árvore é definido por $\{t_0, \ldots, t_i\}$. Então, se t_i é o nó filho esquerdo de t_{i-1}, por exemplo, sua região é definida como $r(t_i) = r(t_{i-1}) \cap h_{t_{i-1}}^-$ (Figura 2.14). Os nós são particionados recursivamente até que não se encontre mais nenhum hiperplano que consiga obter uma determinada relação de partição τ, se tornando nó folha. Essa relação de partição τ é definida na implementação na qual, usualmente, utiliza-se uma razão de proporção entre os objetos presentes em ambos os lados da partição, bem como o número mínimo de objetos que cada sub-região pode conter.

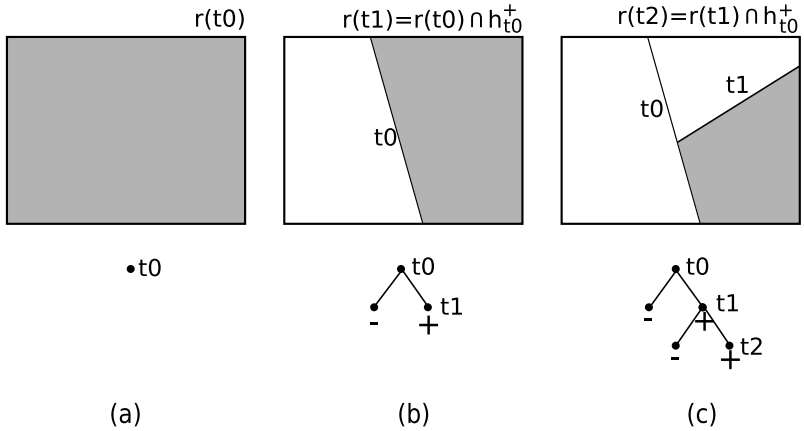


Figura 2.14: Ambiente particionado usando a estrutura espacial BSP-Tree com sua respectiva representação hierárquica. Em (a), (b) e (c), observam-se três regiões destacadas, assim como as equações para a determinação dessas regiões.

BSP-Trees podem ser utilizadas para resolver vários tipos de problemas em Computação Gráfica, tais como: iluminação global, geração de sombras, *ray casting* e visibilidade [44]. São também utilizadas para representar modelos sólidos e para obtenção do menor caminho em um ambiente 3D. Uma das vantagens da BSP-Tree é a possibilidade de realização de pesquisa espacial utilizando-se simples comparações envolvendo o lado do hiperplano associado com os nós da estrutura. Entretanto, as maiores desvantagens da BSP-Tree são: limitação de sua utilização em ambientes dinâmicos, uma vez que, neste caso, a estrutura necessita ser reconstruída; e impossibilidade de evitar que os hiperplanos cortem os objetos da cena. Além disso, a BSP-Tree pode se tornar bastante profunda e, dependendo dos hiperplanos escolhidos, pode gerar uma árvore desbalanceada.

2.3.5 Tabela Comparativa

Na Tabela 2.1, as principais vantagens e desvantagens das estruturas de dados espaciais apresentadas neste trabalho são comparadas de forma sucinta.

Estrutura	**Vantagens e Desvantagens**
Grid Irregular	Algoritmo simples
	Escolhe-se o número de partições em cada eixo de coordenadas
	Não é adaptativo ao modelo 3D
	Número de partições influencia no desempenho
Octree	Estrutura hierárquica
	Adaptativo ao modelo 3D
	Posição e tamanho dos octantes são fixos
	Necessita de uma grande quantidade de memória
Portal *Octree*	Particiona o ambiente 3D em células e portais
	Adaptativo ao modelo 3D
	Adaptativo a ambientes dinâmicos
BSP-Tree	Estrutura hierárquica
	Algoritmo de pesquisa 3D simples
	Adaptativo ao modelo 3D
	Número de primitivas resultante é maior que o número de primitivas existentes no ambiente original
	Em geral é desbalanceada
	Não se adapta a ambientes dinâmicos

Tabela 2.1: Tabela comparativa de algumas estruturas de dados espaciais.

No próximo capítulo, a API gráfica OpenGL ES será introduzida e detalhada, descrevendo os seus aspectos evolutivos, as suas funcionalidades básicas e o seu *status* atual.

Capítulo 3
OpenGL ES

OpenGL ES (OpenGL para Sistemas Embarcados, ou *Embedded Systems*) é uma especificação livre de uma API gráfica, de baixo nível, portável, que permite a utilização de funções 2D e 3D em sistemas embarcados, incluindo consoles, telefones, PDAs, computadores portáteis, entre outros [45]. Consiste em um subconjunto bem definido da OpenGL [46], uma API voltada para computadores pessoais, de grande utilização no desenvolvimento de aplicações em Computação Gráfica. Apresenta uma interface de baixo-nível flexível e robusta localizada entre o *software* e o acelerador gráfico.

A API OpenGL ES faz acesso direto ao sistema operacional e/ou ao *hardware* gráfico, caso esteja disponível. Aplicações nativas podem ser realizadas para acessar essa API, de modo a usufruir de suas funcionalidades gráficas. OpenGL ES foi especificada de forma compacta e otimizada, influenciando no surgimento de implementações para vários modelos de dispositivos móveis, com diferentes restrições de recursos computacionais. Desta forma, aplicações nativas tornaram-se portáveis, necessitando apenas de uma implementação da API executando em um determinado modelo de dispositivo.

Nas próximas seções, serão detalhados os aspectos evolutivos, características básicas e o *status* atual da API OpenGL ES, bem como as suas funcionalidades [47, 48], haja visto que esta foi estudada e utilizada no desenvolvimento deste trabalho.

3.1 Aspectos Evolutivos e *Status* Atual

Os primeiros motores gráficos para dispositivos portáteis foram disponibilizados em 2000 e 2001 [49] e, durante alguns anos, somente os motores gráficos proprietários dominaram o mercado. Contudo, novos motores gráficos e APIs padrões surgiram em 2004 e começaram a substituir os motores gráficos proprietários. Com base em algumas destas APIs, já existem atualmente sistemas gráficos interativos voltados para plataformas móveis, tais como PDAs, *Smartphones* e telefones celulares.

Devido à existência de uma grande quantidade de bibliotecas gráficas (muitas delas, diferentes para cada modelo de dispositivo móvel, mesmo sendo de um mesmo fabricante), surgiu uma necessidade natural de padronização. Ou seja, da criação de uma especificação de uma API gráfica comum para dispositivos móveis e sistemas embarcados. Em 2003, a Khronos Group, um consórcio de empresas do qual fazem parte a Intel, Nokia, NVidia, entre outras, lançou a primeira versão da especificação da API OpenGL ES [45]. Baseada na versão 1.3 da OpenGL [50], a versão 1.0 teve como principal meta a compactação da OpenGL para a utilização em dispositivos com recursos computacionais restritos, com suporte em *hardware* apenas para ponto fixo. Todos os tipos de pontos flutuantes de precisão dupla, disponíveis na OpenGL, foram então substituídos por pontos flutuantes de precisão simples na OpenGL ES. Além disso, para tornar a OpenGL ES ainda mais compacta, muitas das operações 2D que podiam ser emuladas via funcionalidades 3D foram removidas.

Em 2004, a Khronos Group lançou a especificação da OpenGL ES 1.1, baseada na especificação da OpenGL 1.5. Ao contrário da versão 1.0, que visava essencialmente a criação de uma API compacta, a versão 1.1 foi desenvolvida com o objetivo de operar em *hardware* especializado, produzir imagens com melhor qualidade e realizar otimizações em funcionalidades existentes na versão anterior (reduzindo o número de acessos à memória, etc). Em 2005, complementando a especificação da versão 1.1, foi lançado o Pacote de Extensão da OpenGL ES 1.1, com novas funcionalidades que serão incluídas na especificação da OpenGL ES 1.2.

Idealizando a programação de um *pipeline* gráfico que possibilitasse a criação de novos modelos de tonalizadores e objetos programáveis, no mesmo ano de 2005, a OpenGL ES 2.0 foi lançada, baseada na especificação da OpenGL 2.0. Ainda em 2005, também foi lançada a API OpenGL ES-SC 1.0 (*Safety Critical*) [45]. A API OpenGL ES-SC foi definida relativa à OpenGL 1.3 (assim como a versão 1.0). Porém, foi estruturada para suprir a necessidade de execução em ambientes críticos, onde erros não são toleráveis, por exemplo, nos *softwares* embarcados (disponíveis nas áreas de aviação, industrial, militar e indústria automobilística,

entre outras).

A especificação padrão da OpenGL ES 1.1 motivou o aparecimento de várias implementações, entre as quais, Rasteroid [51] e Vincent [52]. Rasteroid é uma implementação comercial feita pela Hybrid, voltada para *Pocket PCs*, *Smartphones* e BREW. Atualmente, a Rasteroid foi comprada pela NVidia. PowerVR apresenta implementações para diversos dispositivos, desde celulares até *Pocket PCs*. Já Vincent 3D *Mobile Library*, é uma API gratuita, de código-fonte aberto, voltada para *Pocket PCs* e *Smartphones*. Atualmente, o futuro da Vincent é incerto. Algumas plataformas apresentam implementação nativa da API OpenGL ES disponibilizadas pelo próprio fabricante, como por exemplo, a API presente para Nokia Série 60.

3.2 Características Básicas

Existem dois métodos principais na OpenGL ES, o *init()* e o *display()*. O primeiro método é executado uma única vez no início da aplicação. Normalmente é utilizado para a inicialização do ambiente gráfico e dos objetos da cena. O segundo é executado antes da renderização de cada quadro e é utilizado para realizar a interação com a aplicação. Na API OpenGL, pode-se especificar um objeto através de um vetor de vértices ou pode-se utilizar um bloco *begin-end* para a definição de um objeto e de seus atributos de visualização. Em particular, essa última forma de representação foi removida na OpenGL ES, mantendo-se apenas a especificação por um vetor de vértices. Para a representação de um objeto, a API OpenGL ES apresenta as seguintes primitivas geométricas: ponto, linha e as primitivas baseadas em triângulos (quadriláteros e polígonos foram eliminados da especificação, podendo ser emulados utilizando-se triângulos). Para cada vértice do objeto, um vetor de vértices é utilizado para a definição das cores RGB e transparência (*alpha*).

Na OpenGL ES, as operações com matrizes (de modelagem, que posicionam e orientam os objetos na cena; de visualização, que determinam o posicionamento da câmera; e de projeção, que determinam o volume de visualização) são realizadas através de duas pilhas: uma para a manipulação das matrizes de modelagem e de visualização (*modelview*) e outra para a manipulação das matrizes de projeção (*projection*). As transformações são acumulativas, isto é, podem ser aplicadas umas sobre as outras.

Texturas 2D podem ser utilizadas, mas multitexturas somente estão disponíveis a partir da versão 1.1. Dentre outras funcionalidades gráficas disponíveis na API podem ser desta-

cados o *vertex skinning*[1] para animação, planos de recortes definidos pelo usuário, geração automática de texturas de *mipmap*, filtro de texturas para aumentar a qualidade das imagens, *point sprites*, efeitos especiais do tipo *fog* e programação em *hardware* (tonalizador de vértices ou *vertex shader* e tonalizador de *pixels*) [53]. *Mipmap* é uma coleção de imagens otimizadas, acompanhadas de uma textura principal, utilizada para aumentar a velocidade da renderização. *Sprites* são objetos gráficos 2D que se movem sobre uma tela, são compostos por uma única imagem ou por uma sequência de imagens, que definem a animação de um determinado elemento [54]. *Point sprites* são *sprites* representados por apenas um ponto no espaço, ao invés de um polígono. Tonalizador de vértices ou de *pixels* corresponde a um programa para tonalização, normalmente executado em nível de *hardware*.

3.2.1 *Pipeline* de Renderização

O *pipeline* de renderização da OpenGL ES 1.0 e 1.1 (Figura 3.1) não pode ser modificado, ou seja, as funcionalidades presentes nos estágios do *pipeline* não podem ser alteradas, embora algumas possam ser parametrizadas (por exemplo, o *blending* ou mistura de cores) ou desabilitadas (neste caso, os dados passam por esse estágio sem sofrerem alteração). Funcionalidades pertencentes ao início do *pipeline* podem ser emuladas no código da aplicação (modos de seleção, listas de exibição, geração automática de coordenadas de texturas, planos de recorte definidos pelo usuário, etc). Já funcionalidades do final do *pipeline* são mais difíceis de serem emuladas, especialmente em sistemas nos quais o *hardware* provê suporte à rasterização, não permitindo o acesso direto ao *frame buffer*.

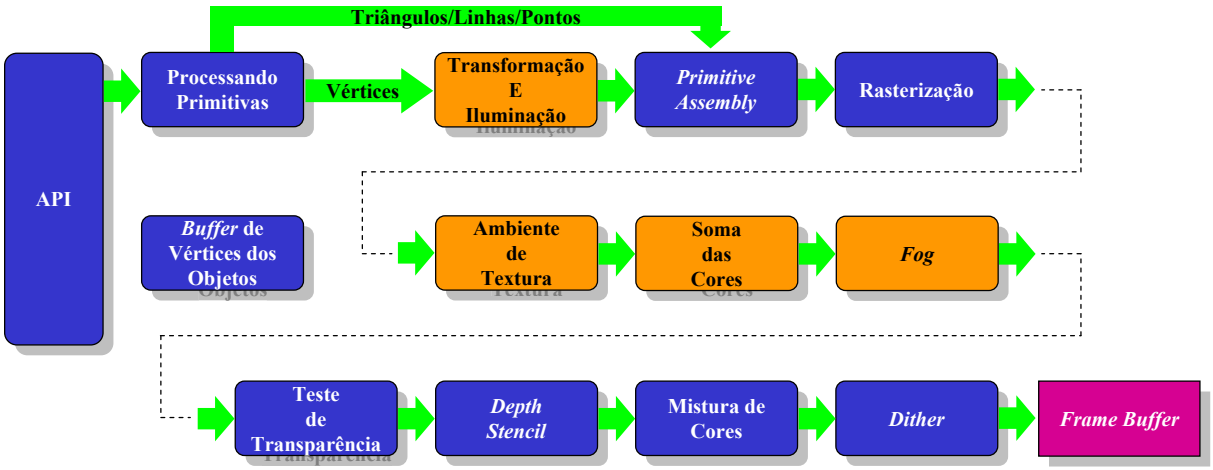


Figura 3.1: *Pipeline* da OpenGL ES 1.x.

[1] *Vertex skinning* trata o problema da junção das articulações de objetos articulados em uma animação, gerando um personagem mais realista.

O *pipeline* de renderização da OpenGL ES 1.0 inicia-se com a definição de um vetor de vértices das posições dos objetos, normais, coordenadas de textura, cores e propriedades dos materiais. Os vértices são transformados utilizando-se o modelo de visualização e projeção de matrizes e tonalizados pelo modelo de iluminação simples (*Flat Shading*) ou *smooth* (*Gouraud Shading*) [49]. Os vértices podem ser utilizados para compor primitivas geométricas (pontos, linhas, e triângulos) as quais, por sua vez, são rasterizadas. Além disso, pode-se combinar a cor das próprias primitivas, com texturas a elas associadas (caso a primitiva geométrica suporte a textura aplicada). Testes envolvendo *buffers* de profundidade (*depth buffer*), de cor (*alpha*), *stencil buffer*, etc, também podem ser utilizados para a determinação da cor final de um *pixel*.

OpenGL oferece dois modos para a definição das coordenadas de textura de um objeto: simples e multitextura. Comparando-se com a OpenGL, a implementação da OpenGL ES manteve apenas a versão para multitextura. A versão 1.1 teve como alvo sistemas com um número maior de recursos computacionais, incluindo *hardware* gráfico especializado. Oferece um melhor suporte ao mapeamento de texturas, através do uso de multitexturas. Disponibiliza, ainda, o mapeamento de texturas por ponto, permitindo a renderização consistente de objetos com poucos polígonos através de *point sprites*. *Point sprites* permitem a definição de um mapeamento de textura que pode ser projetado como um simples vértice (ao contrário do uso de 2 triângulos e 4 vértices, geralmente utilizados em *sprites* convencionais). O uso de *point sprites* particularmente permite um melhor suporte à síntese de cenas que contêm efeitos especiais (fogo, fumaça, chuva, etc), modelados por um sistema de partículas. O mapeamento de texturas pode ser encapsulado em objetos de textura, o que permite o armazenamento dos dados no motor gráfico, conseqüentemente, obtendo um melhor desempenho. Texturas 1D e 3D foram eliminadas, mas texturas 2D (que podem simular texturas 1D) foram mantidas. Os dados dos vértices também podem ser encapsulados em *vertex buffers* do objeto, o que facilita o acesso aos vértices, gerando também um melhor desempenho. Há duas extensões opcionais na versão 1.1: *draw texture*, que permite o uso do *texturing machinery* para copiar retângulos alinhados aos eixos de coordenadas para o *frame buffer*; e *matrix palette*, que suporta a aplicação de interpolação lineares, útil para o controle de objetos articulados em movimento, tais como, personagens humanos.

Já a versão 2.0 (Figura 3.2) possui um *pipeline* programável que, para tal, teve as transformações dos vértices (de modelagem e projeção) e estágios de iluminação do *pipeline* anteriormente proposto, substituídas pelo tonalizador de vértices. O tonalizador de vértices, corresponde a uma sequência de instruções executadas para cada vértice. Normalmente, recebe como entrada atributos dos vértices (posição, cor, coordenada de textura e vetor normal). A seqüência de instruções irá manipular os atributos dos vértices para criar efeitos como texturas

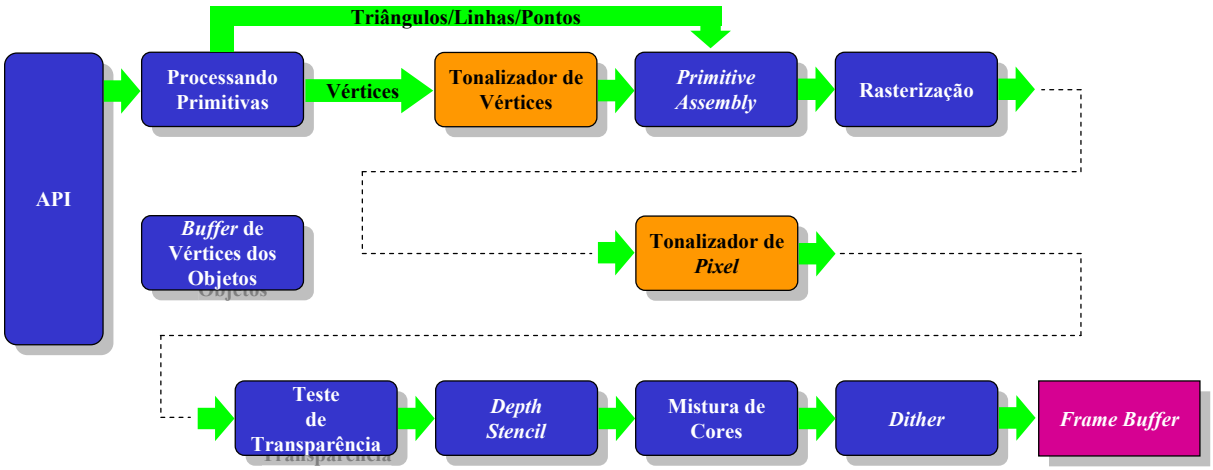


Figura 3.2: *Pipeline* da OpenGL ES 2.0 (programável).

procedurais, movimentos complexos, manipulação dinâmica de cores, etc. Adicionalmente, na versão 2.0, o *fragment pipeline* foi substituído pelo tonalizador de *pixels*, que pode ser utilizado para a renderização em tempo-real de objetos complexos realistas (cabelo, roupa, vidro, materiais orgânicos, etc); criação de novos modelos de iluminação (e não somente os modelos apresentados na OpenGL ES); e geração de novos tonalizadores não-fotorealistas, como renderizações do tipo *cartoon*, etc. O tonalizador de *pixels* corresponde a um conjunto de instruções que são executadas para cada pixel individualmente, antes deste ser preenchido com a cor desejada. Embora a API 2.0 não seja compatível com a 1.x, estas versões foram projetadas para que os fabricantes de *hardware* pudessem prover *drivers* que possibilitassem a execução de ambas versões em um mesmo dispositivo.

3.2.2 Sistemas de Coordenadas Global e da Câmera

O sistemas de coordenadas da OpenGL ES, assim como da OpenGL, é o sistema de coordenadas da mão direita. O sistema de coordenadas da câmera é definido de forma que a direção de visão coincida com o eixo z negativo, o eixo x positivo aponte para à direita e o eixo y positivo aponte para cima. As projeções da câmera podem ser do tipo ortográfica (geradas pelo método *glOrthof*) ou perspectiva (geradas pelo método *glFrustumf*). Os parâmetros do modelo da câmera que devem ser passados para esses métodos são: o ângulo de visão, a razão de aspecto e os planos de recorte próximo e distante.

3.2.3 Outras Funcionalidades

Encontra-se também disponível na API OpenGL ES um método de *antialiasing*, que suaviza o efeito serrilhado de objetos, principalmente, dos segmentos de reta. Vale ressaltar que novos efeitos de visualização podem ser implementados através do *hardware* programável, disponível na versão 2.0.

A definição de texturas é realizada utilizando-se um vetor de vértices. A API OpenGL ES apresenta geração automática de *mipmaps*; filtro de texturas; *tiled* (repetição de um padrão de textura no mesmo objeto); e *point sprites*.

A aparência dos objetos é definida por um vetor de vértices. As informações que definem a aparência são: a influência da luz sobre o objeto, a textura, a transparência (se o objeto é translúcido ou opaco), etc. OpenGL ES apresenta *backface culling* para a remoção de partes ocultas da geometria de objetos que não estão visíveis pela câmera. Pode ser habilitado ou não, utilizando-se o método *glEnable(GL_CULL_FACE)*.

Apresenta quatro tipos de luzes que podem ser ativadas para iluminar uma cena: ambiente, direcional, pontual e *spot*. Para se definir uma luz cria-se uma instância dessa luz com a informação das cores, habilita-se o seu uso e informa-se a sua influência (quantidade de luz que será absorvida) pelos materiais. É possível escolher o modelo de tonalização utilizando-se o método *glShadeModel*.

Para a movimentação de objetos são utilizadas transformações geométricas básicas. Por exemplo, para a translação, utiliza-se o método *glTranslatef*. Para a rotação e a escala são utilizados, respectivamente, os métodos *glRotatef* e *glScalef*. Apresenta também o método *glLoadIdentify* que zera a matriz, removendo todas as transformações anteriormente existentes. Os métodos *glPushMatrix* e *glPopMatrix* são utilizados para a realização de operações de empilha e desempilha, respectivamente, na pilha de matrizes de projeção e *modelview*.

OpenGL ES não apresenta nenhum controle de animação implementado. Técnicas para o controle da animação podem ser implementadas utilizando-se o método *display()*, que é executado a cada quadro antes da renderização, porém, apresenta o método *vertex skinning* como alternativa para o controle local das junções de objetos articulados.

3.3 Comentários Finais

Nesta dissertação é usada a *Vincent Mobile Graphics* [52], que implementa a versão 1.1 da API OpenGL ES e disponibiliza o código-fonte. Atualmente, há versões da OpenGL

ES para PocketPC e *Smartphone*. Durante a fase de desenvolvimento deste trabalho, foi identificado um erro na implementação da *Vincent*. Em algumas ocasiões, durante a locomoção no ambiente 3D, este erro gerava uma visualização incorreta dos polígonos recortados. Em particular, os experimentos realizados nesta dissertação contribuíram para que esse erro fosse corrigido na versão do código-fonte da *Vincent*, disponível na *Internet*. Adicionalmente, foi também usada a implementação da OpenGL ES disponível na SDK da Nokia Série 60.

No próximo capítulo, será introduzido e detalhado o VisMobile, um sistema gráfico 3D para renderização interativa em dispositivos móveis, implementado durante o desenvolvimento deste trabalho.

Capítulo 4
VisMobile

4.1 Visão Geral

VisMobile é o sistema proposto e implementado para a renderização interativa em pequenos dispositivos móveis, utilizando a API OpenGL ES. Consiste em um conjunto de pacotes de classes (as quais incluem algoritmos de visibilidade e estruturas de particionamento espacial) que pode ser reutilizado e estendido para o desenvolvimento de aplicações gráficas 3D. Em particular, a implementação corrente do VisMobile tem como objetivo principal garantir a locomoção, em tempo real, do observador (câmera) pelo ambiente 3D exibido na tela de dispositivos móveis, os quais tipicamente possuem recursos computacionais limitados. A versão atual da implementação está disponível para *Smartphones* e *Pocket PCs* com Windows Mobile, bem como para celulares Nokia Série 60 com Symbian OS.

VisMobile é composto por dois módulos básicos: 1) o que particiona o ambiente 3D em estruturas de dados espaciais; e 2) o que contém os pacotes das estruturas de particionamento espacial e dos algoritmos de visibilidade, a serem utilizados nas aplicações gráficas desenvolvidas. Como já mencionado, o uso combinado de algoritmos de visibilidade e estruturas de particionamento espacial é importante como possível estratégia empregada para diminuir o número de polígonos a serem renderizados na cena e, desta forma, garantir um melhor desempenho da aplicação. Tanto o primeiro módulo quanto o segundo podem ser executados em computadores pessoais e em dispositivos móveis (Figura 4.1). Entretanto, o primeiro módulo opera em fase de pré-processamento e, portanto, é dependente do poder de processamento da plataforma de execução (seja um computador pessoal tradicional ou um dispositivo móvel). Já o segundo módulo, pode ser executado em diferentes plataformas, desde que apresentem implementação da

OpenGL ES.

Dentre alguns dos desafios para o desenvolvimento do VisMobile, podem ser destacadas as otimizações realizadas nos algoritmos de visibilidade e em algumas estruturas de particionamento espacial, de tal forma a garantir a locomoção do observador, a taxas interativas, no ambiente 3D exibido no dispositivo móvel.

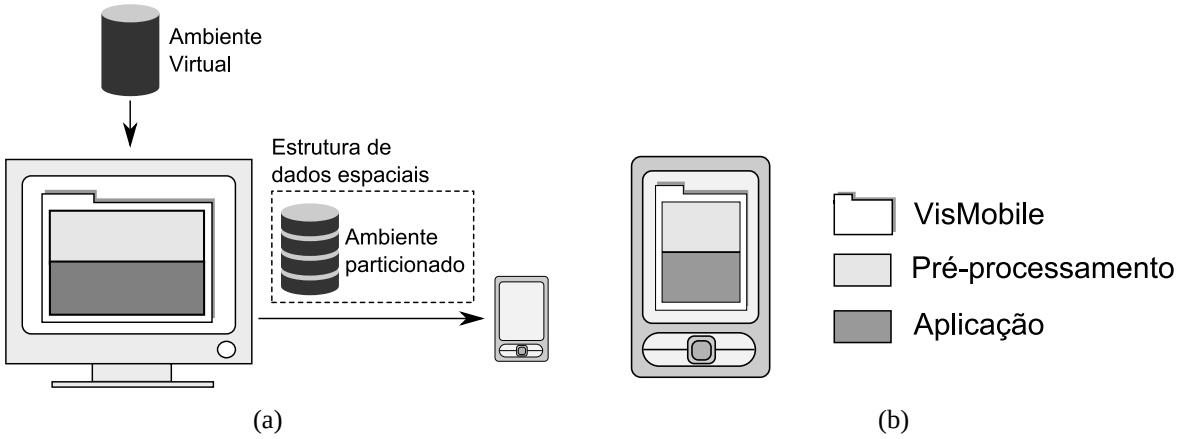


Figura 4.1: Um exemplo de duas plataformas de execução. Em (a), devido às restrições de poder de processamento do dispositivo, o módulo de pré-processamento executa no computador pessoal (nesse caso o ambiente particionado pode ser acessado localmente, por exemplo, via USB ou remotamente, via rede). Em (b), os dois módulos executam no dispositivo.

VisMobile foi desenvolvido utilizando programação orientada a objeto, incluindo características básicas, tais como: herança, polimorfismo, interface, abstração, etc, as quais permitem a sua reutilização e extensão. Duas linguagens de programação foram utilizadas no desenvolvimento do VisMobile: a primeira, para a implementação do módulo que particiona o ambiente em estruturas de dados espaciais, executado em fase de pré-processamento; e a segunda, para a implementação do módulo que contém as classes das estruturas de particionamento espacial e dos algoritmos de visibilidade, a serem utilizadas nas aplicações gráficas desenvolvidas. Java 1.6 foi utilizada para a implementação do módulo executado em fase de pré-processamento por ser uma linguagem de alto nível, portável e que disponibiliza um conjunto de pacotes os quais facilitam o processo de implementação (por exemplo, os pacotes que contêm estruturas de dados). Adicionalmente, esse módulo utiliza a API gráfica Java3D para fazer a leitura e a prévisualização dos modelos geométricos. Além disso, em conjunto com Java, foi desenvolvida uma biblioteca nativa em C ANSI para otimizar o uso do processador. Dessa maneira, os métodos são executados nativamente, obtendo um melhor desempenho. Já o módulo que contém as classes das estruturas de dados espaciais e dos algoritmos de visibilidade foi implementado utilizando-se C++ por ser uma linguagem de baixo nível, leve, robusta e portável (dependendo de como é realizada a implementação). Em conjunto com a linguagem C++ foi utilizada a

API gráfica OpenGL ES, para otimizar ainda mais o desempenho da aplicação gráfica. Utiliza também os métodos de interface e interação da API GLUT|ES[1]. Mais especificamente, para *Windows Mobile* foi utilizada a IDE *Embedded* Visual C++ 4.0 em conjunto com o seu SDK e para o Nokia Série 60 foi utilizada a IDE Carbide.C++ *Express* em conjunto com o SDK S60 FP2.

Na Seção 4.2, será descrita e detalhada a arquitetura do VisMobile.

4.2 Arquitetura

O VisMobile segue o padrão de arquitetura MVC (*Model-View-Control*) [55], usado para particionar o ambiente em três partes: **Modelo**, **Visão** e **Controle** (neste trabalho, o **Controlador do Dispositivo**).

O **Modelo** administra os dados correntes do sistema e o comportamento dos objetos 3D, disponibiliza para a **Visão** os dados requisitados e executa as instruções que foram interpretadas pelo **Controlador do Dispositivo**. Isso significa que o **Modelo** encapsula não apenas o estado do sistema, como também toda a lógica de como o sistema funciona. Todos os objetos gráficos que compõem o ambiente fazem parte do **Modelo**. Adicionalmente, no **Modelo** estão implementados os algoritmos de visibilidade, assim como as estruturas de particionamento espacial (Figura 4.2).

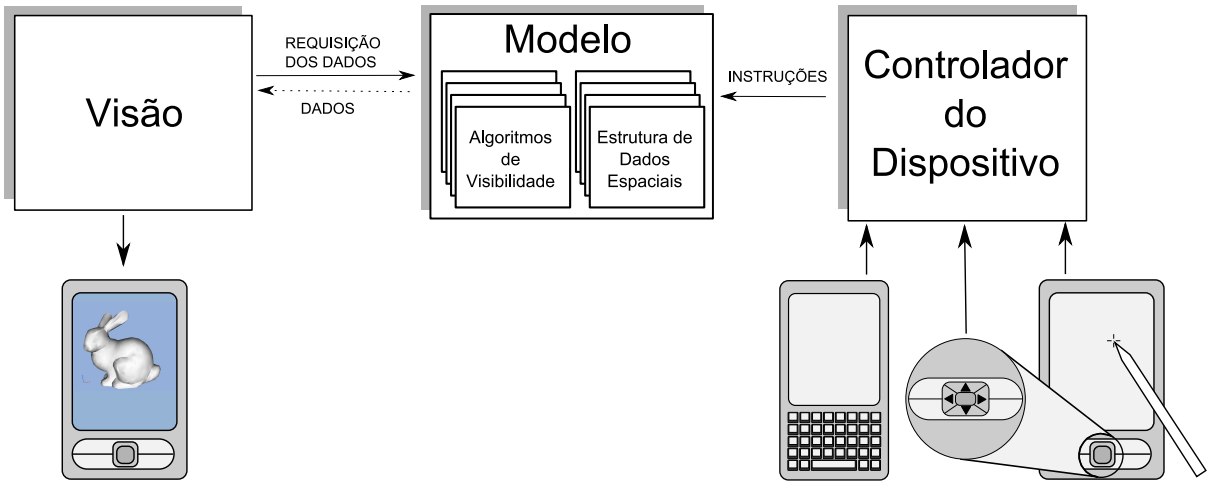


Figura 4.2: Arquitetura do VisMobile, baseada no padrão de arquitetura MVC.

A **Visão** contém informações básicas para a renderização das cenas 3D na tela do dispositivo, assim como as informações da câmera (ângulo de visão, razão de aspecto, plano

[1]GLUT|ES é uma API para a criação de janelas gráficas. Possibilita também identificar e interpretar as mensagens do sistema operacional.

próximo e plano distante). Os parâmetros definidos para a câmera podem influenciar no resultado final do processo de visualização de uma cena. Por exemplo, quanto mais longe da câmera o plano distante estiver localizado, mais triângulos estarão contidos no *frustum*. A cada quadro gerado durante a locomoção do observador (câmera), a **Visão** requisita ao **Modelo** os dados a serem renderizados. No VisMobile, outros perfis de visões podem ser criados e/ou anexados, por exemplo, visando a geração de imagens com diferentes níveis de detalhamento, dependendo dos recursos computacionais disponíveis no dispositivo. Além disso, a API OpenGL ES pode ser substituída por outras APIs gráficas disponíveis para dispositivos móveis, desde que estas apresentem características e funcionalidades similares a OpenGL ES como, por exemplo, tipos de primitivas geométricas e formas de agrupamento, mapeamento e normalização entre sistemas de coordenadas, etc.

O **Controlador do Dispositivo** interpreta as operações de entrada de um dispositivo, tais como: os comandos do teclado, da caneta e do *joystick*, repassando as instruções já traduzidas ao **Modelo** (Figura 4.2). Durante a locomoção em um ambiente 3D, as interações do usuário são interpretadas pelo **Controlador do Dispositivo** que informa as instruções do usuário ao **Modelo**. Na realidade, o **Controlador do Dispositivo** corresponde à interface de interação do usuário com a aplicação. Cada vez que o usuário muda a posição ou a direção de visão da câmera, o **Modelo** atualiza a área visível e a **Visão** requisita do **Modelo** a região do ambiente a ser renderizada (Figura 4.3).

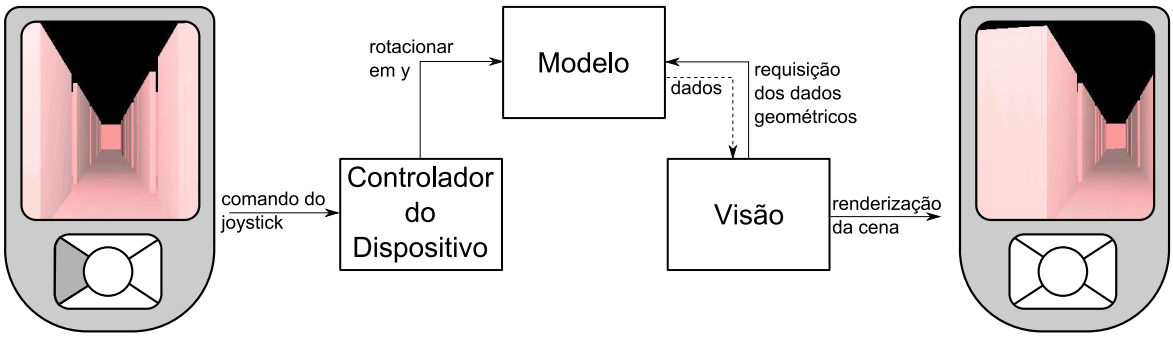


Figura 4.3: Comunicação entre os componentes do VisMobile durante a locomoção no ambiente.

4.3 Algoritmos de Visibilidade Implementados

Os algoritmos de visibilidade *view-frustum culling*, *occlusion culling* e *backface culling* são implementados no **Modelo**. O VisMobile disponibiliza classes para que as aplicações gráficas possam fazer uso desses algoritmos.

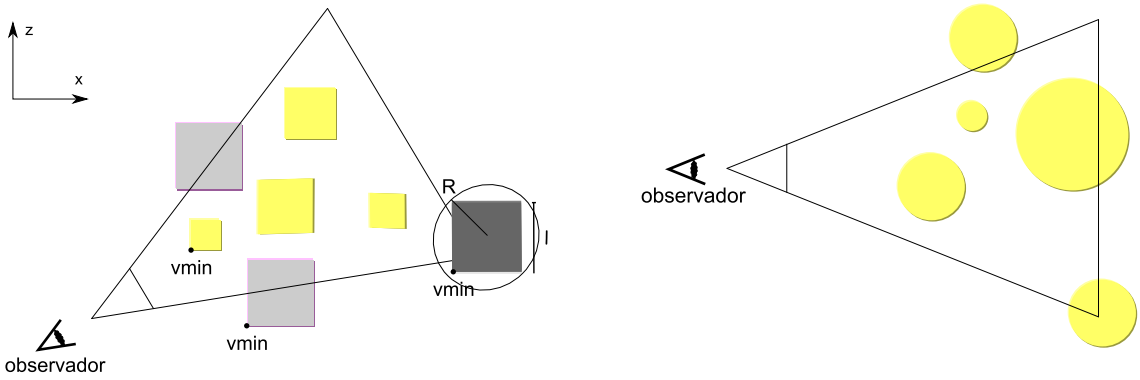
4.3.1 *View-Frustum Culling*

Para o *view-frustum culling*, o VisMobile disponiliza uma classe com 3 métodos principais: um para identificar os planos que compõem o *frustum* de visualização; e dois para verificar a interseção de objetos da cena com o volume de visualização. A verificação da interseção com o *frustum* de visualização requer diferentes testes de interseção, dependendo da estrutura de dado espacial utilizada. Por exemplo, em uma *Octree* os testes de interseção são realizados entre os volumes cúbicos dos nós e o *frustum*; já em uma *BSP-Tree*, os testes de interseção são realizados entre o *frustum* de visualização e o plano de partição.

Os algoritmos de *view-frustum culling* podem descartar muitos polígonos da cena, dependendo de sua complexidade, removendo os que estão fora do volume de visualização. São algoritmos leves, relativamente simples de serem implementados e podem ser associados a outros algoritmos de visibilidade. Dentre os métodos de interseção implementados nesta dissertação, podem ser citados o teste de interseção entre o volume de visualização e um ponto (utilizado no teste de interseção entre o *frustum* de visualização e o cubo, especificado no Algoritmo 1 e na Figura 4.4(a)); e entre o volume de visualização e uma esfera (Figura 4.4(b)). Outros testes de interseção com o *frustum* podem ser realizados. Por exemplo, no teste de interseção do volume de visualização com um cubo utilizam-se os seus vértices, verificando se os mesmos estão fora do *frustum*. Se, pelo menos, um dos vértices do cubo estiver contido no volume de visualização, o cubo estará visível ao observador. Se nenhum dos vértices estiver contido no volume de visualização, verifica-se ainda se o cubo se encontra na situação exibida em (a) da Figura 4.4 (representada pelo cubo escuro). Nesse caso, todos os vértices do cubo estão fora do *frustum*, porém, este é visível ao observador. Em particular, a abordagem utilizada nesse trabalho é envolver o cubo de lado l em um volume esférico de centro $c_{(x,y,z)} = \mathbf{vmin}_{(\mathbf{x},\mathbf{y},\mathbf{z})} + l/2$ e de raio $r = \sqrt{3 \times l^2}$, testando se a esfera está ou não fora do *frustum* (sendo $\mathbf{vmin}_{(\mathbf{x},\mathbf{y},\mathbf{z})}$ o vértice do cubo que apresenta o menor valor nos 3 eixos de coordenadas).

Algoritmo 1 Algoritmo *view-frustum culling*

```
ViewFrustumCulling(Planos, Cubos) //Realiza o View-Frustum Culling
Entrada: Planos do volume de visualização e os cubos que são verificados para identificar se interceptam o frustum
 1: //Percorre todas os cubos
 2: para cada Cubos faça
 3:     se Cubo ∩ Planos então
 4:         //O modelo envolvido pelo cubo é renderizado quando este intercepta o volume de visualização
 5:         Renderiza(Cubo)
 6:     senão
 7:         //Envolve o cubo com uma esfera
 8:         Esfera Envoltória ( Cubo )
 9:         se Esfera Envoltória ∩ Planos então
10:             //O modelo envolvido pelo cubo é renderizado quando sua esfera envoltória intercepta o volume de visualização
11:             Renderiza(Cubo)
12:         fim se
13:     fim se
14: fim para
```



(a) (b)

Figura 4.4: Em (a), testes identificando os cubos que estão fora do *frustum* e, em (b), os testes são realizados com esferas.

4.3.2 *Occlusion Culling*

Como apresentado no Capítulo 2, além do algoritmo de *view-frustum culling*, existem outros métodos para descarte dos polígonos que não estão visíveis ao observador. No caso, o *occlusion culling* identifica os polígonos ocultos por outros polígonos da cena. Após a execução do algoritmo de *view-frustum culling*, o resultado deste algoritmo é enviado como parâmetro de entrada para o algoritmo de *occlusion culling*, desta forma, reduzindo o número de testes para a identificação dos objetos ocultos, otimizando o desempenho do método. Adicionalmente, pode-se utilizar coerência temporal para a escolha dos obstáculos. O algoritmo de *occlusion culling* pode ser dividido em duas etapas: a etapa de escolha dos polígonos a serem utilizados como obstáculos e a etapa de teste de oclusão entre os obstáculos e os objetos (Algoritmo 2).

Algoritmo 2 Algoritmo para identificar objetos ocultos usando *occlusion culling*

```
OcclusionCulling(Polígonos) //Identifica os polígonos ocultos
1: //Percorre todos os polígonos do modelo
2: para cada Polígono do modelo faça
3:     se Polígono não está oculto então
4:         //Envia ao pipeline de renderização o polígono
5:         renderiza(Polígono)
6:     fim se
7: fim para
8: //Todos os polígonos enviados ao pipeline de renderização serão os obstáculos no próximo quadro
9: Obstáculos ← Polígonos Renderizados Neste Quadro
```

No VisMobile, a cada quadro, os polígonos renderizados no quadro anterior são ordenados segundo uma métrica simples, baseada nos ângulos sólidos[2]. O valor estimado do ângulo sólido pode ser obtido por $\frac{-A(\mathbf{N}\cdot\mathbf{V})}{\|\mathbf{D}\|^2}$ onde A e $\mathbf{N}$ representam a área e o vetor normal da face do obstáculo, respectivamente; $\mathbf{V}$ o vetor de visão do observador; e $\mathbf{D}$ o vetor gerado a partir do

[2] Ângulo Sólido é um valor utilizado para medir o tamanho de um determinado objeto a um ponto de referência, sendo que pequenos objetos próximos ao ponto de referência podem apresentar um tamanho igual aos grandes objetos localizados distante desse mesmo ponto.

observador em direção ao centro do obstáculo (Figura 4.5). Gera-se então uma lista ordenada em ordem decrescente, a qual é percorrida identificando-se os objetos ocultos.

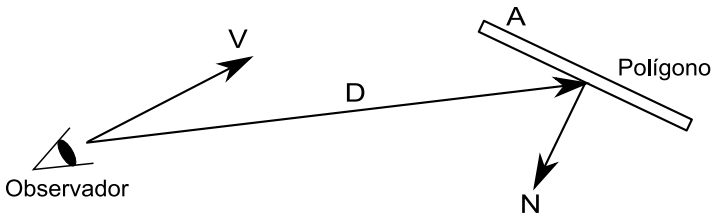


Figura 4.5: Parâmetros da métrica utilizada para a escolha dos obstáculos.

Para identificar se um objeto está oculto por um obstáculo, utilizam-se os planos gerados a partir da posição do observador com os vértices do obstáculo (Figura 4.6). Para cada plano gerado, verifica-se se este intercepta o objeto. Caso um ou mais planos interceptem o objeto, este não estará oculto. Se nenhum plano interceptar o objeto, verifica-se ainda se o objeto encontra-se posicionado no volume gerado pelos planos. Em caso afirmativo, o objeto estará oculto pelo obstáculo.

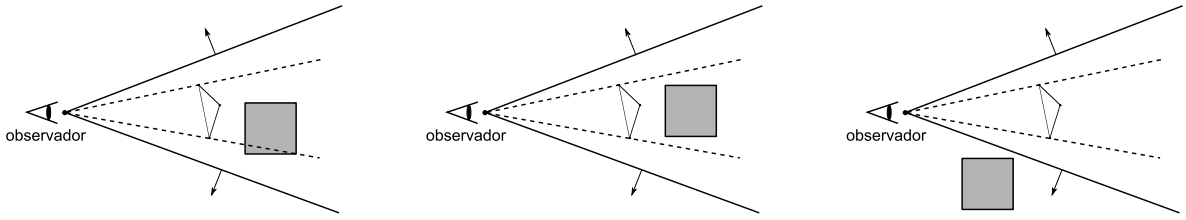


Figura 4.6: Ilustração do teste de identificação dos objetos ocultos em um espaço 2D. Em (a), o objeto não está sendo ocultado pelo obstáculo. Em (b) e (c) os planos não interceptam o objeto. Porém, em (b) o centro do objeto está contido no volume gerado pelos planos, estando oculto, e em (c), o centro do objeto não está contido no *frustum*.

4.3.3 *Backface Culling*

Como descrito no Capítulo 3, a API OpenGL ES apresenta o algoritmo de *backface cullling* implementado. Porém, o algoritmo de *backface cullling* da API OpenGL ES, em alguns casos, degradou o desempenho da aplicação (maiores detalhes a respeito serão apresentados no Capítulo 5). Este fato motivou o desenvolvimento de um novo algoritmo de *backface culling* chamado, neste trabalho, de *backface culling* conservativo, o qual será detalhado na próxima seção.

4.3.4 *Backface Culling* Conservativo

No algoritmo de *backface culling* conservativo proposto e implementado nesse trabalho (Algoritmo 3), o espaço de visão é inicialmente dividido em um número de regiões *n*, sendo que *n* depende do ângulo definido para o *field of view* (FOV), calculado através da equação $n = \frac{360^\circ}{90^\circ - FOV}$. A região para a qual a normal do polígono aponta (direção do vetor normal dos polígonos), como mostra a Figura 4.7, é assim identificada e pré-processada. Antes da renderização de cada quadro, é verificada se a região do polígono coincide com a região da direção de visão do observador. Em caso afirmativo, o polígono é descartado.

Este algoritmo, diferentemente do algoritmo disponível na OpenGL ES, rapidamente identifica e descarta polígonos com faces não voltadas para o observador. Contudo, este pode não identificar todos os polígonos que não estão com as faces voltadas para o observador (motivo pelo qual foi chamado de *backface culling* conservativo). Por fim, pode-se mencionar a possibilidade de sua utilização em conjunto com o algoritmo de *backface culling* da API OpenGL ES, de tal forma a melhorar o desempenho do algoritmo de *backface culling* da OpenGL ES.

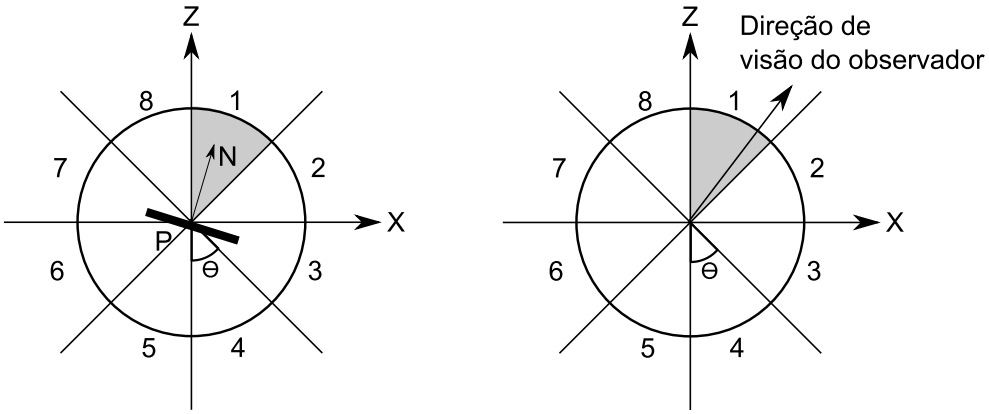


Figura 4.7: Representação 2D da otimização do *backface culling*. À esquerda é visualizado o quadrante do polígono. À direita, o quadrante do observador, sendo θ o ângulo de cada região e *N* a normal do plano *P*.

4.4 Estruturas de Particionamento Espacial Implementadas

Nas próximas seções, serão descritas as estruturas para particionamento espacial implementadas no VisMobile: *Grid* Irregular, *Octree*, Portal *Octree* e *BSP-Tree*.

Algoritmo 3 Algoritmo para otimizar o *back-face culling*

```
Load(Modelo3D) //Carrega o modelo 3D
 1: para cada Polígono do modelo faça
 2:     Triângulo ← Modelo3D //Lê triângulo do modelo
 3:     Triângulo.Região ← CalculaRegião(Triângulo) //Identifica a região do triângulo
 4: fim para
EachFrameRender //Método executado a cada quadro
 5: para cada Poligono faça
 6:     se Estiver no frustum então
 7:         se Não estiver oculto por algum obstáculo então
 8:             se observador.região ≠ triângulo.região então
 9:                 //Renderiza os triângulos que estão em regiões diferentes da região do observador
10:                 renderiza(triângulo)
11:             fim se
12:         fim se
13:     fim se
14: fim para
```

4.4.1 *Grid* Irregular

No *Grid* Irregular, é definido o número de partições em cada um dos eixos de coordenadas, assim como o tamanho de cada uma das divisões. Por exemplo, pode-se informar que no eixo *x* há 4 partições, sendo 2 delas equivalentes a 20% do tamanho original da cena e as outras duas a 30%. O *Grid* Irregular é uma estrutura simples, que não faz uso de muitos recursos computacionais. Além disso, na escolha do número e do tamanho das partições pode-se levar em consideração o conhecimento prévio da geometria e da distribuição dos objetos estáticos no ambiente (Figura 4.8).

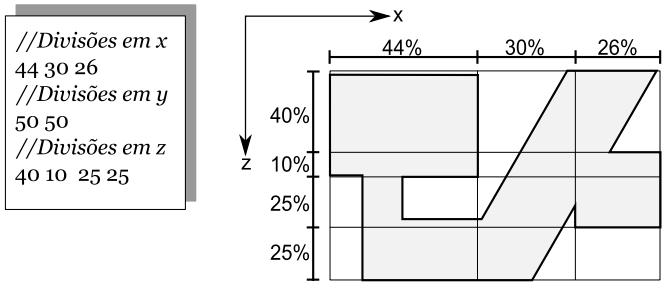


Figura 4.8: Partição de um ambiente utilizando *Grid* Irregular. À esquerda visualizam-se parâmetros definidos para a divisão do *Grid* e, à direita, é exibida a cena particionada correspondente.

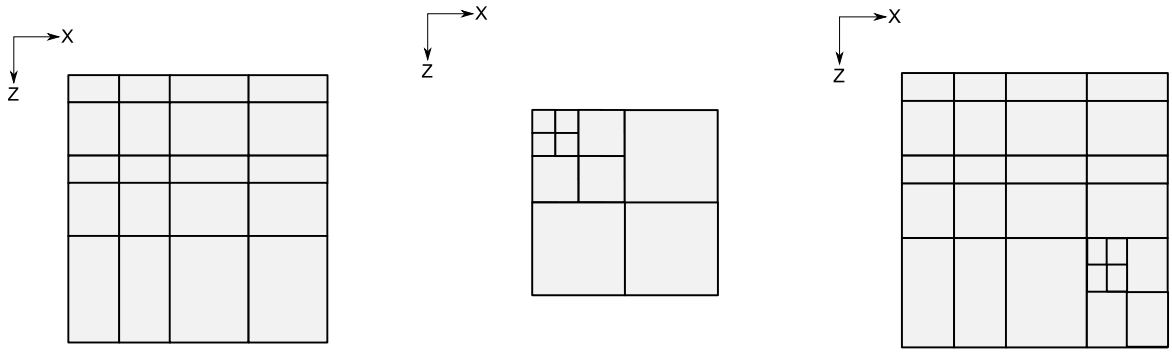
4.4.2 *Octree*

Já na implementação da *Octree*, durante a construção da estrutura, o nós são divididos até que a profundidade da árvore alcance um limite pré-definido pelo desenvolvedor. Este valor limite pode ser alterado durante a criação da estrutura (nível 5 corresponde à profundidade padrão implementada na versão corrente do VisMobile). A raiz da estrutura corresponde ao nó que contém todo o ambiente 3D. Nos casos em que um mesmo polígono pertence a mais de um

octante da *Octree*, ele não é particionado e, é armazenado pelos octantes que o interceptam. Em particular, os nós folhas da *Octree* implementada no VisMobile armazenam apenas os índices dos polígonos para evitar a redundância de dados. A divisão dos polígonos nessa situação aumentaria a quantidade de polígonos do ambiente gerando, desnecessariamente, polígonos adicionais, os quais seriam enviados ao *pipeline* de renderização, prejudicando o desempenho da aplicação.

4.4.3 Portal *Octree*

Por sua vez, a implementação do Portal *Octree* utiliza *Grid* Irregular para particionar o ambiente, com os *voxels* do *Grid* correspondendo às células (Figura 4.9). Conhecendo-se o ambiente 3D *a priori*, partições específicas podem ser escolhidas para que o *Grid* melhor se adapte à estrutura geométrica da cena. No VisMobile, para cada célula é realizada a partição espacial usando *Octrees*. Assim, similarmente ao que acontece na *Octree*, na Portal *Octree* pode-se também definir previamente o nível de profundidade máximo da árvore.



(a) *Grid* Irregular (b) *Octree* (c) Portal *Octree*

Figura 4.9: Diferentes estruturas de dados espaciais. Em (c), pode-se observar a Portal *Octree* composta por (a) e (b).

4.4.4 *BSP-Tree*

Finalmente, a *BSP-Tree* implementada no VisMobile testa todos os planos dos polígonos do ambiente e escolhe um deles como hiperplano de partição. A regra de escolha deste plano baseia-se na melhor distribuição de polígonos entre os planos particionados. O critério de parada da construção da *BSP-Tree* também pode ser previamente definido pelo desenvolvedor. Na implementação corrente, o critério de parada segue a seguinte regra: $\frac{h^+}{h^-} < 0.8$ *ou* $\frac{h^+}{h^-} > 1.25$, onde h^+ e h^- correspondem a todos os polígonos presentes nas regiões positiva e negativa, respectivamente, sendo definidas pelo hiperplano de partição [56]. Os polígonos que se en-

contram nos dois lados da partição não são particionados, sendo suas referências armazenadas pelos nós de ambos os lados da partição. Novamente, como na *Octree*, objetivando minimizar o número de polígonos armazenados na estrutura.

Vale ressaltar que a combinação de algoritmos de visibilidade com diferentes estruturas de particionamento espacial pode obter desempenhos variados. Por exemplo, na *Octree* faz-se necessário $5 * n * 8$ testes para a realização do algoritmo de *view-frustum culling*. Sendo 5 o total de testes para identificar se um cubo está contido no *frustum* (4 testes são realizados utilizando-se os 4 vértices do cubo e 1 teste é realizado utilizando-se a sua envoltória esférica), n a profundidade da estrutura e 8, o número de filhos possíveis para cada nó. Já na *BSP-Tree*, faz-se necessário n testes para a realização do algoritmo de *view-frustum culling* (sendo n a profundidade da estrutura).

Na próxima seção, serão apresentadas as possíveis formas de interação do usuário com a aplicação gráfica via uma interface simples, projetada para o VisMobile.

4.5 Interface

No VisMobile, no início da execução da aplicação, o usuário tem a opção de escolher as estruturas de particionamento espacial a serem utilizadas[3] (Figura 4.10).

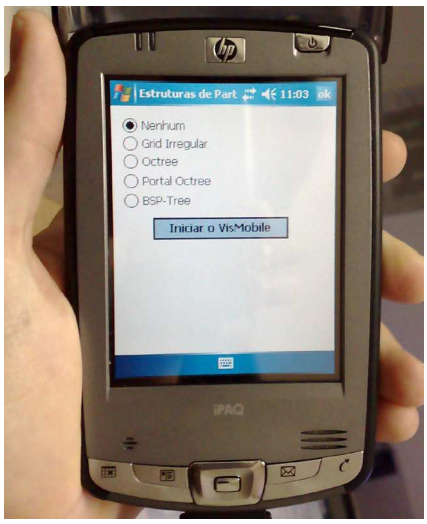


Figura 4.10: Interface inicial do VisMobile.

Após a escolha da estrutura de dados espacial a ser utilizada na aplicação, o usuário pode se locomover no ambiente usando o *joystick* do dispositivo, no qual os botões *cima* e *baixo* transladam o observador, respectivamente, para frente e para trás; e os botões para os

[3]No entanto, o nível de profundidade das estruturas deve ser definido pelo desenvolvedor.

lados *esquerda* e *direita* rotacionam o observador em torno do eixo *y* (Figura 4.11). Na versão atual do VisMobile, o usuário não tem controle sobre a velocidade de locomoção e a aplicação não contempla detecção de colisão entre objetos.

Na tela de visualização, o usuário pode verificar os algoritmos de visibilidade que estão ativos, a quantidade de memória utilizada, o número de triângulos enviados ao *pipeline* de renderização e a taxa de quadros por segundo gerada durante a locomoção do observador pelo ambiente. Adicionalmente, o usuário pode acessar o menu da aplicação, o qual disponibiliza ferramentas para gravar e carregar uma trajetória qualquer pelo ambiente, bem como para executar a locomoção automática do observador por um caminho previamente armazenado no dispositivo. Além disso, através do menu, o usuário pode habilitar (ou não) algoritmos específicos de visibilidade.

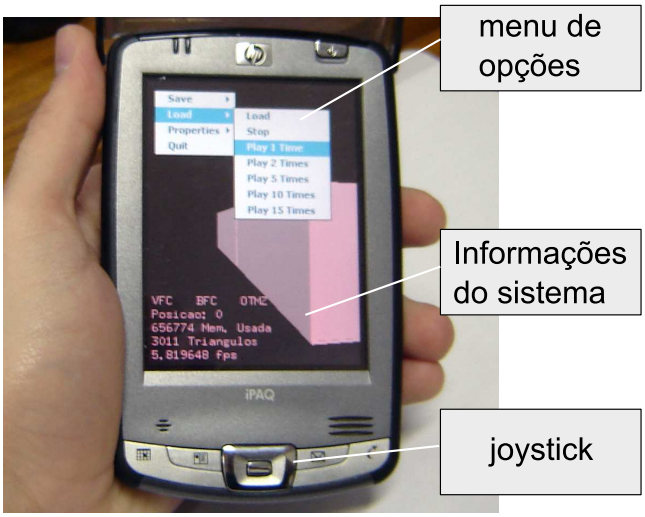


Figura 4.11: Tela do dispositivo, no caso (iPaq hx2490b), a partir da qual o usuário pode acessar o menu. Informações específicas sobre a aplicação podem ser visualizadas na tela do dispositivo. Através do *joystick*, o usuário pode se locomover no ambiente.

No próximo capítulo, serão apresentados e detalhados todos os testes realizados no VisMobile. Inicialmente será realizada uma descrição geral da metodologia definida e executada para os testes. Em seguida, serão apresentados os ambientes 3D modelados e utilizados para a análise de desempenho (4 ambientes internos e 2 externos), bem como as trajetórias de locomoção utilizadas nos ambientes. Serão então combinados diferentes algoritmos de visibilidade e tipos de estruturas de particionamento espacial. Baseando-se nos resultados obtidos, serão traçadas considerações sobre a eficiência do uso de combinações de algoritmos de visibilidade e, de variações de profundidades de algumas das estruturas de particionamento espacial implementadas, levando-se em consideração os diferentes ambientes gráficos testados, bem

como a limitação de recursos computacionais das plataformas de execução utilizadas.

Capítulo 5
Testes e Análise de Desempenho

5.1 Descrição Geral

Foram conduzidos 120 testes, os quais serviram como base para a análise de desempenho dos algoritmos e estruturas de dados implementadas no VisMobile. As plataformas utilizadas nos testes foram os dispositivos iPaq e N82. A configuração dos dispositivos se encontra na Tabela 5.1.

	iPaq hx2490b	**Nokia n82**
Plataforma	Windows Mobile 5.0	Symbian OS 9.2, S60
Tamanho do Visor	3.5”	2.4”
Número de Cores	16M	16M
Memória Interna	128mb	100mb
Memória Heap	64mb	128mb
Processador	520MHz Intel PXA270	ARM 11 332 MHz
Apresenta GPU	não	sim

Tabela 5.1: Configuração dos dispositivos móveis utilizados nos casos de estudo.

Estes testes inicialmente incluíram oito combinações de algoritmos de visibilidade: sem o uso de algoritmos de visibilidade, *backface culling*, *backface culling* conservativo, *backface culling* com *backface culling* conservativo, *view-frustum culling*, *view-frustum culling* com *backface culling*, *view-frustum culling* com *backface culling* conservativo e *view-frustum culling* com *backface culling* e *backface culling* conservativo. Por não ter apresentado resultados significativos, o algoritmo de *occlusion culling* (implementação atual do mesmo) não será incluída na discussão da análise de desempenho.

As melhores combinações de algoritmos de visibilidade obtidas nesses experimentos

foram novamente comparadas, variando-se agora as estruturas de particionamento espacial utilizadas (*Grid* Irregular, *Octree*, Portal *Octree* e *BSP-Tree*), bem como, para a *Octree* e a Portal *Octree*, os níveis de profundidade destas duas estruturas.

5.2 Ambientes 3D Modelados

Estas variações foram sistematicamente testadas em seis ambientes gráficos os quais foram modelados e são descritos nesse trabalho, sendo quatro internos (Mundo 1, Mundo 2, Mundo 3 e Mundo 4) e dois externos (Mundo 5 e Mundo 6), conforme detalhado na Tabela 5.2. Os seis mundos apresentam níveis de complexidade geométrica distintos, contendo uma quantidade variada de triângulos distribuídos pelos respectivos ambientes. Os objetos (abajur, bule, coelho, dragão, flor, pinguim taça e vaca) estão presentes de forma variada nos ambientes e seus respectivos números de triângulos são exibidos na Tabela 5.4.

Ambiente	**Número Total de Triângulos**	**Tamanho do OBJ (KB)**	**Tipo de Ambiente**	**Distribuição dos Objetos**
Mundo 1	6.199	214	Interno	Espalhados uniformemente
Mundo 2	102.232	4144	Interno	Espalhados uniformemente
Mundo 3	102.232	4178	Interno	Quantidade massiva de objetos agrupados em uma região específica
Mundo 4	30.199	1048	Interno	Quantidade massiva de objetos espalhados uniformemente
Mundo 5	2.548	108	Externo	Espalhados uniformemente
Mundo 6	10.040	511	Externo	Espalhados uniformemente

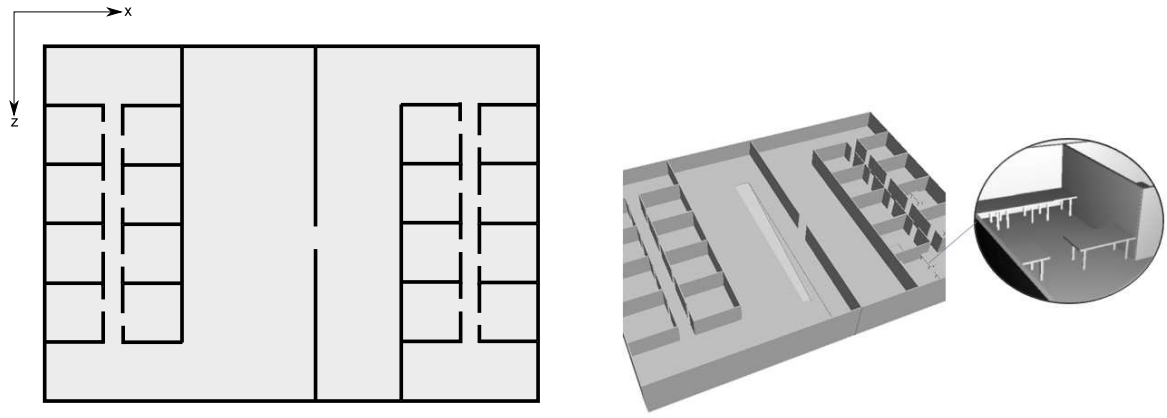
Tabela 5.2: Detalhamento dos ambientes modelados e utilizados nos testes de análise de desempenho do VisMobile.

Utilizando-se as estruturas de dados espaciais mencionadas anteriormente (*Grid* Irregular, *Octree*, Portal *Octree* e *BSP-Tree*), os seis ambientes (Mundo 1, Mundo 2, Mundo 3, Mundo 4, Mundo 5 e Mundo 6) foram particionados, ocupando espaço de armazenamento no dispositivo, de acordo com os valores apresentados na Tabela 5.3.

Dentre os ambientes internos, o Mundo 1 (Figuras 5.1(a) e 5.1(b)) é composto de 2 andares, contendo um conjunto de 40 salas alinhadas aos eixos de coordenadas e 6.199 triângulos distribuídos uniformemente na cena, não apresentando nenhum dos objetos especificados na Tabela 5.4.

Estrutura de Dados	**Mundo 1 (KB)**	**Mundo 2 (KB)**	**Mundo 3 (KB)**	**Mundo 4 (KB)**	**Mundo 5 (KB)**	**Mundo 6 (KB)**
***Grid* Irregular**	784	12.455	11.948	4.262	226	910
***Octree* Nível 3**	633	-	-	2.955	271	976
***Octree* Nível 4**	822	11.098	11.133	3.227	390	1.103
***Octree* Nível 5**	1.478	13.171	13.242	4.035	920	1.657
***Octree* Nível 6**	4.459	-	-	-	3.988	4.815
Portal *Octree* Nível 1	987	13.043	12.810	3.926	299	1.087
Portal *Octree* Nível 2	1.429	14.771	14.597	5.146	353	1.143
Portal *Octree* Nível 3	3.293	-	-	5.591	528	1.338
BSP-Tree	568	9.449	9.512	2.772	229	939

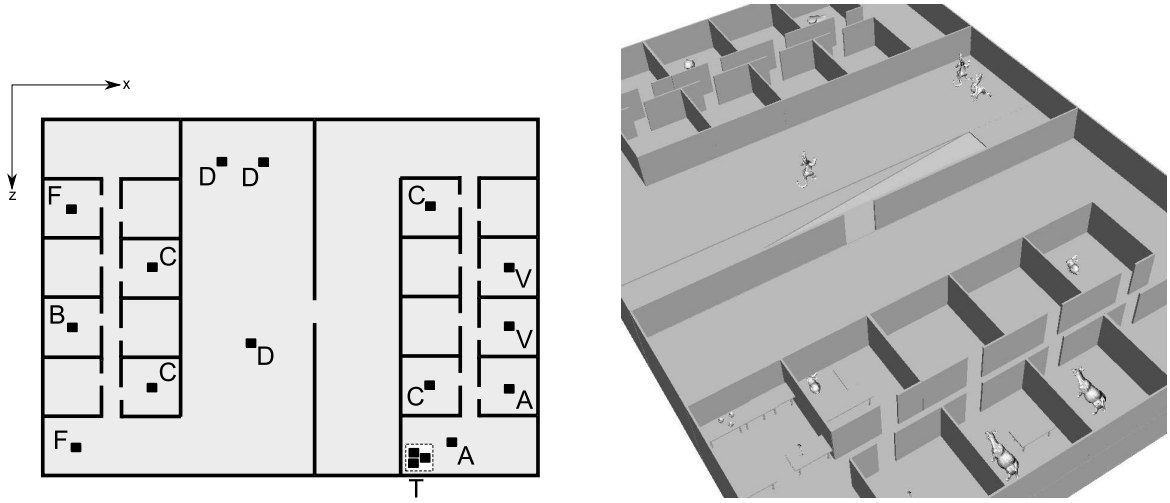
Tabela 5.3: Tamanho do arquivo final do mundo particionado utilizando as estruturas de particionamento espacial implementadas no VisMobile.



(a) (b)

Figura 5.1: Em (a), o diagrama da vista superior de um dos andares que compõe o Mundo 1 e, em (b), a visualização 3D correspondente.

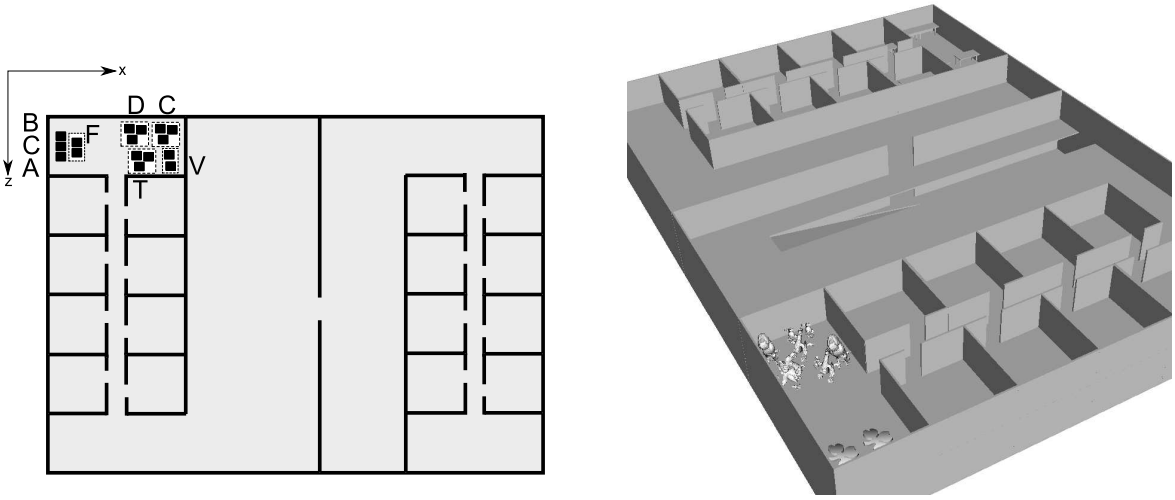
O Mundo 2 (Figuras 5.2(a) e 5.2(b)) tem como base o Mundo 1, porém, é acrescido de 17 objetos (abajur, bule, coelho, dragão, flor e vaca, os quais são especificados na Tabela 5.4). Estes objetos são uniformemente espalhados nas salas, o que resulta em uma maior complexidade geométrica, totalizando 102.232 triângulos.



(a) (b)

Figura 5.2: Em (a), o diagrama da vista superior que representa a disposição dos objetos A, B, C, D, F, V e T (mostrados na Tabela 5.4). Cada quadrado preenchido equivale a uma unidade do referido objeto. Em (b), a visualização 3D do Mundo 2.

O Mundo 3 (Figuras 5.3(a) e 5.3(b)), assim como o Mundo 2, tem como base o Mundo 1 acrescido de 17 objetos, totalizando também 102.232 triângulos. Contudo, ao contrário do Mundo 2, no Mundo 3 os objetos estão localizados em uma mesma sala do ambiente.

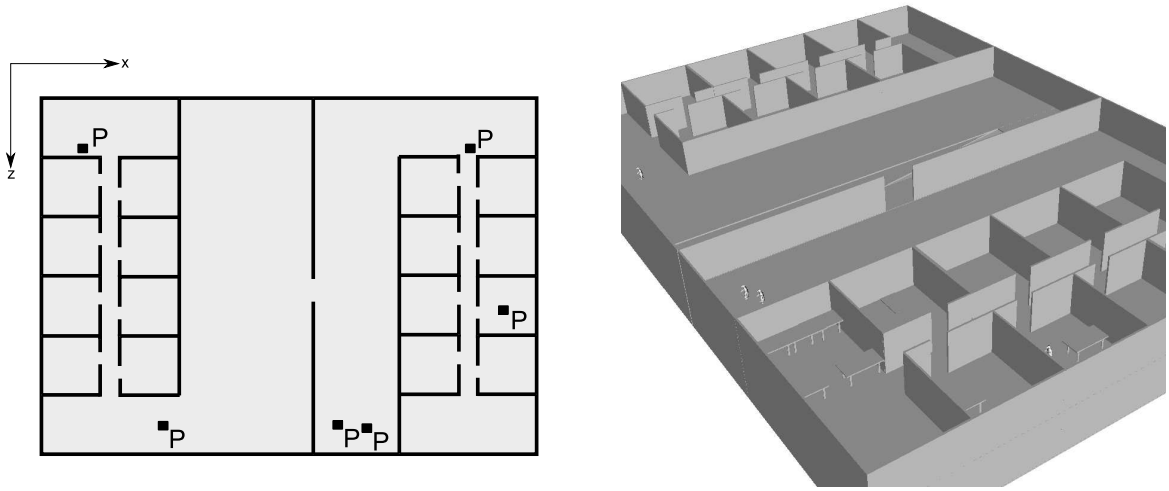


(a) (b)

Figura 5.3: Em (a), o diagrama da vista superior que representa a disposição dos objetos A, B, C, D, F, V e T (mostrados na Tabela 5.4). Cada quadrado preenchido equivale a uma unidade do referido objeto. Em (b), a visualização 3D do Mundo 3.

O Mundo 4 (Figuras 5.4(a) e 5.4(b)) também corresponde à uma extensão do Mundo 1. Entretanto, tem complexidade geométrica inferior a dos Mundos 2 e 3 e superior a do Mundo

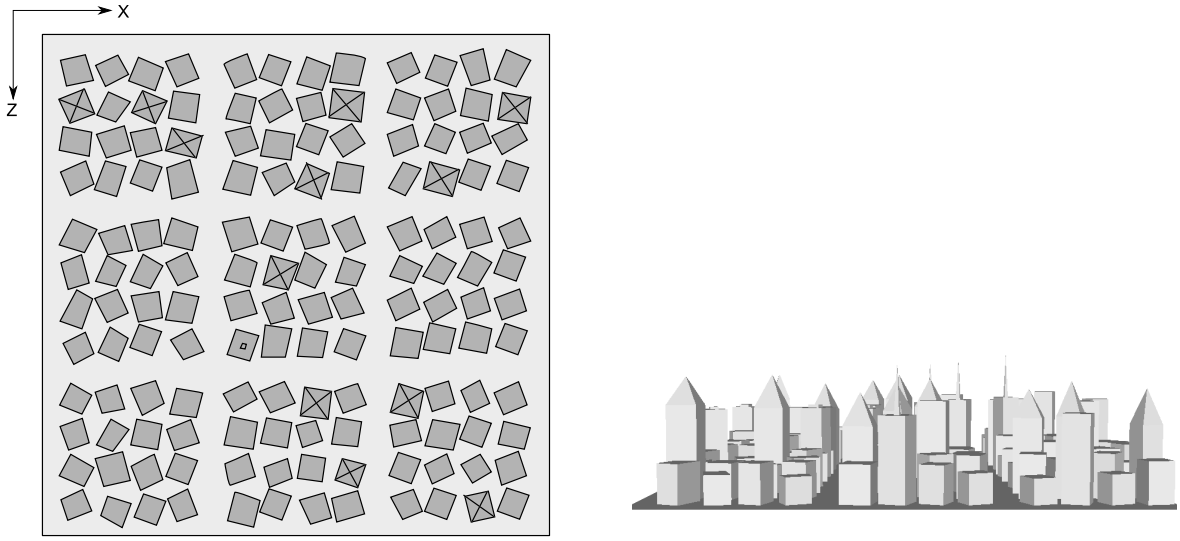
1, totalizando 30.199 triângulos. Dentre os quais, 6.199 correspondem aos triângulos do Mundo 1 e 24.000 aos que compõem os 6 objetos (pinguim) acrescentados no ambiente. Como exibido na Tabela 5.4, cada pinguim é composto por 4.000 triângulos.



(a) (b)

Figura 5.4: Em (a), o diagrama da vista superior que representa a disposição do objeto P (mostrado na Tabela 5.4). Cada quadrado preenchido equivale a uma unidade do referido objeto. Em (b), a visualização 3D do Mundo 4.

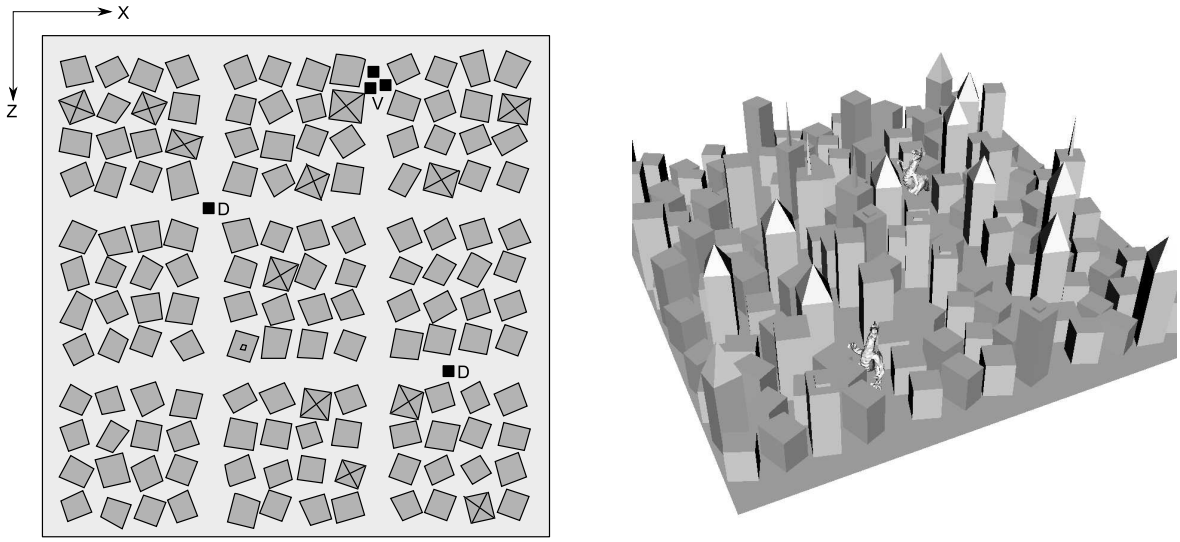
Quanto aos ambientes externos, o Mundo 5 (Figuras 5.5(a) e 5.5(b)) corresponde a um ambiente urbano, apresentando 9 quarteirões, gerado a partir do aplicativo CityGen 0.9 [57]. Este aplicativo cria um ambiente urbano via informações fornecidas em um arquivo no formato XML. Cada quarteirão, por sua vez, contém 16 edifícios. Os edifícios não estão alinhados e apresentam diferentes orientações e alturas. No total, apresenta 2.548 triângulos, uniformemente espalhados no ambiente.



(a) (b)

Figura 5.5: Em (a), diagrama da vista superior que representa o Mundo 5 e, em (b), a visualização 3D correspondente.

O Mundo 6 (Figuras 5.6(a) e 5.6(b)) é uma extensão do Mundo 5 e apresenta um número maior de objetos espalhados pelo ambiente (3 vacas e 2 dragões foram adicionados), totalizando 10.040 triângulos. Conseqüentemente, o Mundo 6 possui uma complexidade geométrica maior que a do Mundo 5.



(a) (b)

Figura 5.6: Em (a), o diagrama da vista superior que representa a disposição dos objetos D e V (mostrados na Tabela 5.4). Cada quadrado preenchido equivale a uma unidade do referido objeto. Em (b), a visualização 3D do Mundo 6.

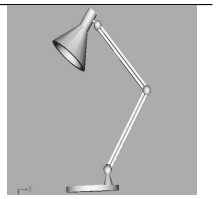
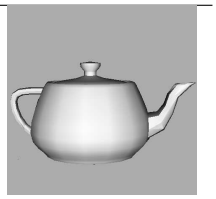
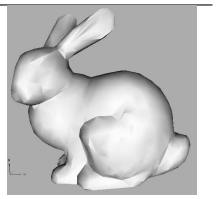
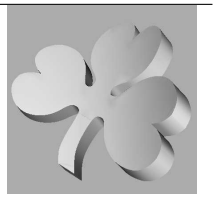
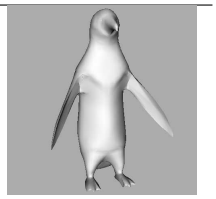
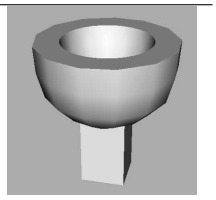
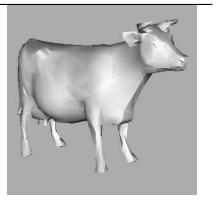
Objeto	**Nome**	**Número de Triângulos**	**Visualização**
A	abajur	600	
B	bule	1.056	
C	coelho	1.500	
D	dragão	1.500	
F	flor	5.288	
P	pinguim	4.000	
T	taça	296	
V	vaca	1.500	

Tabela 5.4: Objetos contidos nos ambientes e seus respectivos números de triângulos.

Todos os seis ambientes apresentam as seguintes características genéricas: 3 luzes direcionais, modelo de tonalização *Gouraud* e *depth buffer*[1] (disponíveis na OpenGL ES), e não apresentam texturas. O método de *backface culling* utilizado é o algoritmo presente na OpenGL ES. A câmera sintética tem seu plano distante localizado a uma distância de 1000 e seu campo de visão (*field of view*) é de 45°.

5.3 Trajetórias de Locomoção

Para a realização dos testes de análise de desempenho foram definidas e gravadas duas trajetórias de locomoção da câmera, uma para ambientes internos (contendo 1.476 quadros) e outra para os externos (contendo 1.999 quadros). A primeira é formada pelos pontos de referência A, B, C, D, E, F e G (Figura 5.7) e foi utilizada nos ambientes internos. A segunda é formada pelos pontos de referência A, B, C e D (Figura 5.8) e foi utilizada nos ambientes externos. Ambas trajetórias foram pré-gravadas para garantir um maior controle nos experimentos conduzidos, bem como para facilitar a reprodução dos testes. Nas Tabelas 5.5 e 5.6, são especificadas as principais características dos polígonos presentes no campo de visão da câmera, nos pontos de referência que compõem cada uma das duas trajetórias.

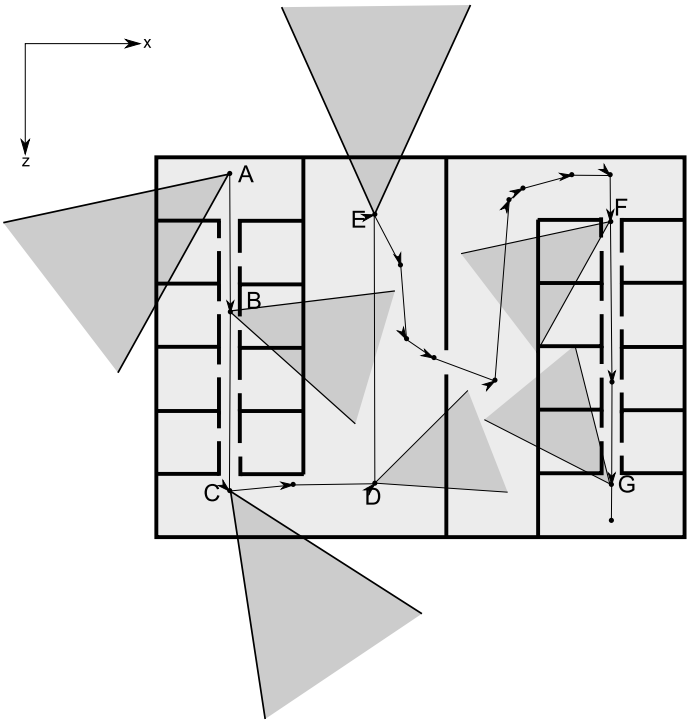


Figura 5.7: Trajetória definida para os ambientes internos.

[1] *Depth buffer* ou *Z-Buffer*, na OpenGL ES, testa e descarta os fragmentos caso a comparação de sua profundidade falhar.

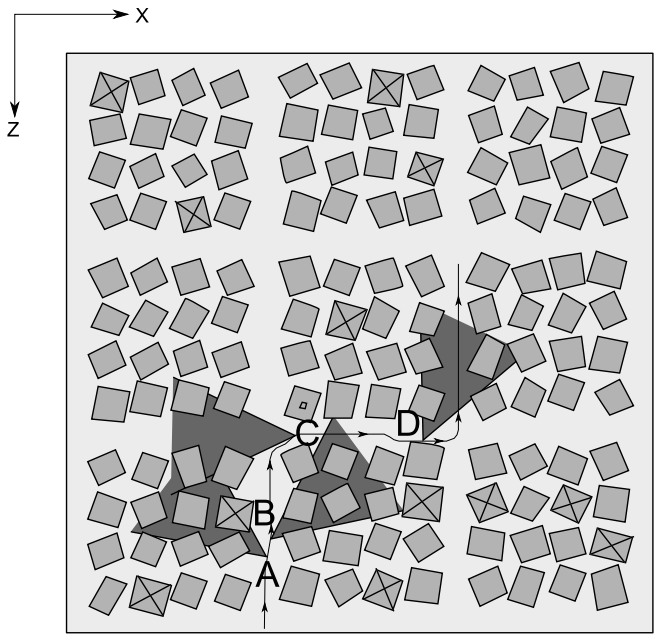


Figura 5.8: Trajetória definida para os ambientes externos.

Ambientes Internos (Mundos 1, 2, 3 e 4)		
Pontos de Referência	**Posição na Trajetória (quadro)**	**Polígonos no Campo de Visão**
A	56	Apenas os triângulos da parede da sala estão visíveis ao observador, porém, muitos triângulos estão contidos no volume de visualização
B	124	Metade da área renderizada é a parede, pode-se observar uma área aberta na outra metade
C	229	Observa-se uma área aberta, com muitos triângulos
D	290	Poucos triângulos estão visíveis ao observador, porém, muitos polígonos estão contidos no volume de visualização
E	462	Observa-se apenas a parede da sala. Apenas os polígonos da parede estão contidos no volume de visualização
F	1000	Poucos polígonos são visualizados apesar de muitos deles estarem contidos no volume de visualização
G	1413	Poucos polígonos são visualizados apesar de muitos deles estarem contidos no volume de visualização

Tabela 5.5: Detalhamento das características do campo de visão da câmera nos pontos de referência que compõem a trajetória para os ambientes internos.

Ambientes Externos (Mundos 5 e 6)		
Pontos de Referência	**Posição na Trajetória (quadro)**	**Polígonos do Campo de Visão**
A	330	Apenas 2 edifícios estão sendo visualizados e poucos triângulos estão contidos no volume de visualização
B	428	Apenas 2 edifícios estão sendo visualizados, porém, muitos triângulos estão contidos no volume de visualização
C	719	Observa-se uma área aberta, com muitos triângulos
D	1156	Poucos triângulos estão visíveis ao observador, porém, muitos triângulos estão contidos no volume de visualização

Tabela 5.6: Detalhamento das características do campo de visão da câmera nos pontos de referência que compõem a trajetória para os ambientes externos.

Para cada um dos seis ambientes, foram executados vários testes com diferentes combinações dos algoritmos de visibilidade e estruturas de particionamento espacial, visando encontrar a combinação com o melhor desempenho.

Nas próximas seções serão apresentados os resultados obtidos nos testes realizados.

5.4 Resultados

Nas próximas seções, serão detalhados os testes realizados e os resultados obtidos em cada ambiente, utilizando inicialmente o dispositivo iPaq e, em seguida, o N82. Comparações com testes realizados nos ambientes internos e externos também serão incluídas. No telefone celular N82 foram testadas somente as melhores combinações dos algoritmos de visibilidade obtidas no iPaq (um para cada um dos seis ambientes). As taxas de quadro por segundo obtidas em ambas plataformas serão então comparadas. Por fim, serão apresentadas recomendações quanto aos algoritmos de visibilidade, assim como às estruturas de particionamento espacial mais apropriada para cada um dos ambientes testados.

5.4.1 Ambientes Internos Executados no iPaq

Como descrito na Seção 5.2, foram modelados 4 ambientes internos: Mundo 1, Mundo 2, Mundo 3 e Mundo 4.

Mundo 1

Inicialmente, no Mundo 1, foram realizados oito testes de desempenho, referentes às diferentes combinações implementadas entre os algoritmos de visibilidade (Figura 5.9), incluindo um teste utilizado como referencial, sem aplicação de nenhum método de visibilidade (no caso, pré-classificado neste trabalho como o pior desempenho).

Os resultados mostram que a combinação dos algoritmos de *view-frustum culling*, *backface culling* e *backface culling* conservativo apresenta o melhor desempenho neste cenário (curva amarela na Figura 5.9), gastando em média $134,10$ milisegundos (ms) para renderizar cada quadro. Mais especificamente, todas as combinações que contêm o algoritmo de *view-frustum culling* (curvas amarela, laranja, azul escura e roxa), obtiveram resultados próximos, no máximo, com 2ms de diferença na renderização de cada quadro. Entre as outras curvas, a combinação *backface culling* com *backface culling* conservativo (curva vermelha) se destaca, obtendo o melhor desempenho entre as combinações que não contêm o algoritmo de *view-frustum culling*, em média, consumindo $192,9$ms para renderizar cada quadro. O algoritmo de *backface culling* (curva verde) quando utilizado sozinho obteve melhor desempenho que o algoritmo de *backface culling* conservativo (curva azul clara). Contudo, o algoritmo de *backface culling* em conjunto com o algoritmo de *view-frustum culling* (curva roxa), obteve um desempenho inferior ao do *backface culling* conservativo com o *view-frustum culling* (curva laranja) (desempenho inferior médio de 2ms). Quando comparados ao teste que não utilizou nenhum algoritmo de visibilidade, todos os outros testes utilizando algoritmos de visibilidade obtiveram taxas de desempenho superiores, em torno de $41,41\%$ (*view-frustum culling*, *backface culling* e *backface culling* conservativo representado pela curva amarela) e $2,36\%$ (*backface culling* conservativo, representado pela curva azul clara), no melhor e no pior caso, respectivamente.

Quanto às estruturas de dados espaciais, primeiramente, fez-se necessário a realização de testes de desempenho para identificar a melhor profundidade para cada estrutura, particularmente, para a *Octree* (foram testados os níveis 3, 4, 5 e 6) e Portal *Octree* (foram testados os níveis 1, 2 e 3). Estes testes foram baseados na combinação de algoritmos de visibilidade que obteve o melhor desempenho no Mundo 1, no caso, o algoritmo de *view-frustum culling* com *backface culling* e *backface culling* conservativo (representado pela curva amarela na Figura 5.9). Em particular, foi identificado que para a *Octree* (curva azul na Figura 5.10) a profundidade 4 obteve o melhor desempenho. Os gráficos produzidos com diferentes profundidades da Portal *Octree* não foram incluídos neste trabalho porque não apresentaram bom desempenho.

Em seguida, diferentes estruturas de particionamento espacial foram investigadas (*Octree*, Portal *Octree*, *Grid* Irregular e *BSP-Tree*). Foi identificado que a estrutura *Octree*

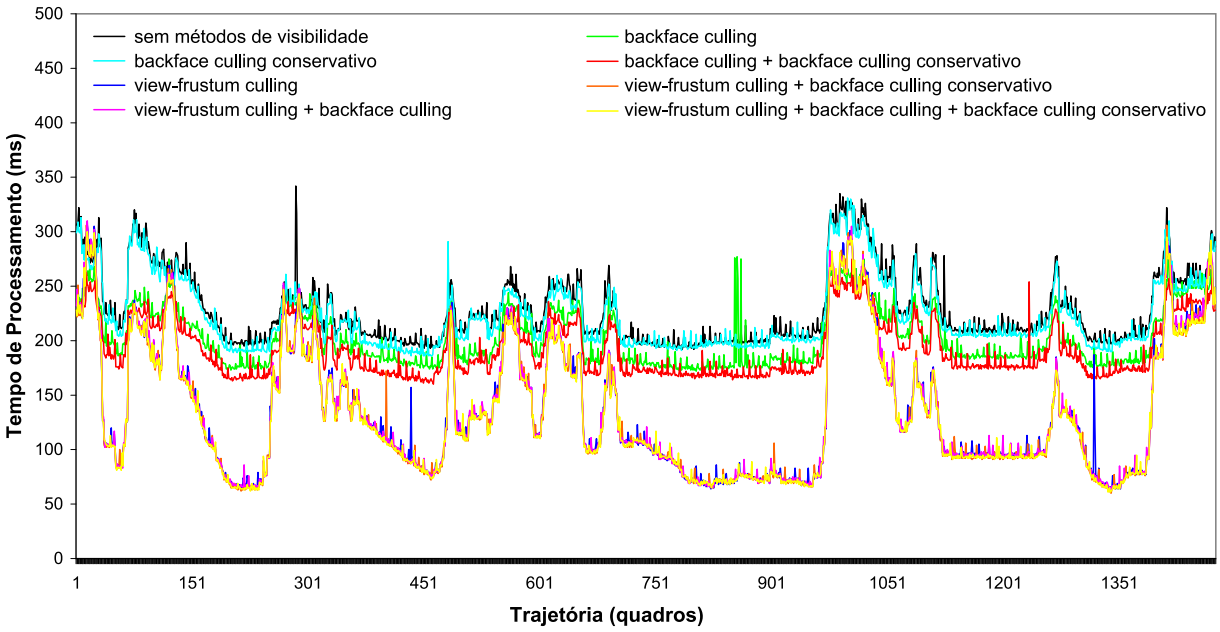


Figura 5.9: Tempo de processamento para a renderização de diferentes combinações de algoritmos de visibilidade, ao longo da trajetória no Mundo 1.

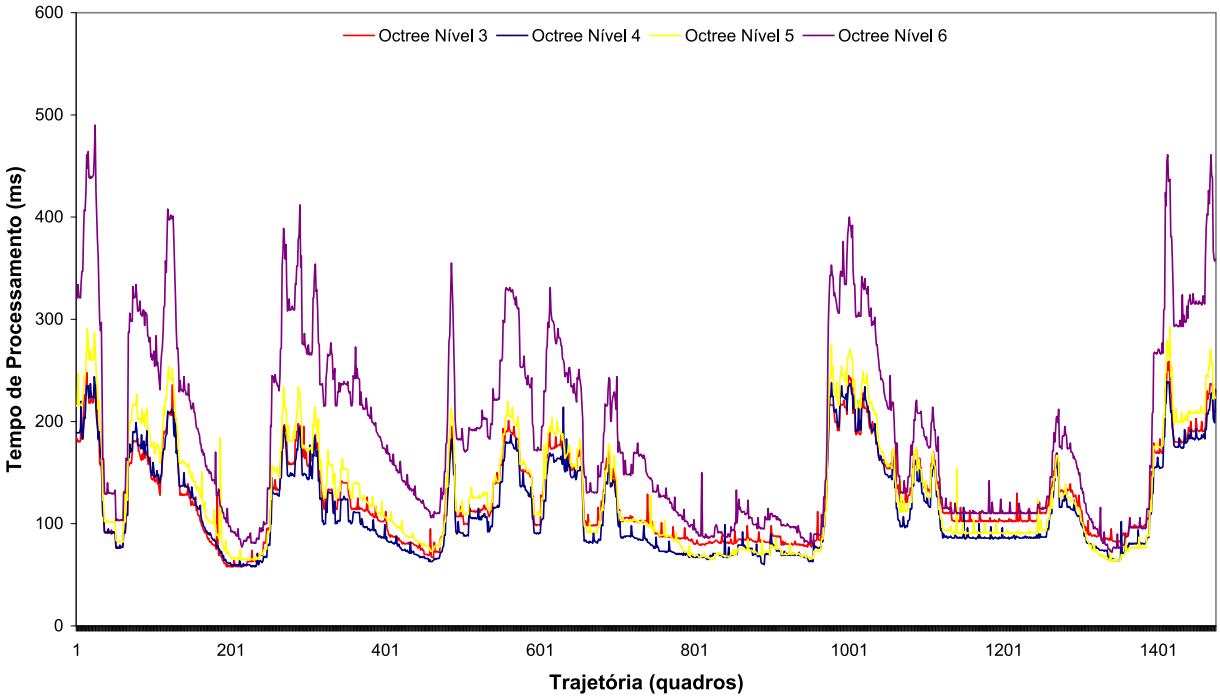


Figura 5.10: Tempo de processamento para a renderização da *Octree* com diferentes níveis de profundidade, ao longo da trajetória no Mundo 1.

com 4 níveis de profundidade (curva azul na Figura 5.11) obteve o melhor desempenho, necessitando de 114,6ms, em média, para renderizar cada quadro. A *BSP-Tree* (curva vermelha) obteve o segundo melhor desempenho, consumindo 172,3ms, em média, para renderizar cada quadro.

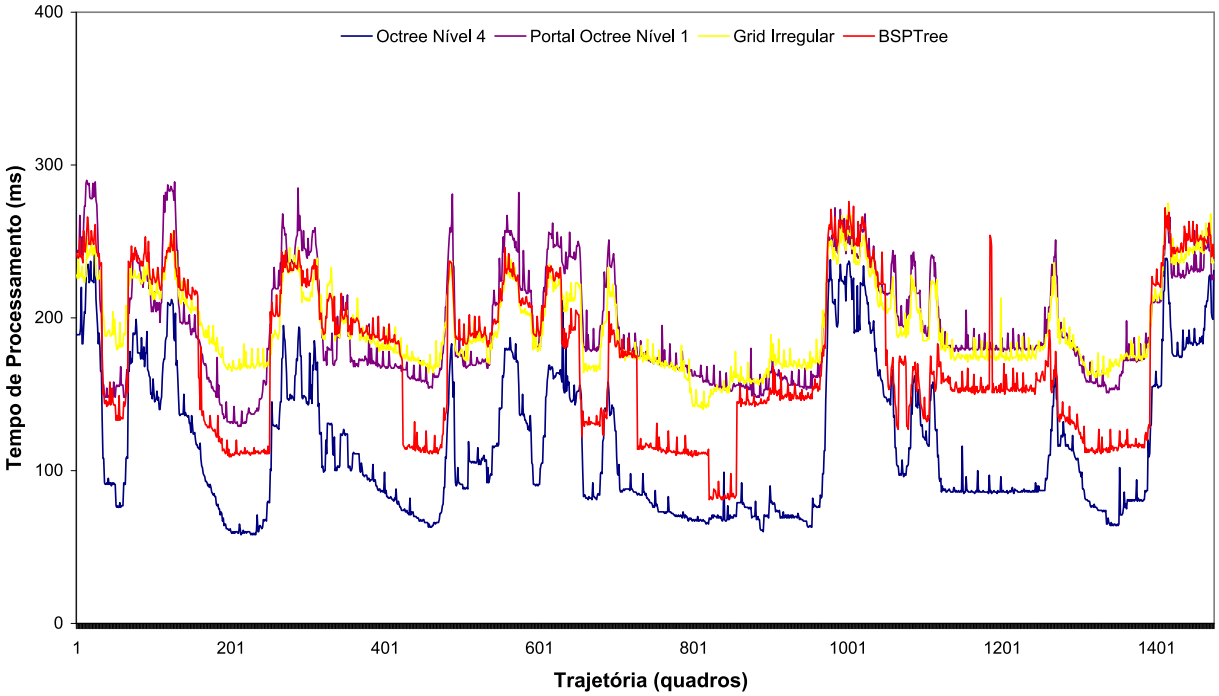


Figura 5.11: Tempo de processamento necessário utilizando diferentes estruturas de particionamento espacial, para a renderização ao longo da trajetória.

A estrutura *Octree* com 4 níveis de profundidade obteve taxas aproximadas de 4 e 17 quadros por segundo (q/s), no pior e no melhor caso, respectivamente (a taxa média foi de $10,07$q/s), como exibido na Figura 5.12. As estruturas Portal *Octree* e Grid Irregular (curvas roxa e amarela na Figura 5.12, respectivamente) obtiveram os piores desempenhos (em torno de $3,5$ e $7,5$q/s no pior e no melhor caso, respectivamente). As taxas médias obtidas pelo Portal *Octree* e pelo *Grid* Irregular foram $5,27$ e $5,34$q/s, respectivamente (curva amarela e roxa da Figura 5.12).

Neste cenário, na *Octree* e na Portal *Octree*, o tempo de processamento necessário para renderizar cada quadro está relacionado ao número de triângulos enviados ao *pipeline* de renderização (curvas azul e roxo na Figura 5.13, respectivamente) ao longo da trajetória. Pode-se observar subjetivamente que a curva que representa o tempo de processamento gasto na renderização tem formato semelhante à curva que representa o número de triângulos enviados ao *pipeline*. Mais objetivamente, a correlação entre estas duas curvas é de $+0,87$. Quanto mais próximo de +1, maior o grau de relacionamento entre essas duas variáveis, significando que o tempo de processamento aumenta proporcionalmente à medida que o número de triângulos enviados ao *pipeline* de renderização aumenta. Analisando o desempenho da *BSP-Tree* é observado que esta envia mais triângulos ao *pipeline* de renderização do que a Portal *Octree*. Apesar disso, a *BSP-Tree* ainda consegue obter um desempenho superior ao da Portal *Octree*. Este fato está relacionado ao tempo de processamento necessário para a execução dos algoritmos de

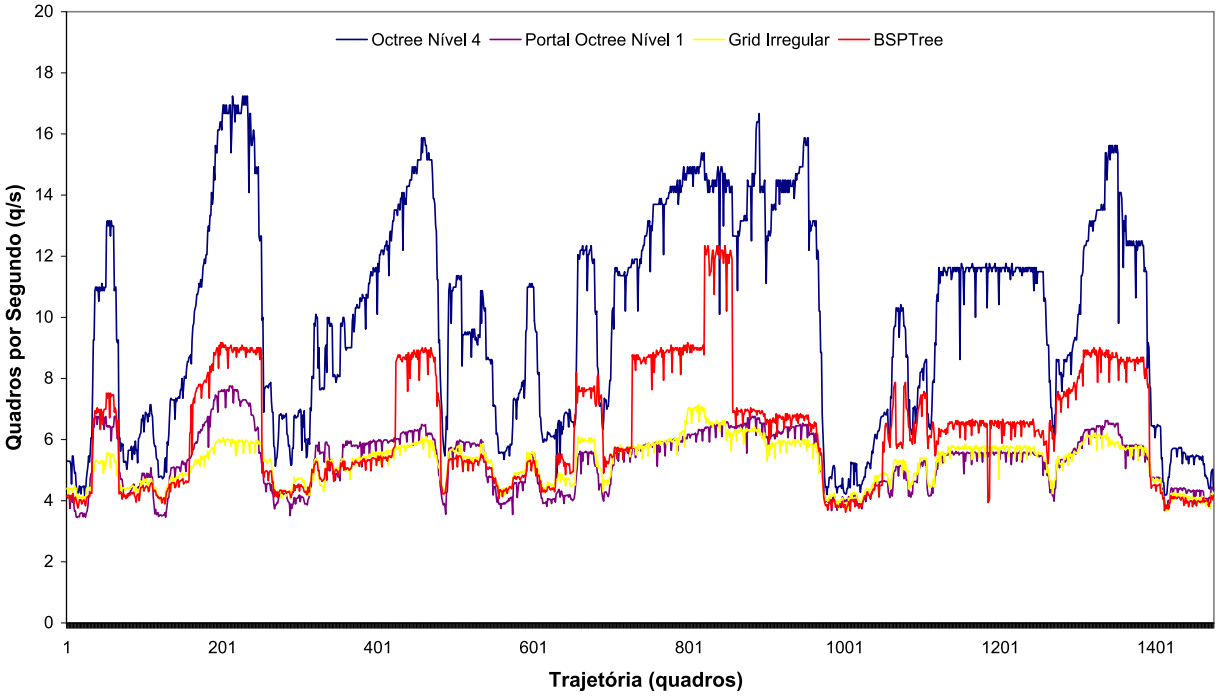


Figura 5.12: Taxa de quadros por segundo obtida ao longo da trajetória no Mundo 1.

visibilidade. Mais objetivamente, as curvas da *BSP-Tree* e Portal *Octree* obtiveram valores de correlações de $+0,88$ e $+0,65$, respectivamente. Isto significa que a curva da *BSP-Tree*, que representa a quantidade de triângulos enviados ao *pipeline*, está mais relacionada à sua curva que representa o tempo necessário para renderização, do que a Portal *Octree*.

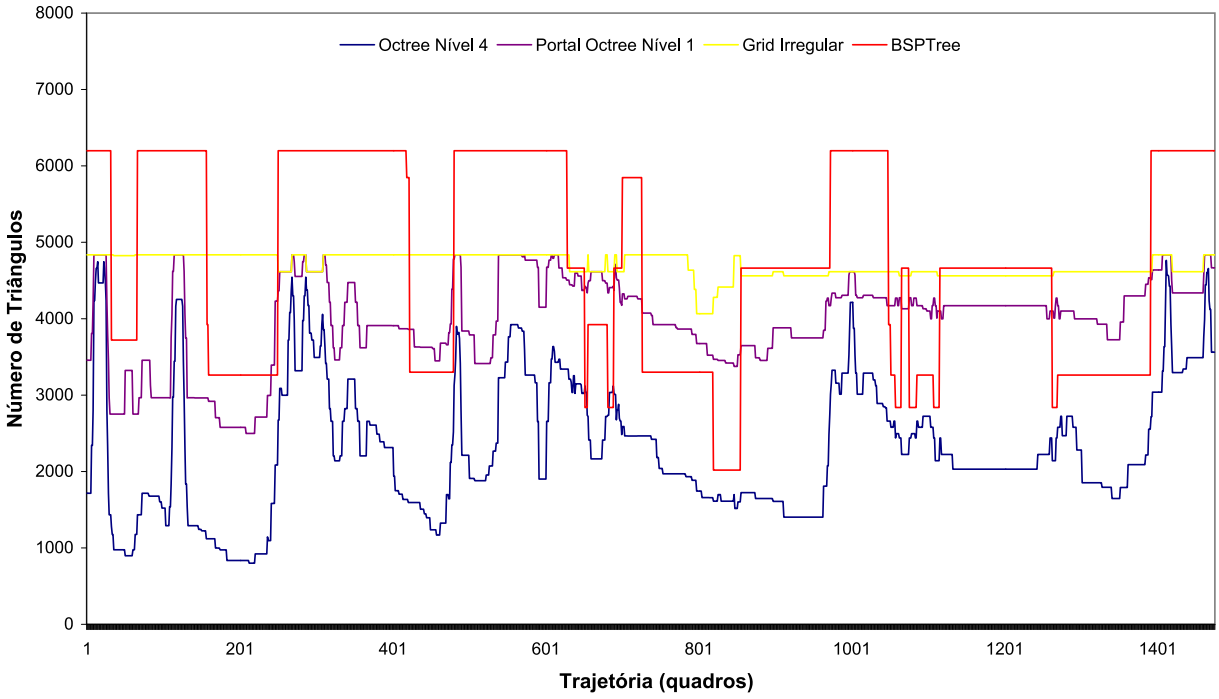


Figura 5.13: Número de triângulos enviados ao *pipeline* de renderização a cada quadro ao longo da trajetória.

Resumindo, para este ambiente a melhor combinação dos algoritmos de visibilidade foi a que utilizou o *view-frustum culling* com o *backface culling* e o *backface culling* conservativo. Quanto às estruturas, o nível de profundidade influenciou no desempenho obtido pela *Octree* e Portal *Octree*, sendo a *Octree* de nível 4 a estrutura que atingiu o melhor desempenho. Dentre as estruturas de particionamento espacial testadas, a *Octree* obteve destaque, seguida pela *BSP-Tree*. Foi também demonstrado que tanto o tempo de processamento necessário para a execução do *view-frustum culling*, quanto o número de triângulos enviados ao *pipeline* de renderização ao longo da trajetória, influenciaram no desempenho das estruturas de particionamento espacial atingindo uma taxa média aproximada de $10,07$q/s. Existe uma relação entre o número de triângulos do ambiente e o tempo de processamento gasto para a execução dos algoritmos de visibilidade. A influência de ambos depende da complexidade geométrica do ambiente.

Mundo 2

No Mundo 2, foram realizados seis experimentos dentre os algoritmos de visibilidade testados. Assim como no Mundo 1, a combinação com o *view-frustum culling*, *backface culling* e *backface culling* conservativo também obteve o melhor desempenho (curva amarela na Figura 5.14), consumindo $492,5$ms, em média, para renderizar cada quadro ao longo da trajetória. A combinação do algoritmo de *view-frustum culling* com o *backface culling* conservativo (curva laranja) atingiu o segundo melhor desempenho, obtendo uma diferença mínima de $2,18\%$ para a combinação que obteve o melhor desempenho. Assim como no Mundo 1, no Mundo 2 as combinações que obtiveram os melhores resultados contêm o algoritmo de *view-frustum culling* (curvas amarela, laranja, magenta e azul). Dentre as combinações que apresentam o *view-frustum culling*, a curva com o pior desempenho tem uma diferença média de $5,24\%$ da combinação que obteve o melhor desempenho. A combinação do *view-frustum culling* com *backface culling* (curva magenta) obteve um desempenho de $518,3$ms, sendo somente um pouco inferior ao desempenho de $517,7$ms, obtido pelo algoritmo de *view-frustum culling* sozinho (curva azul). Entretanto, em conjunto com o *backface culling* conservativo, o algoritmo de *view-frustum culling* com o de *backface culling* (curva amarela) obteve o melhor desempenho entre todas as combinações testadas ($492,5$ms, em média, para renderizar cada quadro ao longo da trajetória). Este fato é possivelmente decorrente da utilização do *depth buffer*(Z-Buffer), o qual contribuiu para descartar (mais rapidamente que o *backface culling*) triângulos que estavam afastados do observador.

Quanto às estruturas de particionamento espacial, assim como realizado no Mundo 1,

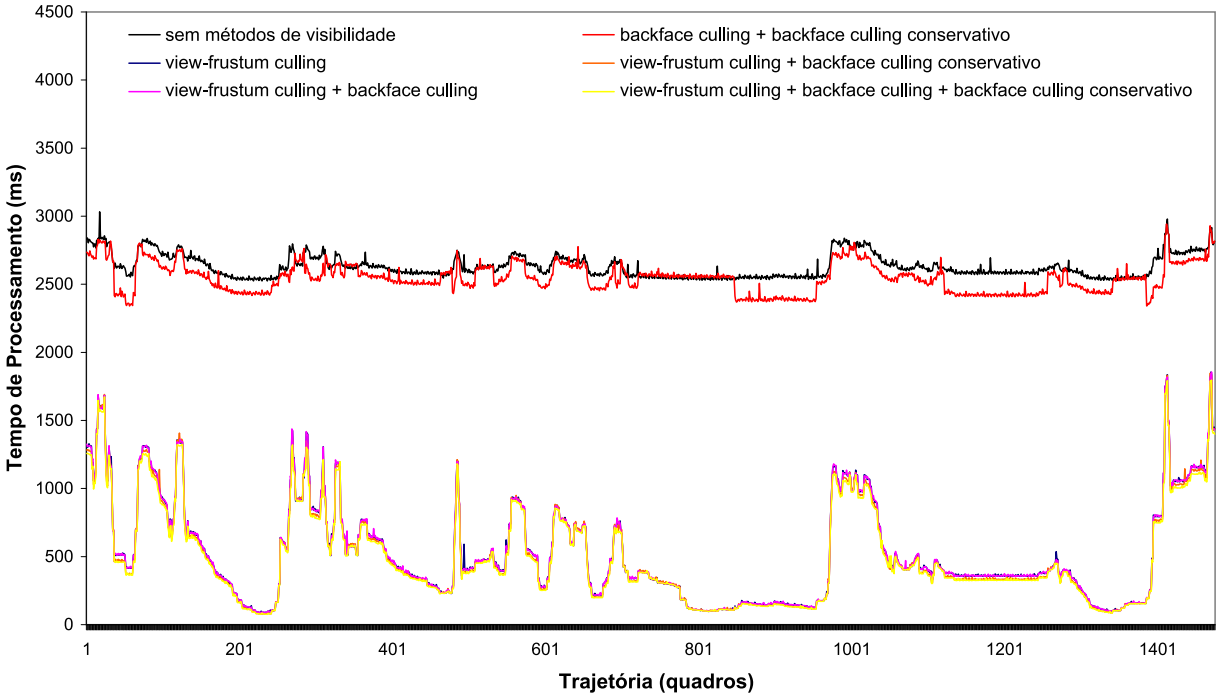


Figura 5.14: Tempo de processamento para a renderização de diferentes combinações de algoritmos de visibilidade, ao longo da trajetória no Mundo 2.

fez-se necessário a realização de testes de desempenho para identificar a melhor profundidade para cada estrutura, particularmente, para a *Octree* e Portal *Octree*. Estes testes utilizaram a melhor combinação de algoritmos de visibilidade encontrada para o Mundo 2. Em particular, foi identificado que para a *Octree* (curva azul na Figura 5.15) a profundidade foi 5 e, para a Portal *Octree*, foi 2 (os gráficos relativos aos testes realizados com a Portal *Octree* não foram incluídos neste trabalho por não apresentarem bom desempenho).

Em seguida, foram realizados testes com diferentes estruturas de particionamento espacial. Foi identificado que a estrutura *Octree* com 5 níveis de profundidade (curva azul na Figura 5.16) obteve o melhor desempenho, necessitando, $492,5$ms, em média, para renderizar cada quadro ao longo da trajetória. A Portal *Octree* com 2 níveis de profundidade (curva roxa na Figura 5.16) obteve o segundo melhor resultado, necessitando, $904,7$ms, em média, para renderizar cada quadro ao longo da trajetória.

Em se observando a taxa de quadros por segundo obtida em cada teste, a estrutura *Octree* com 5 níveis de profundidade (curva azul na Figura 5.17) obteve taxas aproximadas de $0,55$ e $13,1$q/s, no pior e no melhor caso, respectivamente (a taxa média foi de $3,53$q/s). A estrutura *Grid* Irregular (curva amarela na Figura 5.17), obteve o pior desempenho (aproximadamente $0,4$ e $0,72$q/s, no pior e no melhor caso, respectivamente, com taxas médias de $0,49$q/s).

No Mundo 2, o tempo de processamento necessário para renderizar cada quadro

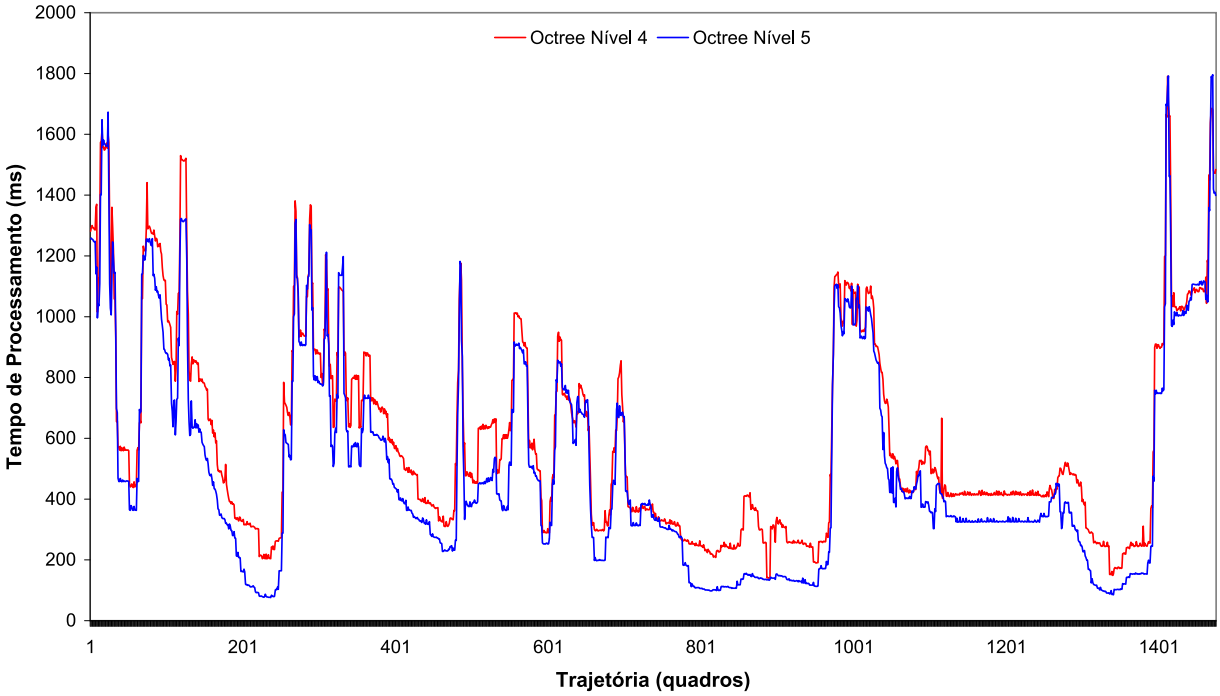


Figura 5.15: Tempo de processamento para a renderização de diferentes níveis da *Octree*, ao longo da trajetória no Mundo 2.

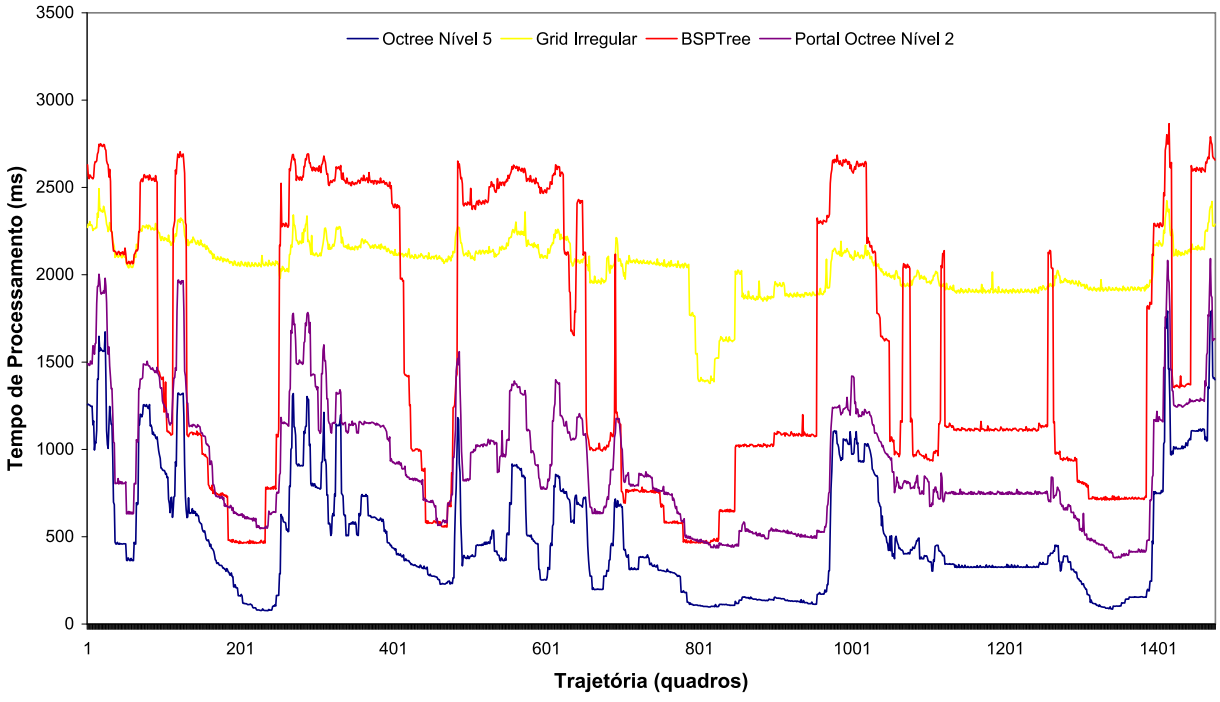


Figura 5.16: Tempo de processamento para a renderização de diferentes estruturas de dados espaciais, ao longo da trajetória no Mundo 2.

(Figura 5.16) está relacionado ao número de triângulos enviados ao *pipeline* de renderização (Figura 5.18). Pode ser observado que as curvas que representam o tempo de processamento gasto na renderização têm formatos semelhantes aos das curvas que representam o número de

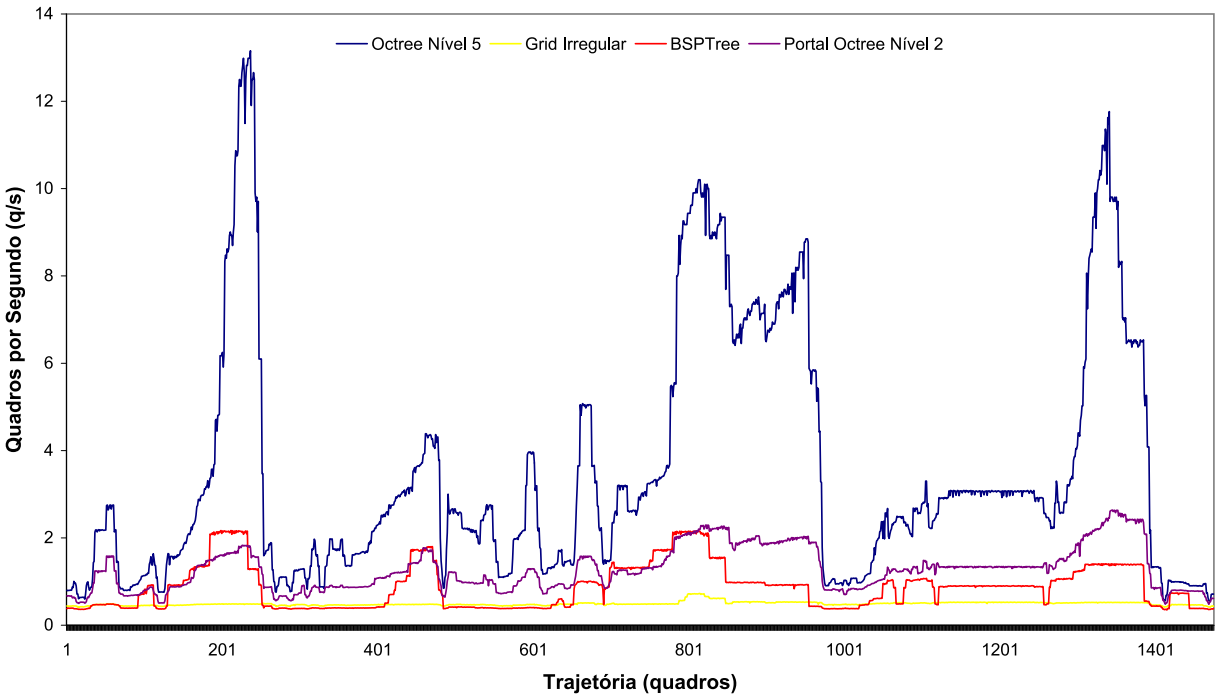


Figura 5.17: Taxa de quadros por segundo obtida ao longo da trajetória no Mundo 2.

triângulos enviados ao *pipeline*. Objetivamente, o valor calculado da correlação para *Octree* é $+0,98$. Isso significa que o tempo de processamento aumenta proporcionalmente à medida que o número de triângulos enviados ao *pipeline* de renderização aumenta.

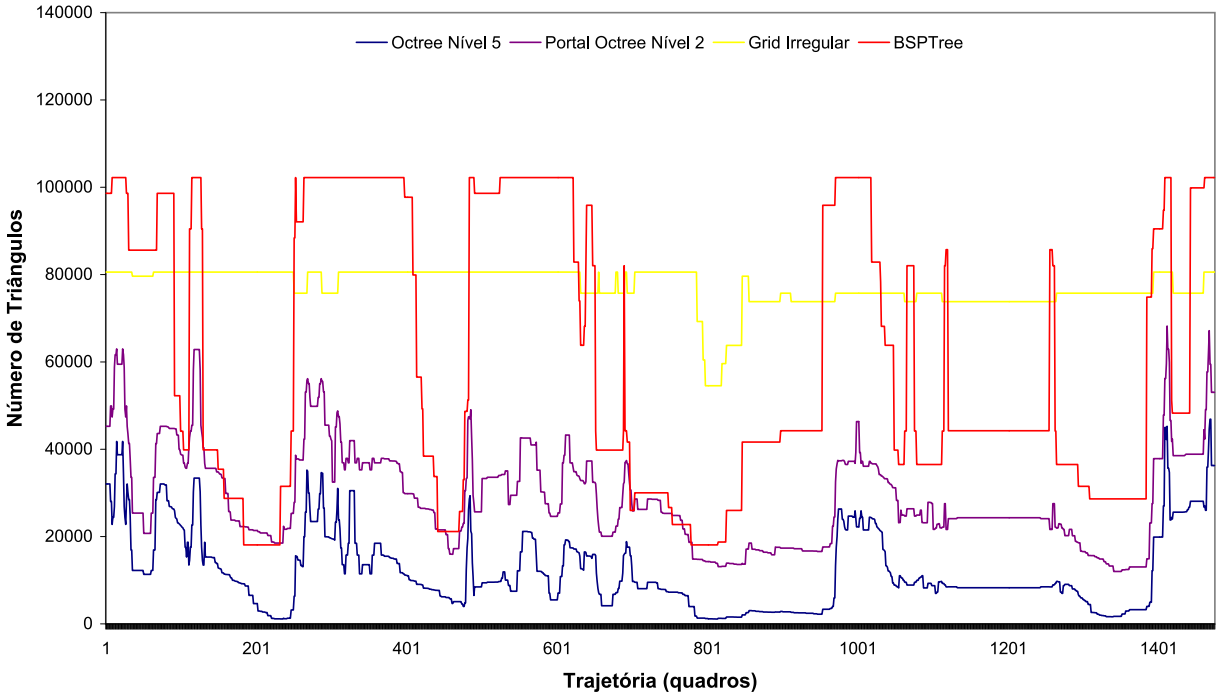


Figura 5.18: Número de triângulos enviados ao *pipeline* de renderização ao longo da trajetória no Mundo 2.

Resumindo, a melhor combinação dos algoritmos de visibilidade para o Mundo 2 foi

a que utilizou o *view-frustum culling*, com o *backface culling* e o *backface culling* conservativo. Quanto às estruturas, o nível de profundidade também influenciou no desempenho da *Octree* e da Portal *Octree*. A *Octree* nível 5 obteve o melhor desempenho (3,53q/s, em média), seguida pela Portal *Octree*. Foi também constatado que o número de triângulos enviados ao *pipeline* de renderização teve mais influência no desempenho das estruturas de particionamento espacial do que o tempo de processamento necessário para a execução do *view-frustum culling*, devido basicamente à grande quantidade de triângulos no ambiente (102.232 triângulos).

Mundo 3

No Mundo 3, foram realizados 5 testes referentes às diferentes combinações possíveis dos algoritmos de visibilidade, incluindo um teste utilizado como referencial, sem a aplicação de nenhum método de visibilidade (no caso, pré-classificado como o pior caso). Algumas combinações que não obtiveram resultados relevantes no Mundo 1 não foram testadas.

Os resultados mostram que a combinação dos algoritmos de *view-frustum culling* e *backface culling* conservativo (curva laranja na Figura 5.19) apresenta o melhor desempenho, consumindo 442,4ms, em média, para renderizar cada quadro ao longo da trajetória. Mais especificamente, todas as combinações que contêm o método de *view-frustum culling* (curvas laranja, amarela e magenta), obtiveram resultados próximos, no máximo, com 25ms de diferença na renderização de cada quadro. Entre as combinações que não contêm o *view-frustum culling*, as quais consumiram muito tempo de processamento, a combinação do *backface culling* com o *backface culling* conservativo (curva vermelha) obteve resultados 3,3% melhores aos resultados obtidos sem o uso de nenhum algoritmo de visibilidade (curva preta). A combinação do algoritmo de *view-frustum culling*, *backface culling* e *backface culling* conservativo obteve um desempenho superior ao *view-frustum culling* combinado ao *backface culling* (curvas amarela e magenta, respectivamente), consumindo 449,4 e 467,4ms, em média, para renderizar cada quadro, respectivamente. Quando comparadas ao teste que não incluiu algoritmos de visibilidade, todas as combinações que incluíram algoritmos de visibilidade obtiveram taxas de desempenho muito superiores (em torno de 480% maiores, no melhor caso).

Quanto às estruturas de dados espaciais, constatou-se que para a *Octree* o melhor desempenho foi obtido utilizando-se 5 níveis de profundidade (curva azul na figura 5.20) e, para a Portal *Octree*, 2 (os gráficos relativo aos testes realizados com a Portal *Octree* não foram incluídos neste trabalho por não terem apresentado resultados significativos).

Em seguida, foram realizados testes com diferentes estruturas de particionamento espacial. Foi identificado que a estrutura *Octree* com 5 níveis de profundidade (curva azul na

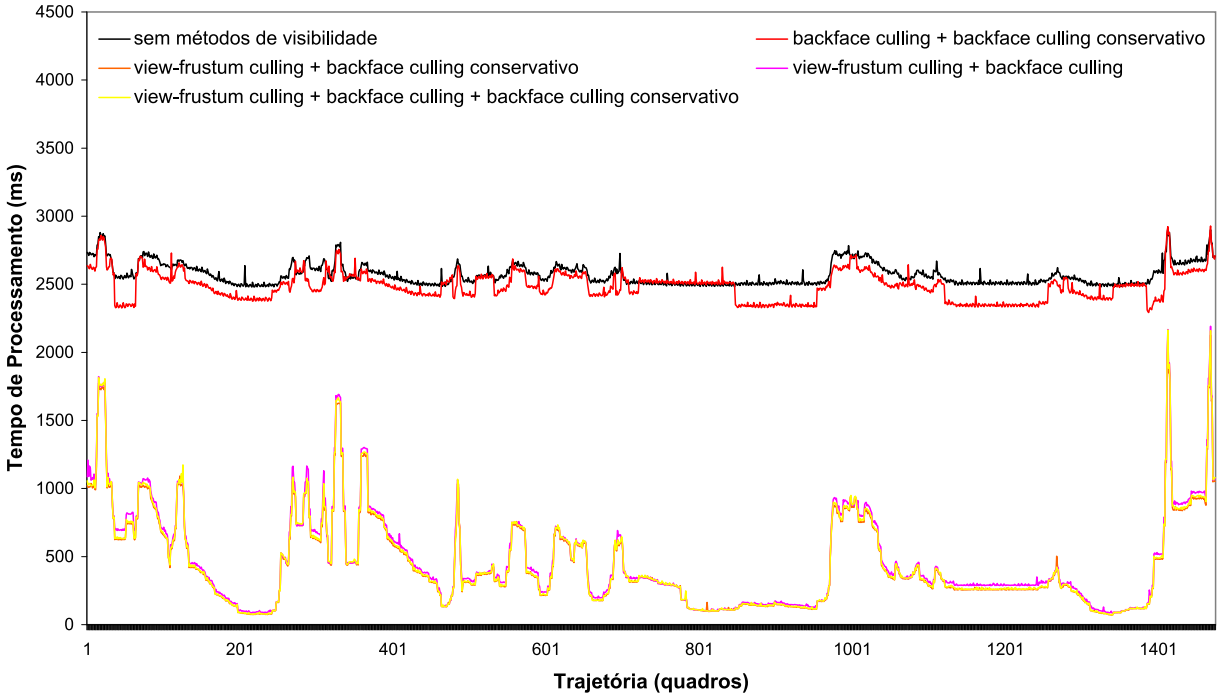


Figura 5.19: Tempo de processamento para a renderização de diferentes combinações de algoritmos de visibilidade, ao longo da trajetória no Mundo 3.

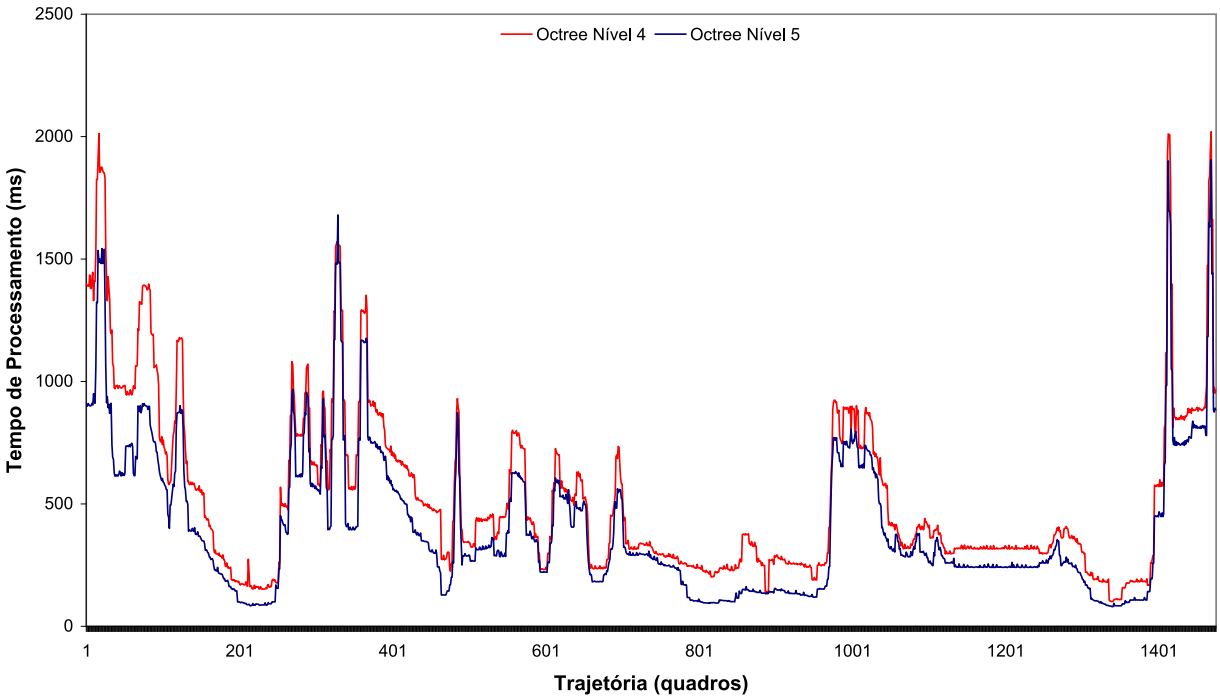


Figura 5.20: Tempo de processamento para renderização da *Octree* com diferentes níveis de profundidade, ao longo da trajetória no Mundo 3.

Figura 5.21) obteve o melhor desempenho, necessitando, 402,5ms, em média, para renderizar cada quadro ao longo da trajetória. A Portal *Octree* com 2 níveis de profundidade (curva roxa na Figura 5.21) obteve o segundo melhor desempenho, necessitando de 801,6ms, em média

para renderizar cada quadro.

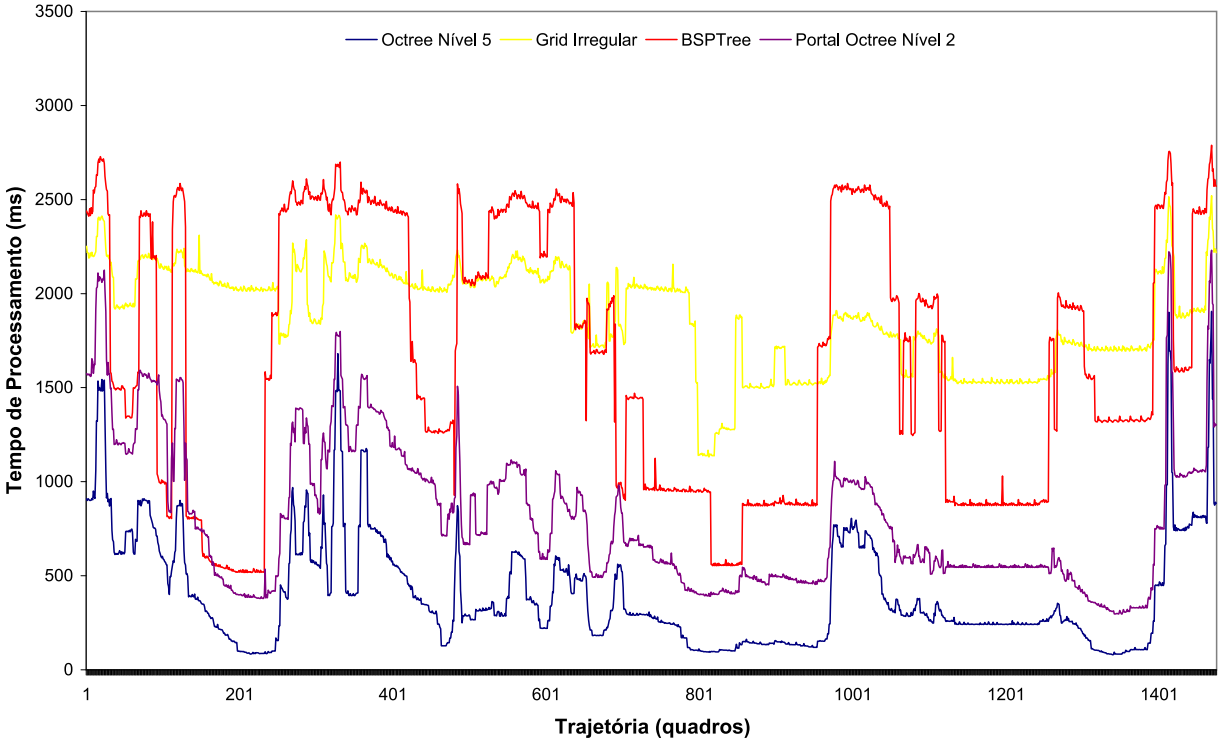


Figura 5.21: Tempo de processamento para renderização de diferentes estruturas de particionamento espacial, ao longo da trajetória no Mundo 3.

Em se observando a taxa de quadros por segundo obtida em cada teste, a estrutura Octree com 5 níveis de profundidade (curva azul na Figura 5.22) obteve taxas aproximadas de $0,5$ e $12,5$ q/s, no pior e no melhor caso, respectivamente (a taxa média foi de $4,13$ q/s). A estrutura *Grid* Irregular (curva amarela na Figura 5.22), obteve o pior desempenho (aproximadamente $0,39$ e $0,88$q/s, no pior e no melhor caso, respectivamente), com taxas médias de $0,54$ q/s.

No Mundo 3, o tempo de processamento necessário para renderizar cada quadro (Figura 5.21) está relacionado ao número de triângulos enviados ao *pipeline* de renderização (Figura 5.23). Pode ser observado subjetivamente que as curvas que representam o tempo de processamento gasto na renderização têm formato semelhante aos das curvas que representam o número de triângulos enviados ao *pipeline*. Objetivamente, o valor calculado da correlação para a *Octree* é de $+0,97$. Isso significa que o tempo de processamento aumenta proporcionalmente à medida que o número de triângulos enviados ao *pipeline* de renderização aumenta.

Diferentemente dos Mundos 1 e 2, a melhor combinação dos algoritmos de visibilidade para o Mundo 3 foi a que utilizou o algoritmo de *view-frustum culling* com o *backface culling* conservativo. Quanto às estruturas, o nível de profundidade também influenciou no

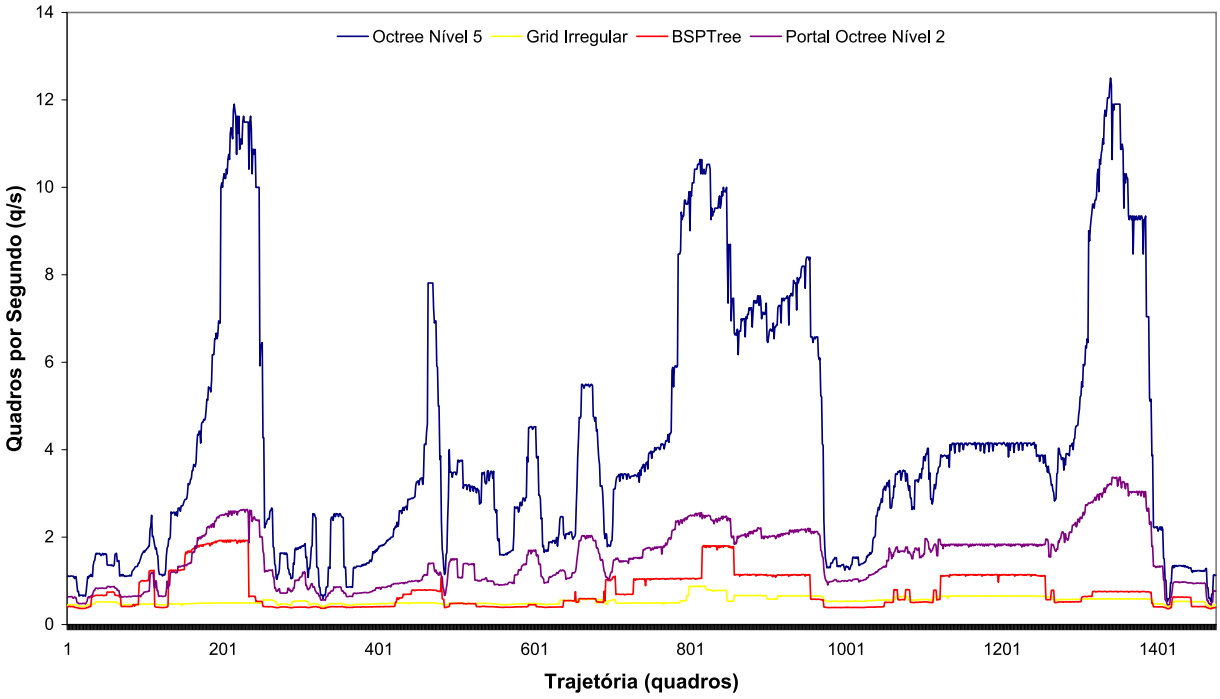


Figura 5.22: Taxa de quadros por segundo obtida ao longo da trajetória no Mundo 3.

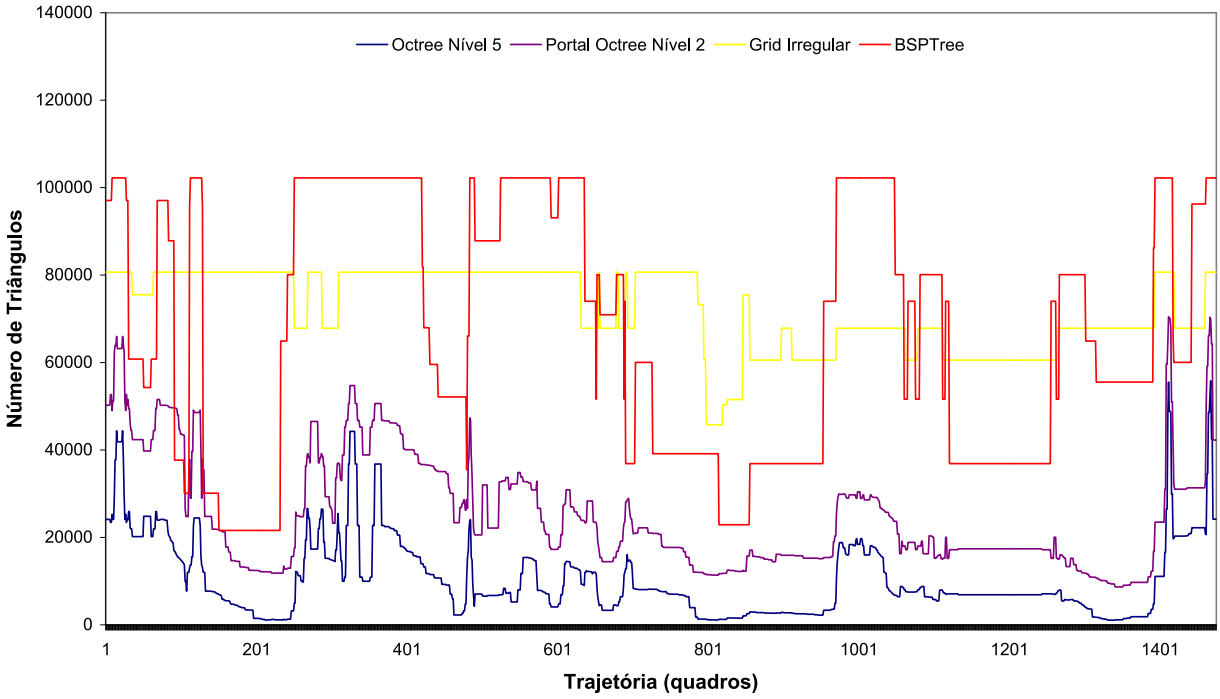


Figura 5.23: Número de triângulos enviados ao *pipeline* de renderização ao longo da trajetória no Mundo 3.

desempenho da *Octree* e da Portal *Octree*. A *Octree* com 5 níveis de profundidade obteve o melhor desempenho (4,13q/s, em média), seguida pela Portal *Octree*. Foi também constatado que o número de triângulos enviados ao *pipeline* de renderização teve mais influência no desempenho das estruturas de particionamento espacial do que o tempo de processamento necessário

para a execução do algoritmo de *view-frustum culling*.

Mundo 4

No Mundo 4, a combinação dos algoritmos de visibilidade *view-frustum culling* e *backface culling* conservativo obteve o melhor desempenho (curva laranja na Figura 5.19), necessitando de $267,8$ms, em média, para renderizar cada quadro ao longo da trajetória. A combinação dos algoritmos de *view-frustum culling*, *backface culling* e *backface culling* conservativo (curva amarela) obteve o segundo melhor desempenho, na realidade, muito próximo à melhor combinação com uma diferença mínima de $0,35\%$. Assim como nos Mundos 1, 2 e 3, no Mundo 4 as combinações que obtiveram os melhores resultados contêm o algoritmo de *view-frustum culling* (curvas amarela, laranja, magenta e azul escuro). Dentre estas as combinações, a curva que apresenta o pior desempenho (magenta), obteve uma diferença média de $5,5\%$ comparado à curva que obteve o melhor resultado (laranja). Neste ambiente, o desempenho obtido pelo algoritmo de *backface culling* (curva verde) foi similar ao desempenho do teste que não utilizou nenhum algoritmo de visibilidade (curva preta), o que indica que, para o Mundo 4, o algoritmo de *backface culling*, testado de forma independente, não contribui significativamente para a otimização do desempenho.

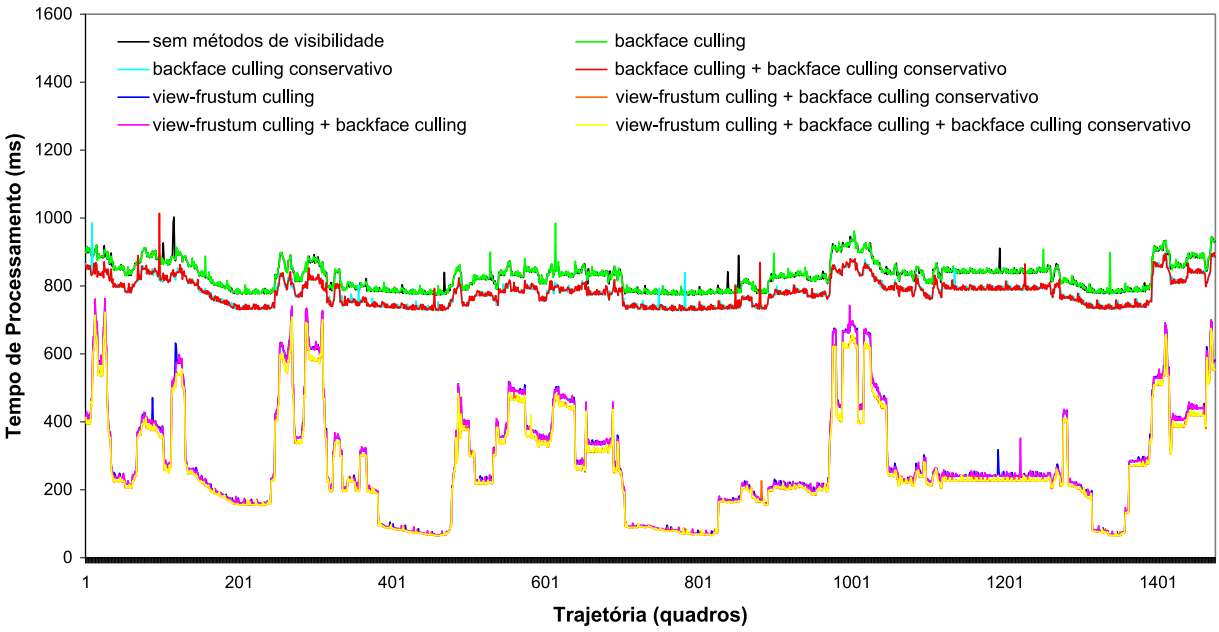


Figura 5.24: Tempo de processamento para a renderização de diferentes combinações de algoritmos de visibilidade, ao longo da trajetória no Mundo 4.

Quanto às estruturas de particionamento espacial, assim como realizado nos ambientes anteriores, no Mundo 4 (curva azul na Figura 5.25) a profundidade que obteve o melhor desempenho para a *Octree* foi 5 e, para a Portal *Octree*, 2. Novamente, a Portal *Octree* não

mostrou desempenho significativo, conseqüentemente, os gráficos relativos aos testes utilizando essa estrutura não foram incluídos neste trabalho.

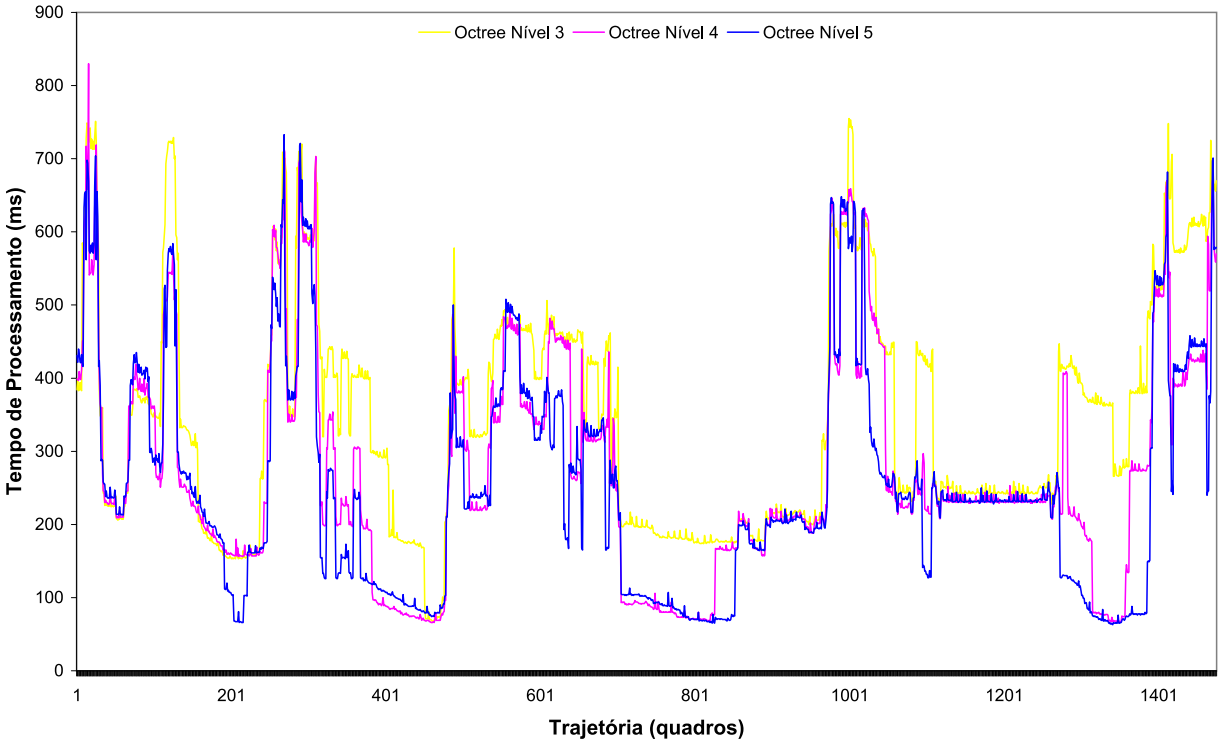


Figura 5.25: Tempo de processamento para a renderização da *Octree* com diferentes níveis, ao longo da trajetória no Mundo 4.

Testes com diferentes estruturas de particionamento espacial foram também realizados. A estrutura *Octree* com 5 níveis de profundidade (curva azul na Figura 5.26) obteve o melhor desempenho, gastando $272,01$ms, em média, para renderizar cada quadro. A *BSP-Tree* (curva vermelha na Figura 5.26) obteve o segundo melhor desempenho, necessitando, $541,8$ms, em média, para renderizar cada quadro. Como descrito anteriormente, no Mundo 4 os objetos foram posicionados propositadamente em locais específicos do ambiente com o objetivo de possivelmente balancear o número de triângulos presente no *frustum* de visualização.

A estrutura *Octree* (curva azul na Figura 5.27) obteve taxas aproximadas de $1,2$ e $15,15$q/s, no pior e no melhor caso, respectivamente (a taxa média foi de $5,32$ q/s). A estrutura *Grid* Irregular (curva amarela na Figura 5.27), obteve o pior desempenho (aproximadamente $0,94$ e $1,61$ q/s, no pior e no melhor caso, respectivamente, com taxas médias de $1,28$q/s.

O tempo de processamento necessário para renderizar cada quadro (Figura 5.26) está relacionado ao número de triângulos enviados ao *pipeline* de renderização (Figura 5.28). Subjetivamente, pode ser visualizado que as curvas que representam o tempo de processamento gasto na renderização têm formato semelhante aos das curvas que representam o número de triângu-

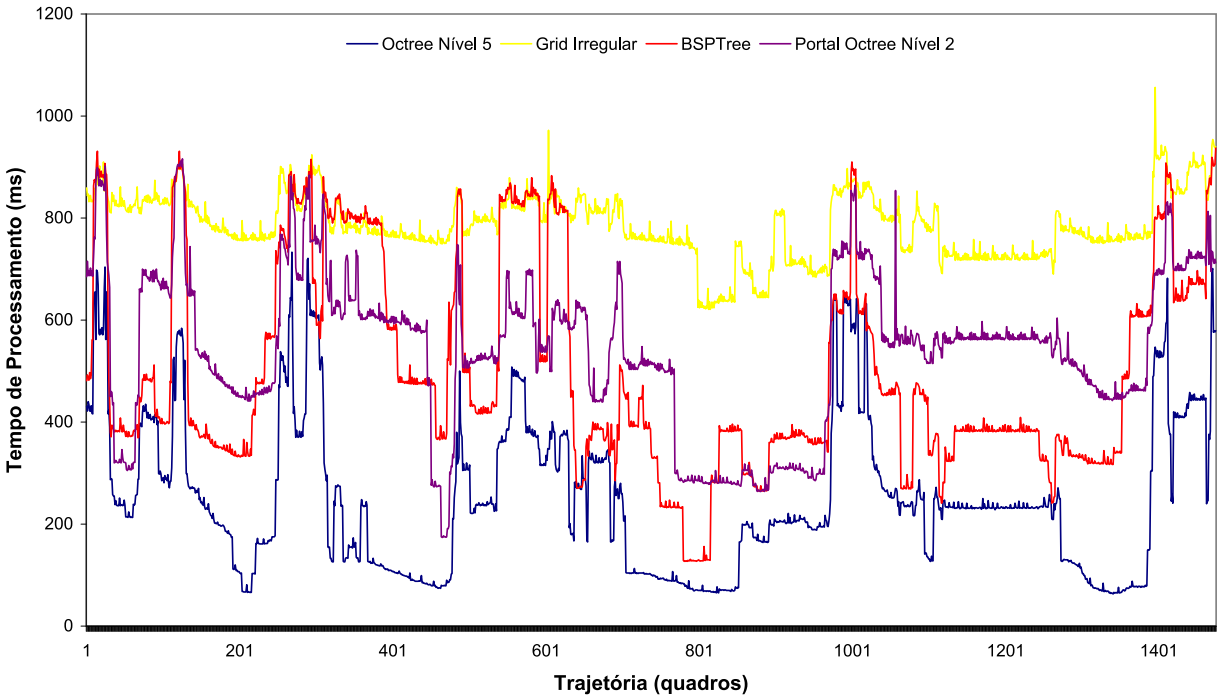


Figura 5.26: Tempo de processamento para a renderização de diferentes estruturas de dados espaciais, ao longo da trajetória no Mundo 4.

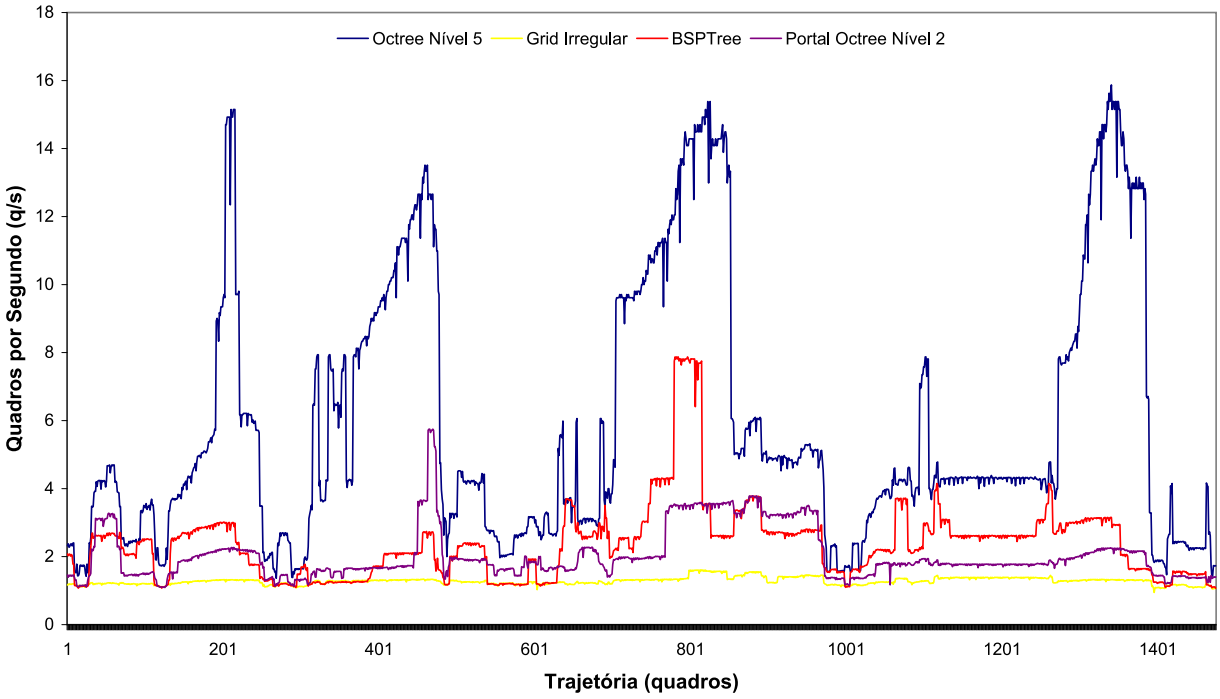


Figura 5.27: Taxa de quadros por segundo obtida ao longo da trajetória no Mundo 4.

los enviados ao *pipeline*. Nas Figuras 5.26 e 5.28, o valor calculado da correlação para a *Octree* é de $+0,95$. Ou seja, o tempo de processamento aumenta proporcionalmente à medida que o número de triângulos enviados ao *pipeline* de renderização aumenta.

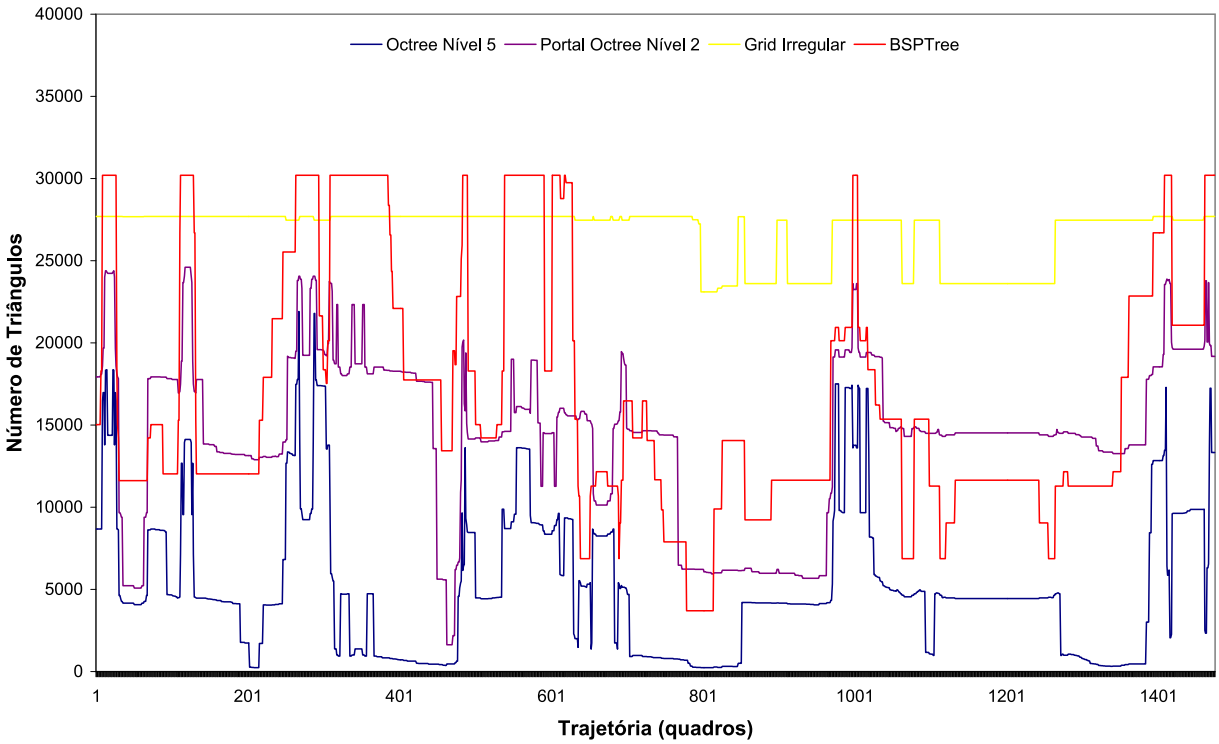


Figura 5.28: Número de triângulos enviados ao *pipeline* de renderização ao longo da trajetória no Mundo 4.

Assim como no Mundo 3, a melhor recomendação dos algoritmos de visibilidade para o Mundo 4 foi a que combinou os algoritmos de *view-frustum culling* e *backface culling* conservativo. Quanto às estruturas, o nível de profundidade também influenciou no desempenho da *Octree* e da Portal *Octree*. A *Octree* com 5 níveis de profundidade obteve o melhor desempenho (em média $5,33$q/s), seguida pela *BSP-Tree*. O número de triângulos enviados ao *pipeline* de renderização teve mais influência no desempenho das estruturas de particionamento espacial, do que o tempo de processamento para a execução do algoritmo de *view-frustum culling*.

5.4.2 Ambientes Externos Executados no iPaq

Conforme apresentado na Seção 5.2, foram modelados 2 ambientes externos: Mundo 5 e Mundo 6.

Mundo 5

No Mundo 5, dentre os algoritmos de visibilidade testados, a combinação com o *view-frustum culling* obteve o melhor desempenho (curva azul escuro na Figura 5.29), consumindo $146,02$ms, em média, para renderizar cada quadro ao longo da trajetória. A combinação do *view-frustum culling* com o *backface culling* conservativo (curva laranja) obteve o segundo me-

lhor desempenho, muito próximo ao anterior com uma diferença média de 0,58%. Assim como nos ambientes internos testados anteriormente, no Mundo 5 as combinações que obtiveram os melhores resultados contêm o algoritmo de *view-frustum culling* (curvas amarela, laranja, magenta e azul escuro). Entre as combinações que apresentam o algoritmo de *view-frustum culling*, a que obteve o resultado menos competitivo atingiu um valor de desempenho inferior a 2,07%, em relação à melhor combinação.

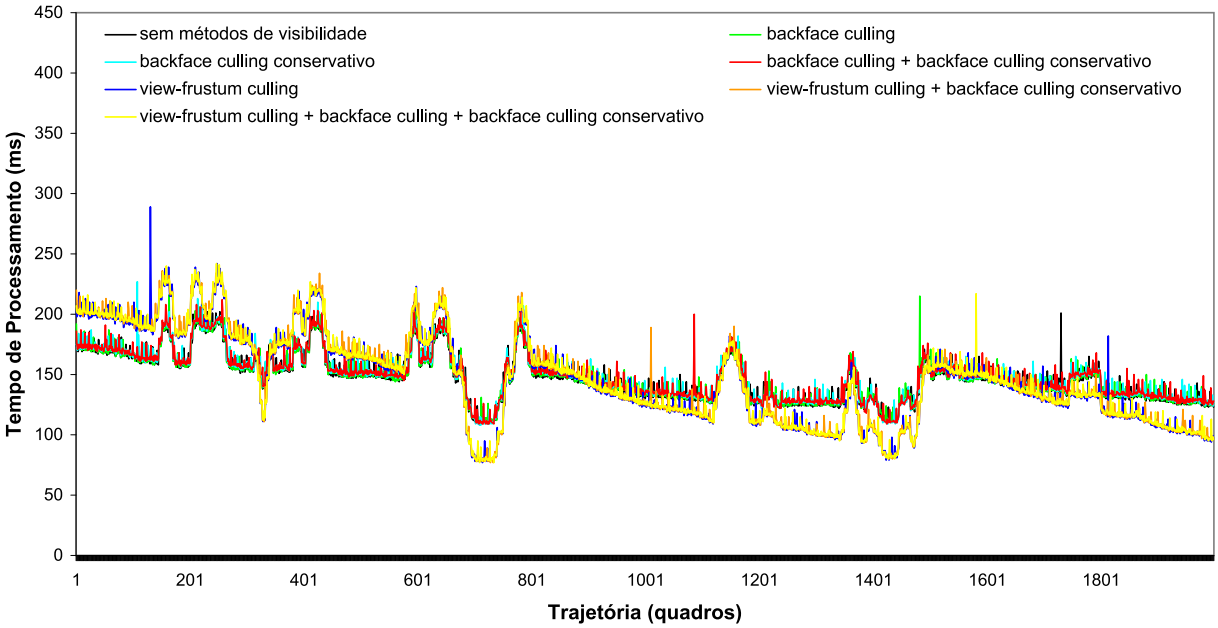


Figura 5.29: Tempo de processamento para a renderização de diferentes combinações de algoritmos de visibilidade, ao longo da trajetória no Mundo 5.

Similarmente, nos testes conduzidos utilizando-se *Octree* e Portal *Octree*, foi observado que a *Octree* com profundidade 3 seria a melhor calibragem para esta estrutura (Figura 5.30). Entretanto, diferentemente do observado, para os ambientes internos, a estrutura *Grid* Irregular (curva amarela na Figura 5.31) obteve o melhor desempenho, consumindo 90,27ms, em média, para renderizar cada quadro. A *Octree* com 3 níveis de profundidade (curva azul na Figura 5.31) obteve o segundo melhor desempenho, necessitando, 108,06ms, em média, para renderizar cada quadro.

A estrutura *Grid* Irregular (curva amarela na Figura 5.32) obteve taxas aproximadas de 5,68 e 13,51 q/s, no pior e no melhor caso, respectivamente (a taxa média foi de 11,02q/s). A estrutura Portal *Octree* com nível de profundidade 1 (curva roxa na Figura 5.32) obteve o pior desempenho (aproximadamente 5,95 e 9,61q/s, no pior e no melhor caso, respectivamente, com taxas médias de 7,91q/s).

No Mundo 5, para a estrutura *Octree* nível 3, o tempo de processamento necessário para renderizar cada quadro (Figura 5.31) está relacionado ao número de triângulos enviados ao

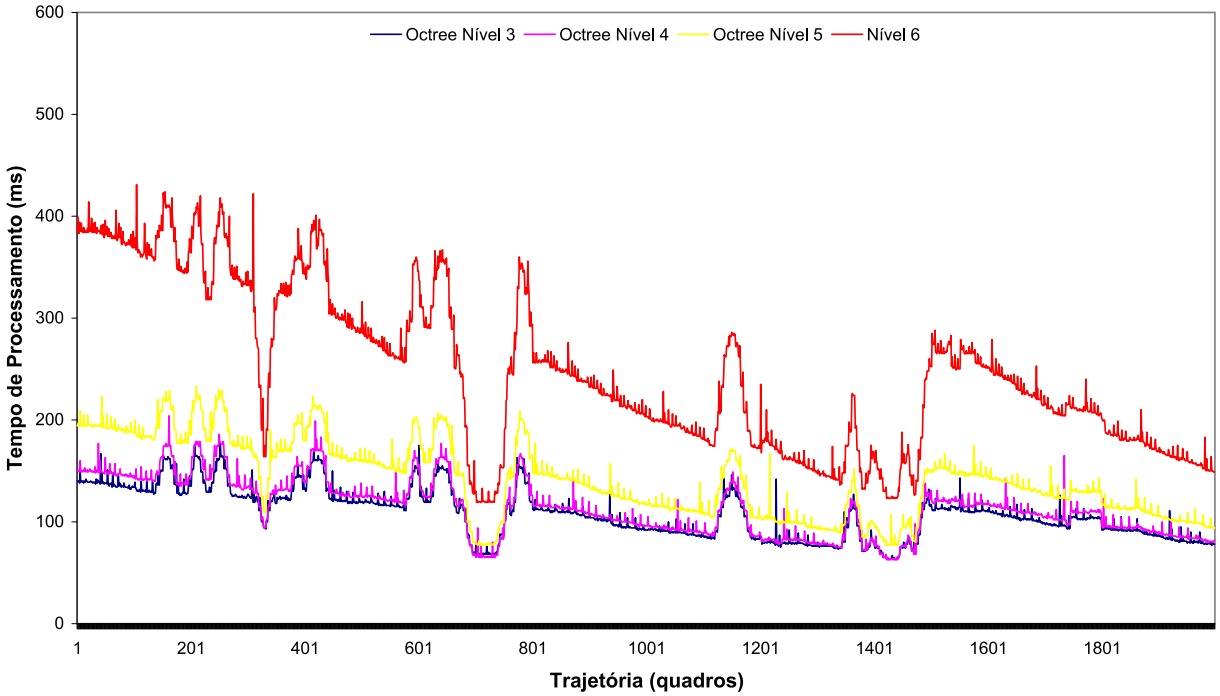


Figura 5.30: Tempo de processamento para a renderização de diferentes níveis da *Octree*, ao longo da trajetória no Mundo 5.

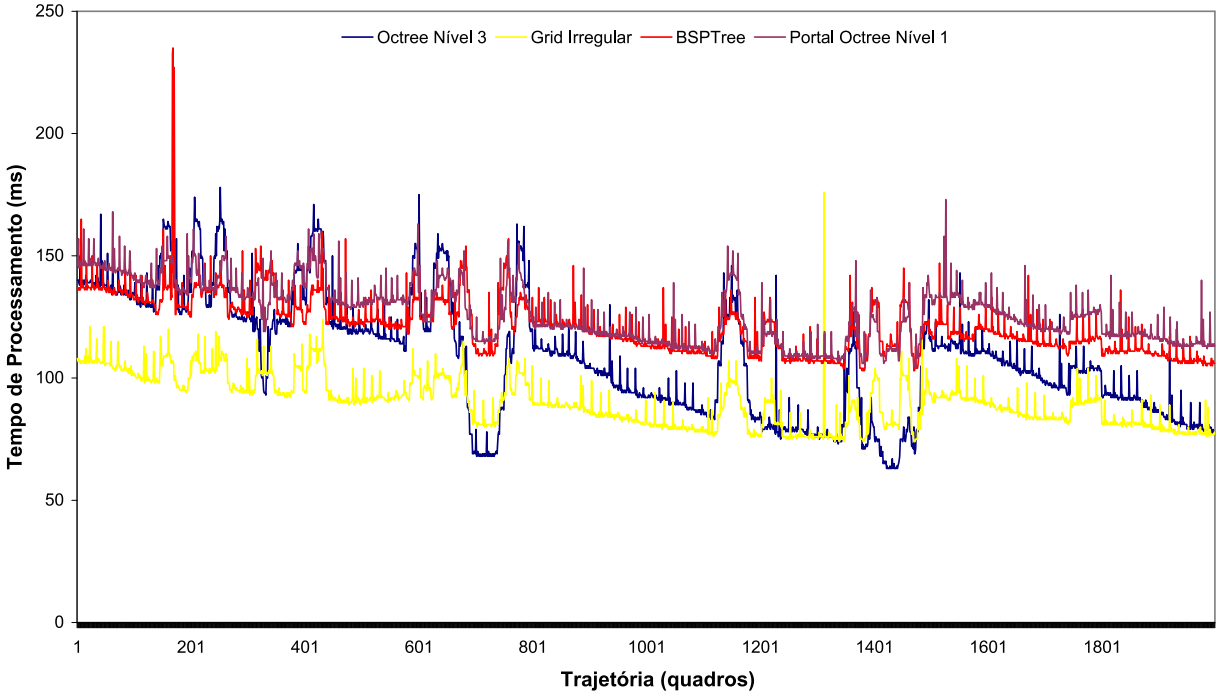


Figura 5.31: Tempo de processamento para a renderização de diferentes estruturas de particionamento espacial, ao longo da trajetória no Mundo 5.

pipeline de renderização (Figura 5.33) e ao tempo de processamento necessário para a execução dos métodos de visibilidade. Entretanto, nos experimentos conduzidos tal comportamento não pode ser observado pelas demais estruturas (*Grid* Irregular, BSP-Tree e Portal *Octree* nível 1).

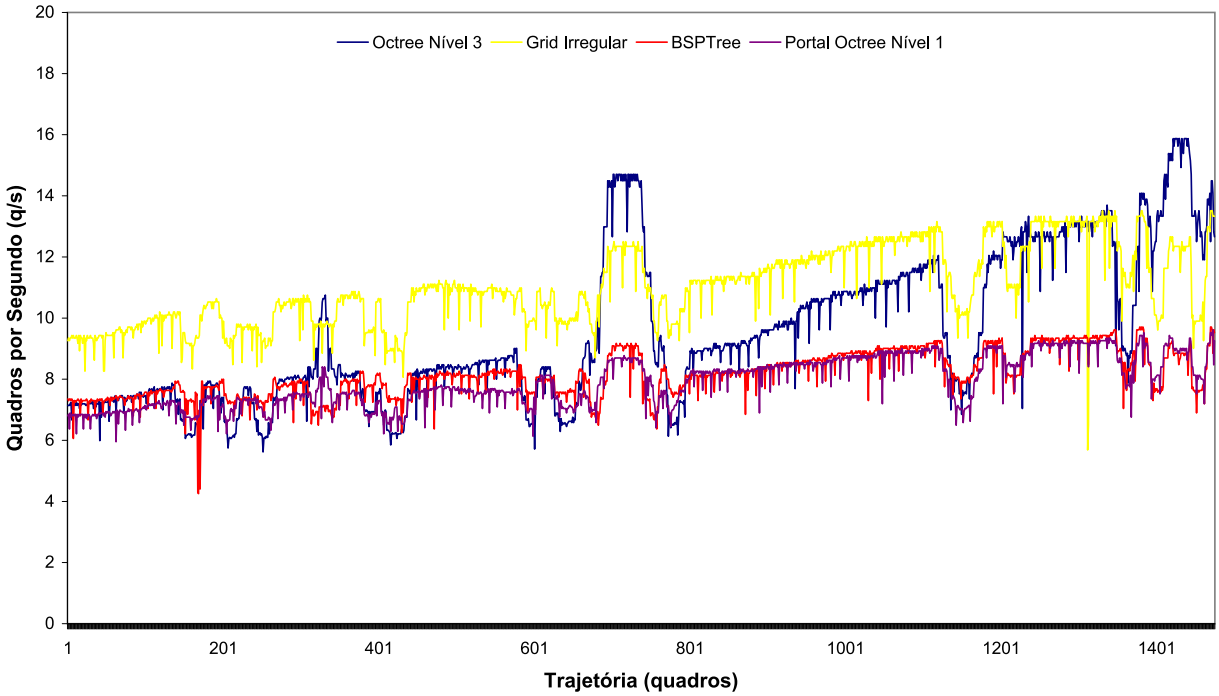


Figura 5.32: Taxa de quadros por segundo obtida ao longo da trajetória no Mundo 5.

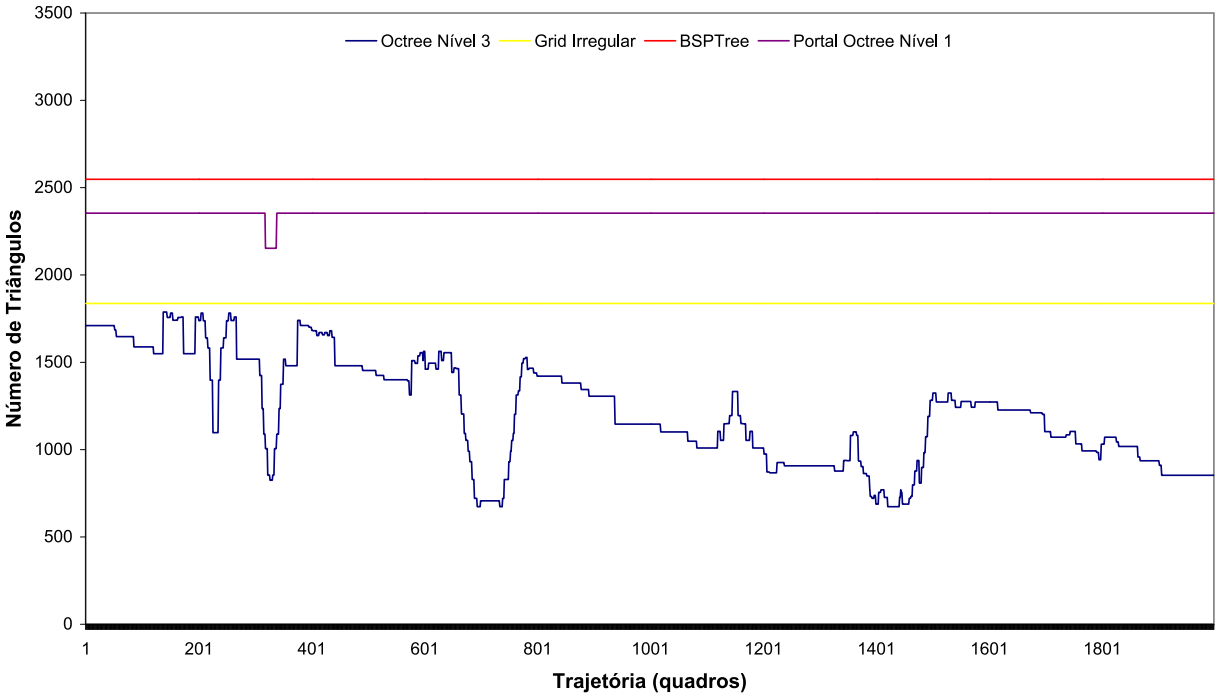


Figura 5.33: Número de triângulos enviados ao *pipeline* de renderização ao longo da trajetória no Mundo 5.

Finalmente, a melhor recomendação de algoritmo de visibilidade para o Mundo 5 foi o *view-frustum culling*. Quanto às estruturas, os níveis de profundidade influenciaram no desempenho da *Octree* e da Portal *Octree*, porém, não produziram resultados competitivos o bastante para ultrapassar o desempenho do *Grid* Irregular. O *Grid* Irregular obteve o melhor

desempenho (em média, 11,02q/s), seguido da *Octree* com 3 níveis de profundidade. Assim como no Mundo 1, tanto o número de triângulos enviados ao *pipeline* de renderização, quanto o tempo de processamento para a execução do *view-frustum culling,* influenciaram na diferença de desempenho das estruturas de particionamento espacial.

Mundo 6

Dentre os algoritmos de visibilidade testados no Mundo 6, a combinação com o *view-frustum culling* e *backface culling* conservativo obteve o melhor desempenho (curva laranja na Figura 5.34), consumindo 224,09ms, em média, para renderizar cada quadro. A combinação dos algoritmos de visibilidade *view-frustum culling*, *backface culling* e *backface culling* conservativo (curva amarela) obteve o segundo melhor desempenho, praticamente idêntico ao da combinação dos algoritmos de *view-frustum culling* e *backface culling*, com uma diferença média mínima de 0,01%. Como nos ambientes anteriormente testados, no Mundo 6 as combinações que obtiveram os melhores resultados contêm o algoritmo de *view-frustum culling* (curvas amarela, laranja, magentae azul). Entre as combinações que contêm o *view-frustum culling,* o pior desempenho foi obtido pelo algoritmo de *view-frustum culling* sozinho (curva azul), sendo aproximadamente 2,25% inferior aos melhores desempenhos obtidos pelas combinações: *view-frustum culling* com *backface culling* conservativo (curva laranja) e; *view-frustum culling* com *backface culling* e *backface culling* conservativo (curva amarela). A combinação do *view-frustum culling* com *backface culling* (curva magenta) obteve um desempenho semelhante ao do algoritmo de *view-frustum culling* (curva azul) sozinho. Para o processamento, o primeiro gastou 229,14ms e, o segundo, 228,82.

Para a *Octree* (curva azul na Figura 5.33) a melhor profundidade foi 4 e, para a Portal *Octree,* 3. A estrutura *Octree* com 4 níveis de profundidade (curva azul na Figura 5.36) obteve o melhor desempenho, gastando, 206,5ms para renderizar cada quadro ao longo da trajetória. A *BSP-Tree* (curva vermelha na Figura 5.36) obteve o segundo melhor desempenho, necessitando de 255,6ms, em média, para renderizar cada quadro.

A estrutura *Octree* (curva azul na Figura 5.37) obteve taxas aproximadas de 2,59 e 15,87q/s, no pior e no melhor caso, respectivamente (a taxa média foi de 5,59 q/s). A estrutura *Grid* Irregular (curva amarela na Figura 5.37) obteve o pior desempenho (aproximadamente 2,45 e 3,67q/s, no pior e no melhor caso, respectivamente, com taxas médias de 3,1q/s).

No Mundo 6, o tempo de processamento para renderizar cada quadro (Figura 5.36) está relacionado ao número de triângulos enviados ao *pipeline* de renderização (Figura 5.38) como pode ser observado nas estruturas BSP-Tree, *Octree* nível 4 e Portal *Octree* nível 3.

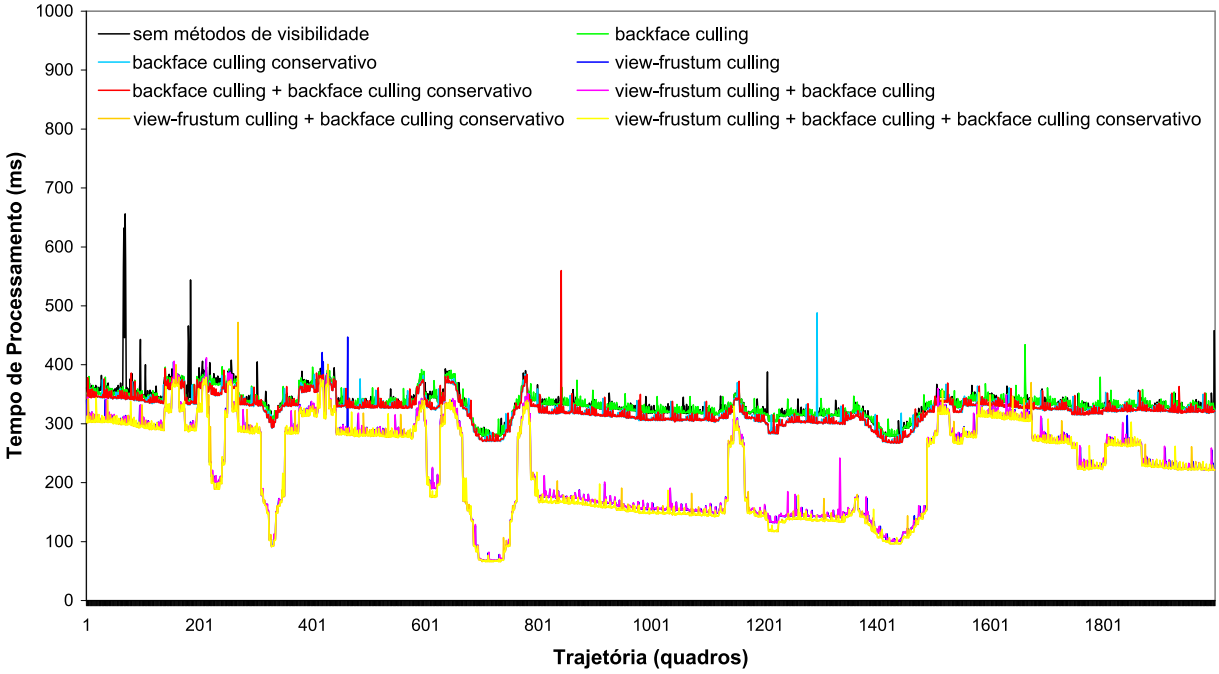


Figura 5.34: Tempo de processamento para a renderização de diferentes combinações de algoritmos de visibilidade, ao longo da trajetória no Mundo 6.

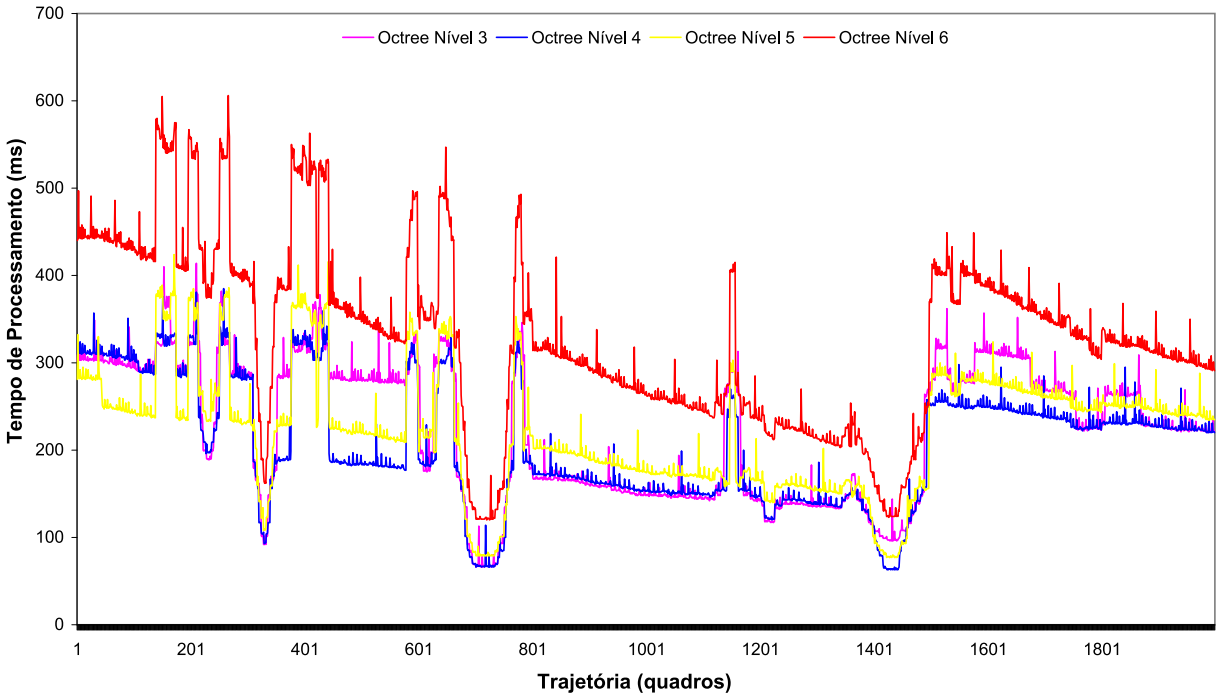


Figura 5.35: Tempo de processamento para a renderização de diferentes níveis da *Octree*, ao longo da trajetória no Mundo 6.

Pode ser observado subjetivamente nestas figuras que os formatos das curvas que representam o tempo de processamento gasto na renderização são semelhantes aos das curvas que representam o número de triângulos enviados ao *pipeline*. Por exemplo, a correlação entre as curvas geradas utilizando-se a *Octree* com nível de profundidade 4 é +0.95. Isso significa que o tempo

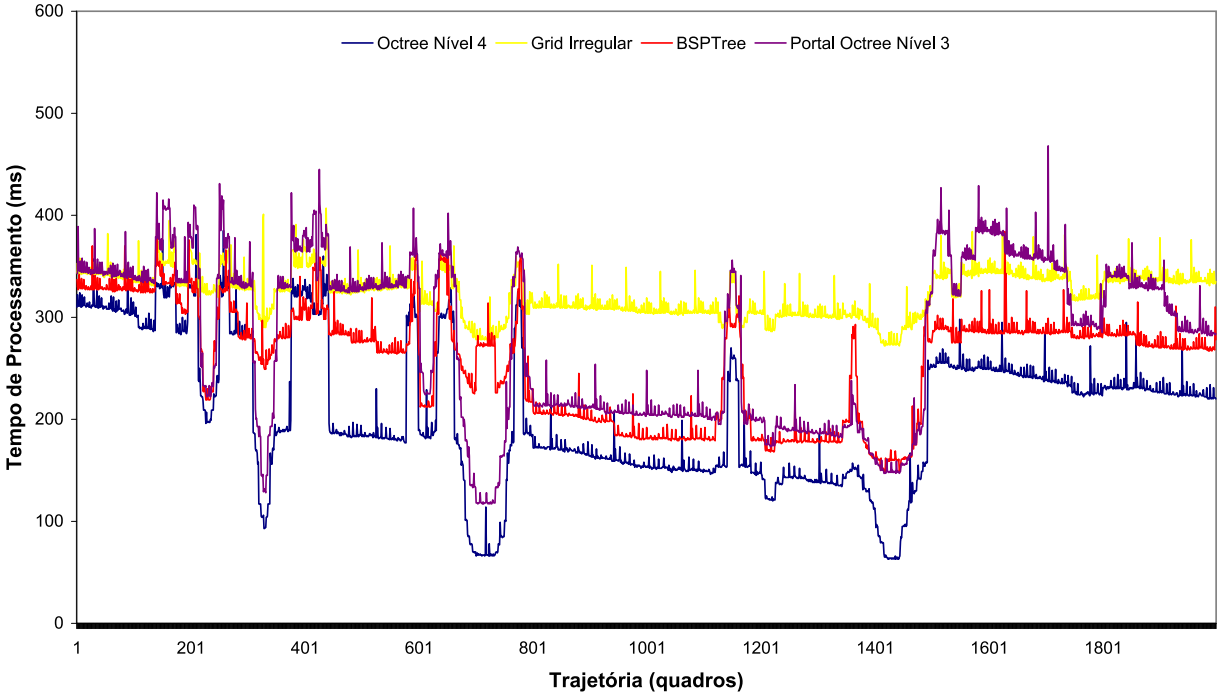


Figura 5.36: Tempo de processamento para a renderização de diferentes estruturas de particionamento espacial, ao longo da trajetória no Mundo 6.

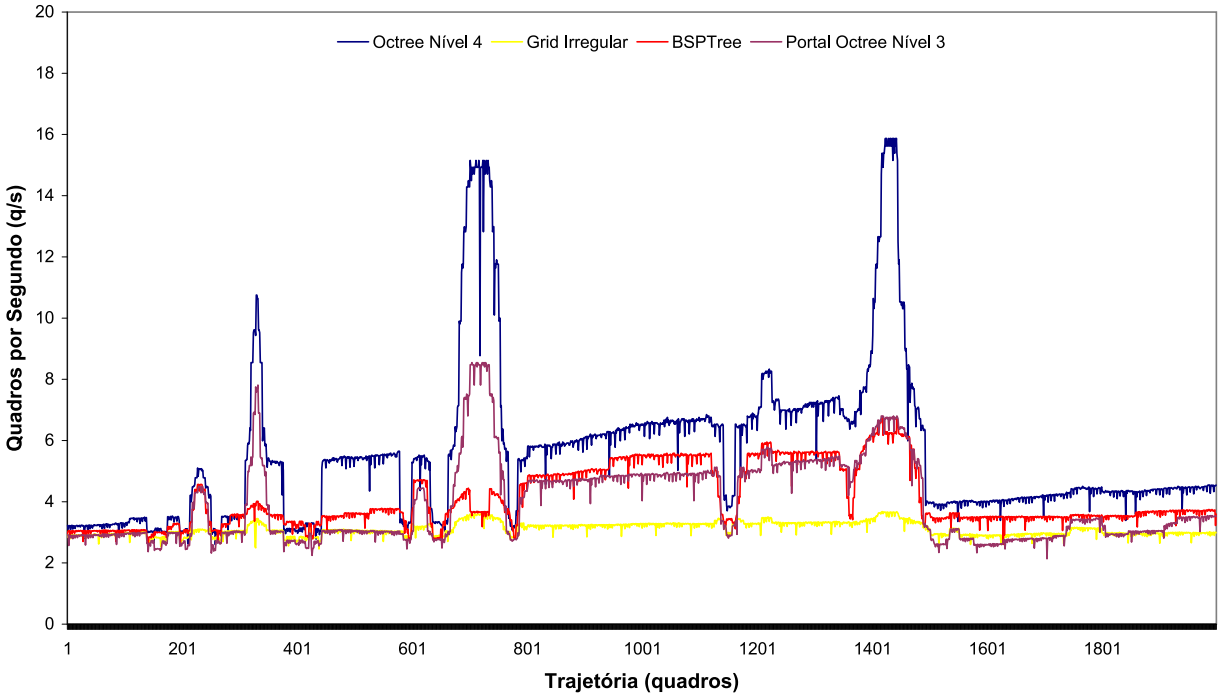


Figura 5.37: Taxa de quadros por segundo obtida ao longo da trajetória no Mundo 6.

de processamento aumenta proporcionalmente à medida que o número de triângulos enviados ao *pipeline* aumenta.

A melhor recomendação para o Mundo 6 foi a combinação dos algoritmos de visibi-

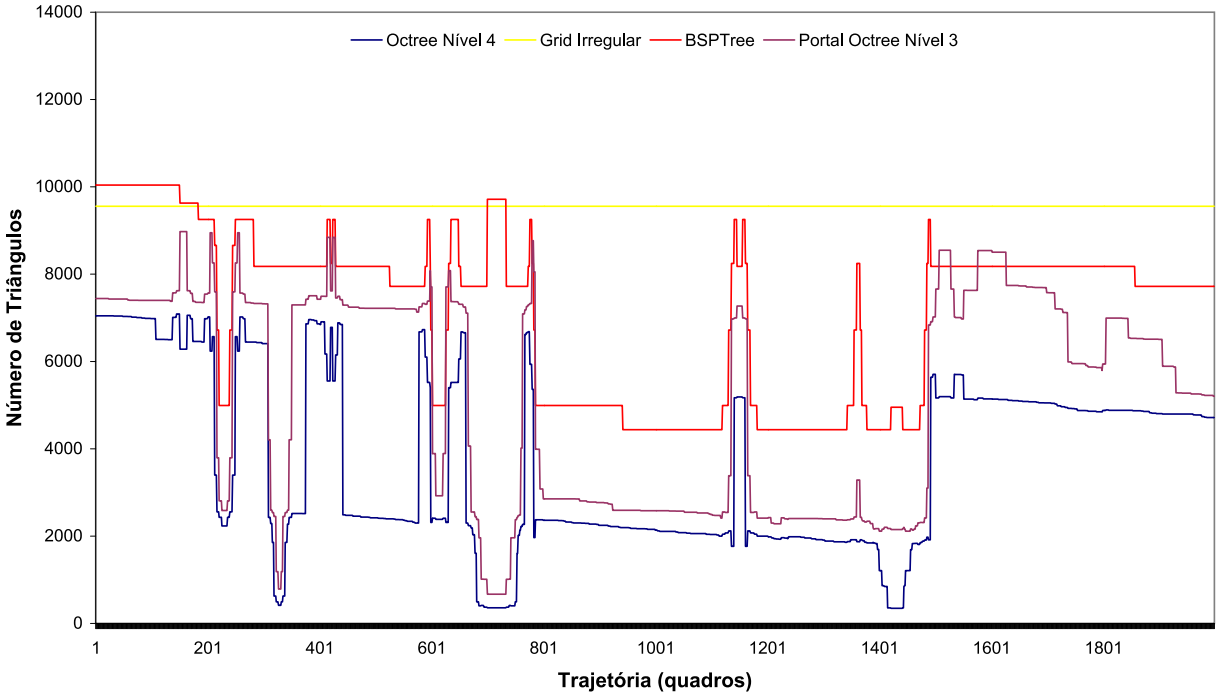


Figura 5.38: Número de triângulos enviados ao *pipeline* de renderização ao longo da trajetória no Mundo 6.

lidade *view-frustum culling* e *backface culling* conservativo. Quanto às estruturas, o nível de profundidade influenciou no desempenho da *Octree* e da Portal *Octree*. A Octree nível 4 obteve o melhor desempenho ($5,59$q/s, em média), seguida pela *BSP-Tree* ($4,11$q/s, em média). Finalmente, foi observado que o número de triângulos enviados ao *pipeline* de renderização teve mais influência no desempenho das estruturas de particionamento espacial do que o tempo de processamento para a execução do algoritmo de *view-frustum culling*.

5.4.3 Melhores Combinações de Algoritmos para os Ambientes Internos e Externos no iPaq

Podem ser observadas nas Figuras 5.39 e 5.40, as curvas que representam as melhores combinações de algoritmos de visibilidade para os ambientes internos, destacando-se os pontos de referência da trajetória de locomoção (Seção 5.3, Figura 5.7). Mais especificamente, a Figura 5.39 exibe o tempo de processamento necessário para a renderização de cada quadro e, a Figura **??**, o número de triângulos enviados ao *pipeline* de renderização.

Nas Figuras 5.41 e 5.42 podem ser visualizadas as curvas que representam as melhores combinações de algoritmos de visibilidade obtidas para os ambientes externos, nas quais destacam-se os pontos de referência da trajetória de locomoção (Seção 5.3, Figura 5.8). Mais precisamente, a Figura 5.41 representa o tempo de processamento necessário para a renderiza-

ção de cada quadro e, a Figura 5.42, o número de triângulos enviados ao *pipeline* de renderização.

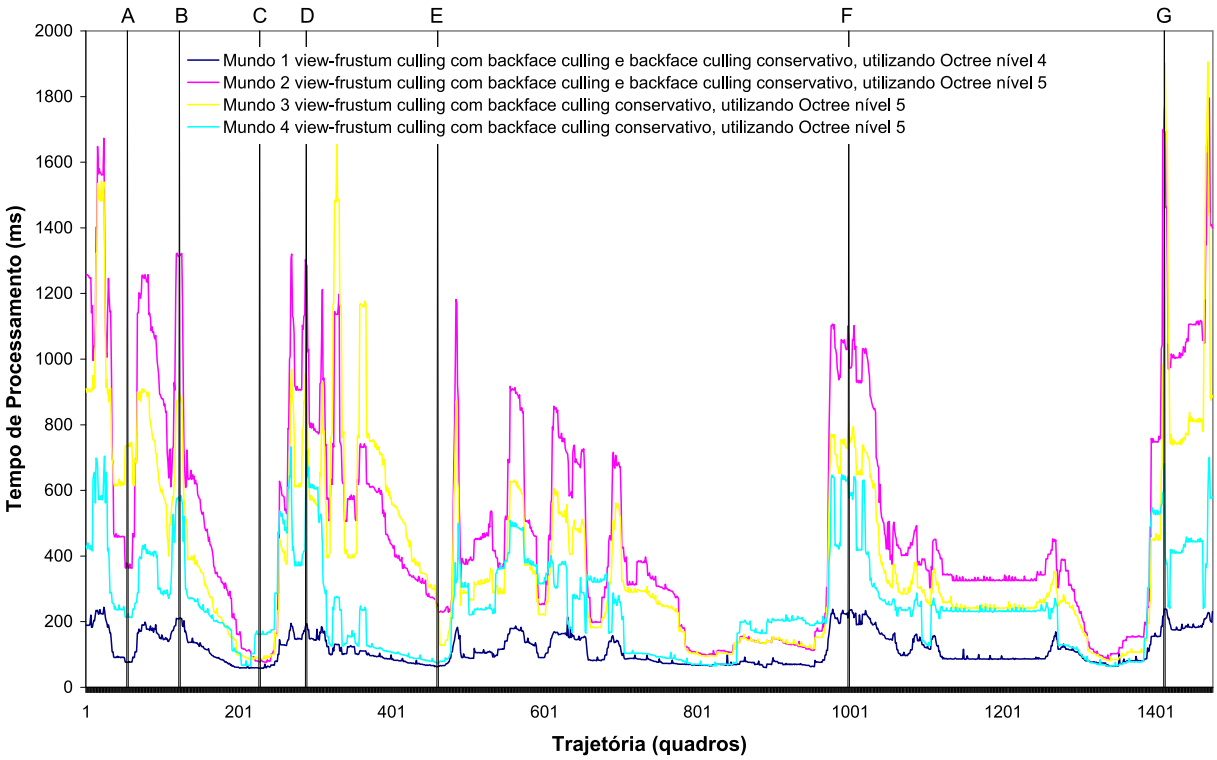


Figura 5.39: Tempo de processamento para renderizar as melhores combinações de algoritmos de visibilidade e estruturas de particionamento espacial para os ambientes internos (Mundo 1, Mundo 2, Mundo 3 e Mundo 4).

5.4.4 Ambientes Internos Executados no N82

Foram realizados testes no telefone celular N82 usando somente a melhor combinação dos algoritmos de visibilidade obtida no iPaq, para cada um dos seis mundos (Seção 5.4.3). Para a obtenção de resultados mais precisos, cada teste foi executado 5 vezes e a média obtida em cada caso de estudo foi utilizada para a geração dos gráficos.

Assim como no iPaq, o ambiente 3D que obteve o melhor desempenho foi o Mundo 1 (curva azul escuro na Figura 5.43), que atingiu taxas de $64,93$ e $11,03$q/s, no melhor e no pior caso, respectivamente (sendo $30,61$q/s, em média). O número de triângulos enviado ao *pipeline* de renderização obteve uma correlação de $+0,97$, em relação ao tempo de processamento necessário para a renderização. O Mundo 2 (curva magenta na Figura 5.43) obteve taxas de $1,18$q/s e $32,25$q/s no pior e no melhor caso, respectivamente (sendo $7,77$q/s, em média). O número de triângulos enviados ao *pipeline* de renderização obteve uma correlação de $+0,99$, em relação ao tempo de processamento necessário para a renderização. No Mundo 3 (curva

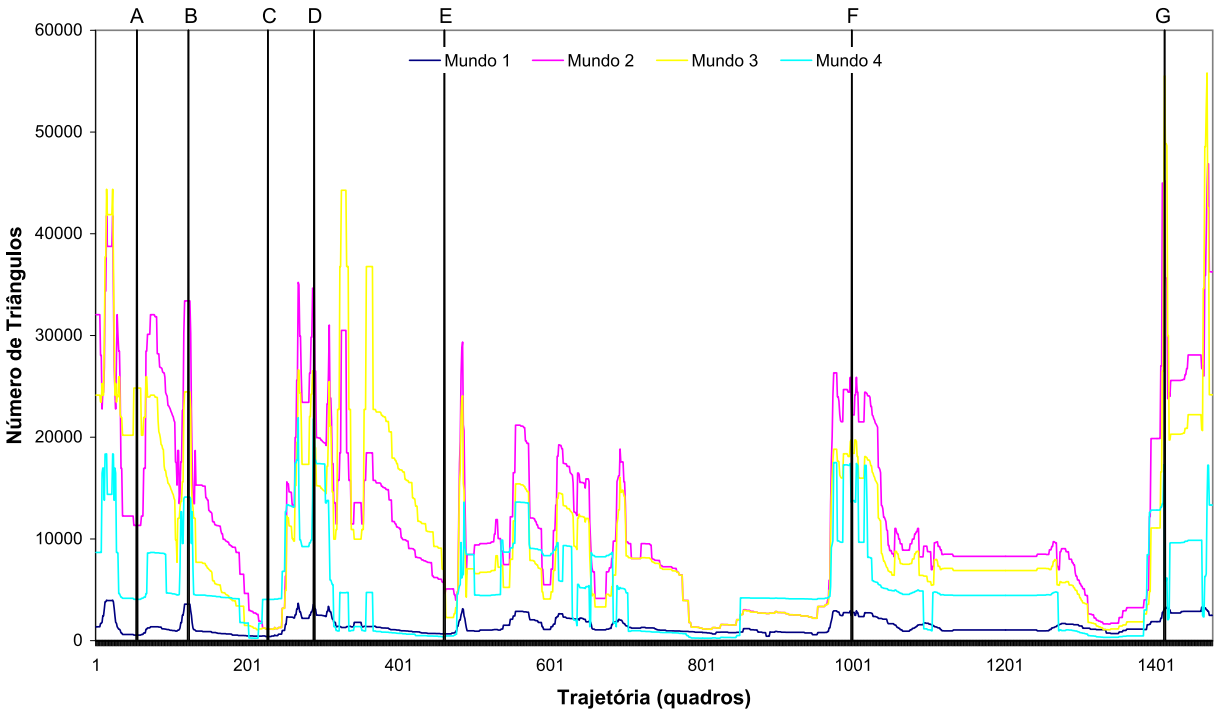


Figura 5.40: Número de triângulos enviados ao *pipeline* ao longo da trajetória, para cada uma das melhores combinações de algoritmos de visibilidade e particionamento espacial nos ambientes internos (Mundo 1, Mundo 2, Mundo 3 e Mundo 4).

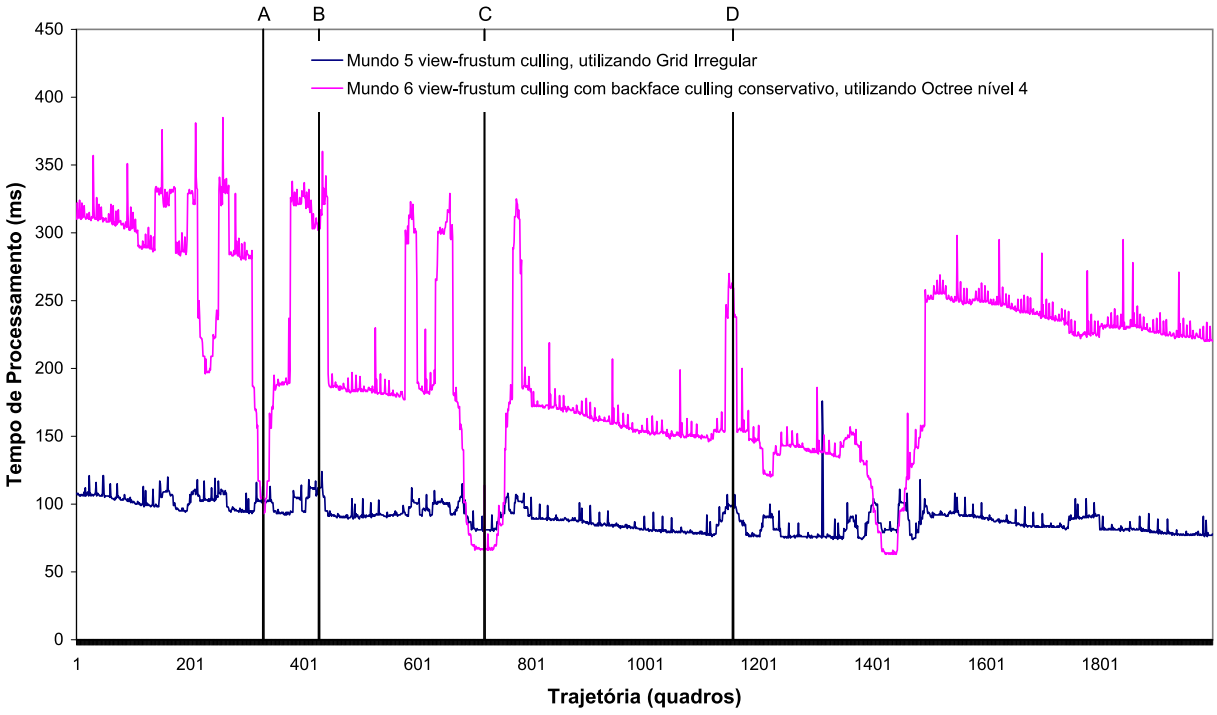


Figura 5.41: Tempo de processamento para renderizar as melhores combinações de algoritmos de visibilidade e estruturas de particionamento espacial para os ambientes externos (Mundo 5 e Mundo 6).

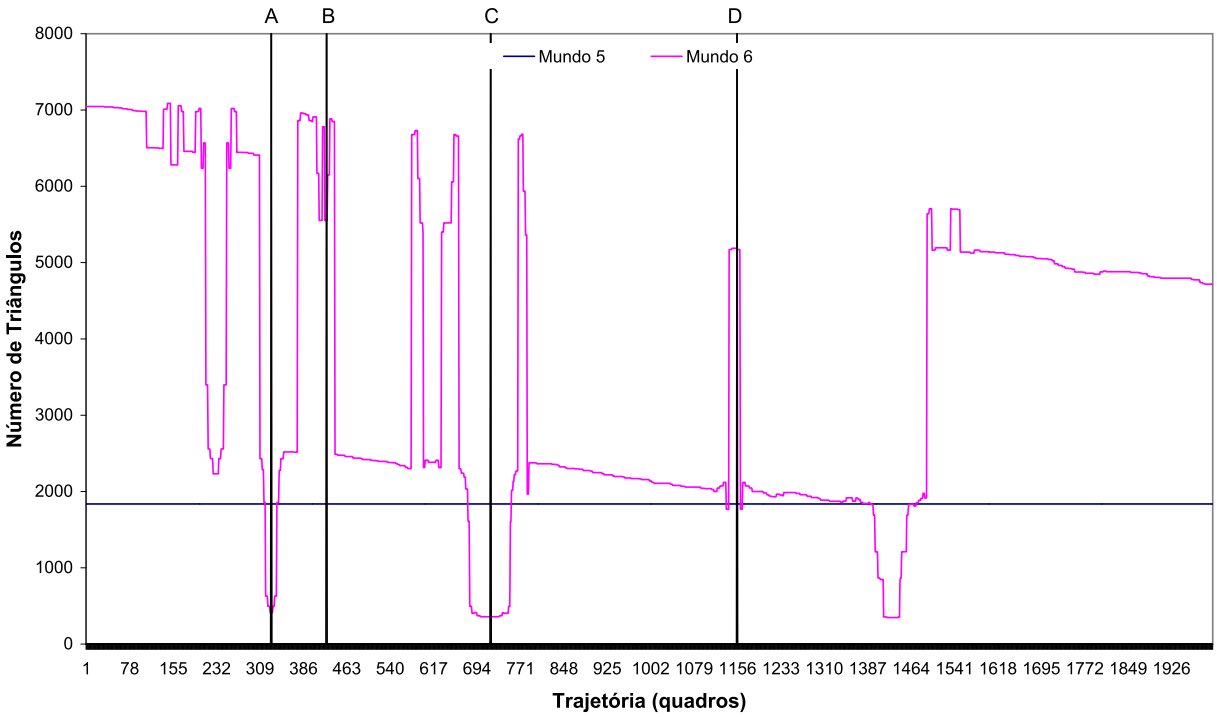


Figura 5.42: Número de triângulos enviados ao *pipeline* ao longo da trajetória, para cada uma das melhores combinações de algoritmos de visibilidade e particionamento espacial nos ambientes externos (Mundo 5 e Mundo 6).

amarela na Figura 5.43), foram obtidas taxas de $0,99$q/s e $32,25$q/s no pior e no melhor caso, respectivamente (sendo $8,66$q/s, em média). No Mundo 3, o número de triângulos enviados ao *pipeline* de renderização obteve uma correlação de $+0,99$, em relação ao tempo de processamento necessário para a renderização. No Mundo 4 (curva azul claro na Figura 5.43), a melhor combinação alcançou taxas de $2,83$ e $64,93$q/s no pior e no melhor caso, respectivamente (sendo $16,65$q/s, em média). No Mundo 4, o número de triângulos enviados ao *pipeline* de renderização obteve uma correlação de $+0,99$, em relação ao tempo de processamento necessário para a renderização.

5.4.5 Ambientes Externos Executados no N82

No celular Nokia N82, o Mundo 5 (curva azul escuro na Figura 5.44) obteve taxas de $20,00$ e $65,78$q/s, no pior e no melhor caso, respectivamente (sendo $36,61$q/s, em média). No Mundo 6, foram produzidas taxas de $7,80$ e $65,78$q/s, no pior e no melhor caso, respectivamente (sendo $18,74$q/s, em média). No Mundo 6 (curva magenta na Figura 5.44), o número de triângulos enviados ao *pipeline* de renderização obteve uma correlação de $+0,99$, em relação ao tempo de processamento necessário para a renderização.

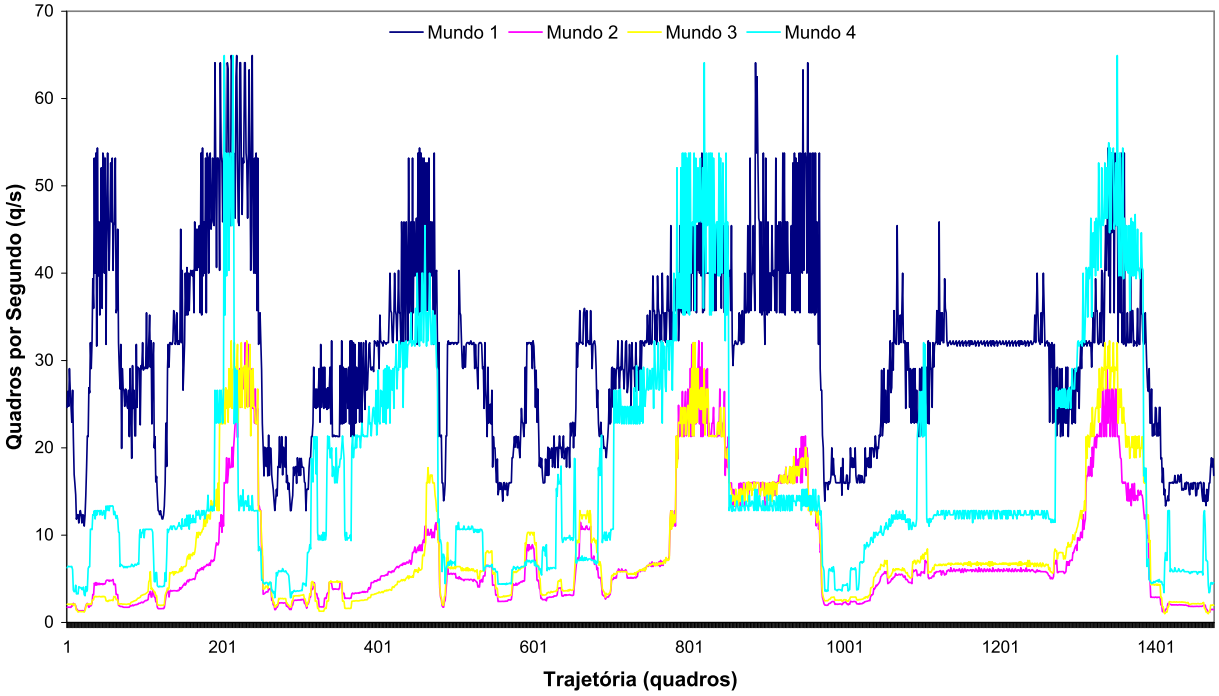


Figura 5.43: Taxa de quadros por segundo obtidas ao longo da trajetória nos mundos internos.

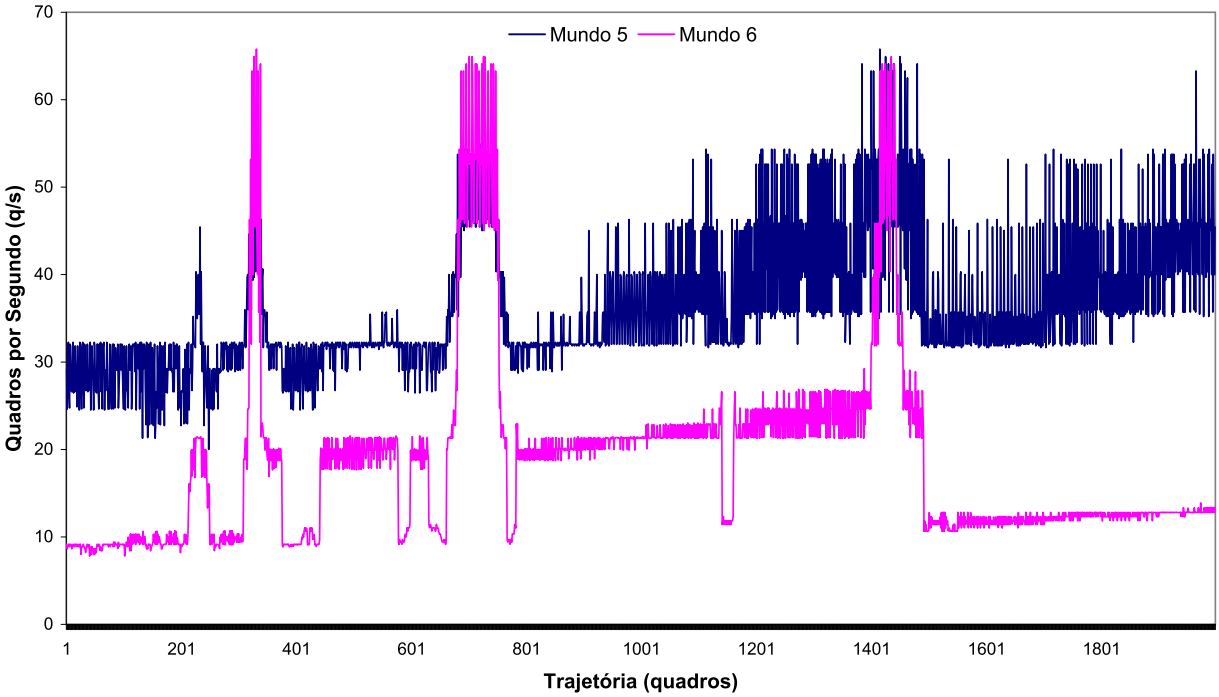


Figura 5.44: Taxa de quadros por segundo obtidas ao longo da trajetória nos mundos externos.

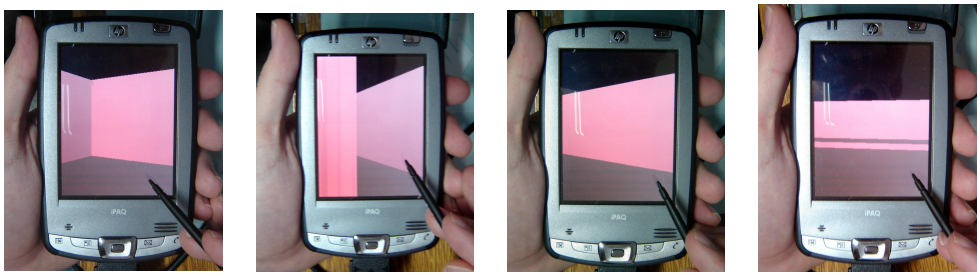
5.5 Discussão Final

Nos testes realizados com o iPaq para o Mundo 1, a *Octree* nível 4 foi a estrutura de dados que obteve o melhor desempenho, atingindo taxas de $4,09$q/s e $17,24$q/s no pior e no melhor caso, respectivamente (sendo $10,06$q/s, em média). O número de triângulos enviado

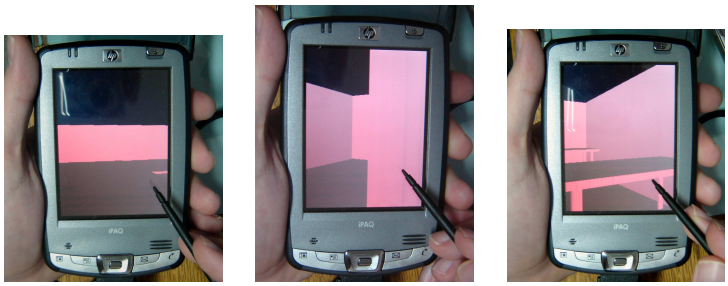
ao *pipeline* de renderização obteve uma correlação de $+0,87$, em relação ao tempo de processamento necessário para a renderização. Já no Mundo 2, a *Octree* nível 5 foi a estrutura que obteve o melhor desempenho, apresentando taxas de $0,55$q/s e $13,15$q/s no pior e no melhor caso, respectivamente (sendo $3,53$q/s, em média). No Mundo 2, o número de triângulos enviados ao *pipeline* de renderização obteve uma correlação de $+0,98$, em relação ao tempo de processamento necessário para a renderização. Assim como no Mundo 2, no Mundo 3, a *Octree* nível 5 foi a estrutura que obteve o melhor desempenho com taxas de $0,52$q/s e $12,5$q/s no pior e no melhor caso, respectivamente (sendo $4,13$q/s, em média). No Mundo 3, o número de triângulos enviados ao *pipeline* de renderização obteve uma correlação de $+0,97$, em relação ao tempo de processamento necessário para a renderização. No Mundo 4, assim como no Mundo 2 e 3, a *Octree* nível 5 foi a estrutura que obteve o melhor desempenho alcançando taxas de $1,36$ e $15,87$q/s no pior e no melhor caso, respectivamente (sendo $5,91$q/s, em média). No Mundo 4, o número de triângulos enviados ao *pipeline* de renderização obteve uma correlação de $+0,95$, em relação ao tempo de processamento necessário para a renderização. Diferentemente dos outros ambientes, no Mundo 5, o *Grid* Irregular foi a estrutura que obteve o melhor desempenho, com taxas de $5,61$ e $15,87$q/s no pior e no melhor caso, respectivamente (sendo $9,55$q/s, em média). No Mundo 6, a *Octree* nível 5 foi a estrutura que obteve o melhor desempenho, produzindo taxas de $2,59$ e $15,87$q/s no pior e no melhor caso, respectivamente (sendo $5,59$q/s, em média). No Mundo 6, o número de triângulos enviados ao *pipeline* de renderização obteve uma correlação de $+0,95$, em relação ao tempo de processamento necessário para a renderização.

Já para os testes realizados com o N82, o Mundo 5 obteve o melhor resultado ($36,6$q/s, em média) seguido pelo Mundo 1 (o Mundo 1 obteve o melhor resultado entre os mundos internos, de $30,61$q/s, em média). O Mundo 2 obteve desempenho próximo ao do Mundo 3 ($7,77$ e $8,66$q/s, em média, respectivamente). Tanto nos mundos internos quanto nos externos, a quantidade de triângulos enviadas ao *pipeline* de renderização está correlacionada ao tempo de processamento necessário para renderizar cada quadro.

Pode ser observado que ao longo da trajetória, as curvas que representam o tempo de processamento gasto e as curvas que representam o número de triângulos enviados ao *pipeline* de renderização apresentam pontos de inflexão (A, B, C, D, E, F e G para os Mundos internos e A, B, C e D para os Mundos externos, exibidos nas Tabelas 5.5 e 5.6, respectivamente, e nas fotos da Figura **??**).



(a) Ponto A (b) Ponto B (c) Ponto C (d) Ponto D

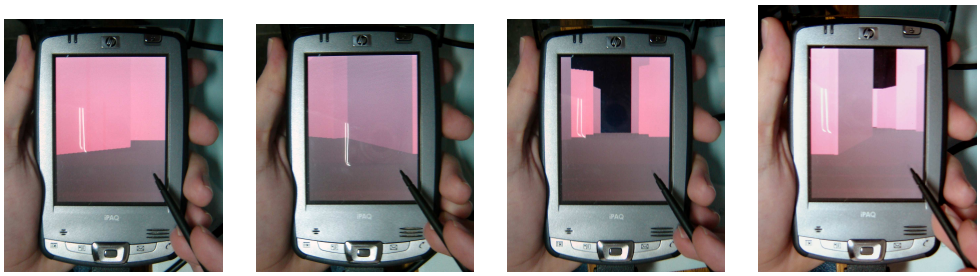


(e) Ponto E (f) Ponto F (g) Ponto G

Figura 5.45: Observa-se a visão da câmera nos mundos internos para os pontos da trajetória de locomoção pelo ambiente.

(a) Ponto A Mundo 3 (b) Ponto G Mundo 2

Figura 5.46: Observam-se os pontos da trajetória nos quais as visões da câmera no Mundo 2 e 3 diferem da visão no Mundo 1.



(a) Ponto A (b) Ponto B (c) Ponto C (d) Ponto D

Figura 5.47: Observam-se as visões da câmera nos pontos da trajetória para os mundos externos.

Nos ambientes internos, pode-se observar subjetivamente que o número de objetos contidos no volume de visualização está relacionado com o tempo gasto para renderizar cada cena. No caso do ponto de referência **A**, o observador visualiza apenas a parede da sala e poucos triângulos estão contidos no volume de visualização enquanto que em **B**, apenas a parede da sala é visualizada, porém, muitos triângulos estão contidos no *frustum*. Nos pontos de referência **C** e **E**, assim como em **A**, poucos triângulos estão contidos no *frustum*, já que grande parte do volume de visualização está localizado fora do ambiente. Já nos outros pontos, **D**, **F** e **G**, apenas geometrias pouco complexas são visualizadas, porém, muitos triângulos estão contidos no volume de visualização.

Nos ambientes externos, assim como nos ambientes internos, subjetivamente pode ser observado que o número de objetos dentro do *frustum* está relacionado ao tempo gasto para renderizar cada cena. No ponto de referência **A**, o *frustum* contém um número pequeno de triângulos. Já nos pontos **B** e **C**, grande parte do *frustum* está localizado fora do ambiente, assim, contendo poucos triângulos. Já o ponto **D**, apresenta uma quantidade de triângulos superior aos pontos **A**, **B** e **C**.

Baseando-se nos resultados obtidos nos testes e na análise comparativa de desempenho realizada, no próximo capítulo serão tecidas as principais conclusões resultantes deste trabalho.

Capítulo 6
Conclusões

6.1 Sumário da Pesquisa

Este trabalho apresentou o VisMobile, um sistema para renderização interativa em dispositivos móveis (testado com sucesso no PocketPC iPaq modelo hc2490b e no telefone celular Nokia N82), utilizando a API OpenGL ES, o qual corresponde a uma biblioteca para desenvolvedores de aplicações gráficas 3D. Diferentes combinações de algoritmos de visibilidade e estruturas de dados espaciais foram implementadas e testadas. Variando-se os ambientes gráficos, os dispositivos e as plataformas de execução, essas diferentes combinações de algoritmos de visibilidade e estruturas de dados foram testadas, obtendo desempenhos distintos.

Este trabalho mostra que, dependendo do ambiente 3D modelado, a renderização dos ambientes, a taxas interativas, pode se tornar uma tarefa bastante complexa, especialmente para ambientes contendo um número grande de polígonos. Outro ponto relevante diz respeito à escolha dos melhores parâmetros (nível de profundidade das estruturas de dados espaciais) para as combinações de algoritmos, os quais produzem resultados variados, dependendo da complexidade do ambiente. Por exemplo, as profundidades da *Octree* e da Portal *Octree* (ou mesmo a resolução do *Grid* Irregular) não puderam ser escolhidas automaticamente, necessitando a realização de um número considerável de testes.

Para o Mundo 1, a melhor combinação dos algoritmos de visibilidade foi a que utilizou o *view-frustum culling* com o *backface culling* e o *backface culling* conservativo. Quanto às estruturas, o nível de profundidade influenciou no desempenho obtido pela *Octree* e Portal *Octree*, sendo a *Octree* de nível 4 a estrutura que atingiu o melhor desempenho. Dentre as es-

truturas de particionamento espacial testadas, a *Octree* obteve destaque, seguida pela *BSP-Tree*. Foi também demonstrado que tanto o tempo de processamento necessário para a execução do *view-frustum culling*, quanto o número de triângulos enviados ao *pipeline* de renderização ao longo da trajetória, influenciaram no desempenho das estruturas de particionamento espacial atingindo uma taxa média aproximada de $30,61$q/s. Existe uma relação entre o número de triângulos do ambiente e o tempo de processamento gasto nos algoritmos de visibilidade e, a influência de ambos depende da complexidade do ambiente.

A melhor combinação dos algoritmos de visibilidade para o Mundo 2 também foi a que utilizou o *view-frustum culling*, com o *backface culling* e o *backface culling* conservativo. Quanto às estruturas, o nível de profundidade também influenciou no desempenho da *Octree* e da Portal *Octree*. A *Octree* nível 5 obteve o melhor desempenho ($7,77$q/s, em média), seguida pela Portal *Octree*. Foi também constatado que o número de triângulos enviados ao *pipeline* de renderização teve mais influência no desempenho das estruturas de particionamento espacial do que o tempo de processamento necessário para a execução do *view-frustum culling*, devido basicamente à grande quantidade de triângulos presente no ambiente (102.232 triângulos).

Já no Mundo 3, diferentemente dos Mundos 1 e 2, a melhor combinação dos algoritmos de visibilidade para o Mundo 3 foi a que utilizou o algoritmo de *view-frustum culling* com o *backface culling* conservativo. Quanto às estruturas, o nível de profundidade também influenciou no desempenho da *Octree* e da Portal *Octree*. A *Octree* com 5 níveis de profundidade obteve o melhor desempenho ($8,66$q/s, em média), seguida pela Portal *Octree*. Foi também constatado que o número de triângulos enviados ao *pipeline* de renderização teve mais influência no desempenho das estruturas de particionamento espacial do que o tempo de processamento necessário para a execução do algoritmo de *view-frustum culling*.

No Mundo 4, assim como no Mundo 3, a melhor recomendação dos algoritmos de visibilidade para o Mundo 4 foi a que combinou os algoritmos de *view-frustum culling* e *backface culling* conservativo. Quanto às estruturas, o nível de profundidade também influenciou no desempenho da *Octree* e da Portal *Octree*. A *Octree* com 5 níveis de profundidade obteve o melhor desempenho (em média $16,65$q/s), seguida pela *BSP-Tree*. O número de triângulos enviados ao *pipeline* de renderização teve mais influência no desempenho das estruturas de particionamento espacial, do que o tempo de processamento para a execução do algoritmo de *view-frustum culling*.

A melhor recomendação de algoritmo de visibilidade para o Mundo 5 foi o *view-frustum culling*. Quanto às estruturas, os níveis de profundidade influenciaram no desempenho da *Octree* e da Portal *Octree*, porém, não produziram resultados competitivos o bastante para

ultrapassar o desempenho do *Grid* Irregular. Nesse caso, o *Grid* Irregular obteve o melhor desempenho (em média, $36,61$q/s), seguido da *Octree* nível com 3 níveis de profundidade. Assim como no Mundo 1, tanto o número de triângulos enviados ao *pipeline* de renderização, quanto o tempo de processamento para a execução do *view-frustum culling*, influenciaram na diferença de desempenho das estruturas de particionamento espacial.

No Mundo 6, a melhor recomendação foi a combinação dos algoritmos de visibilidade *view-frustum culling* e *backface culling* conservativo. Quanto às estruturas, o nível de profundidade influenciou no desempenho da *Octree* e da Portal *Octree*. A Octree nível 4 obteve o melhor desempenho ($18,74$q/s, em média), seguida pela *BSP-Tree*. Também, foi observado que o número de triângulos enviados ao *pipeline* de renderização teve mais influência no desempenho das estruturas de particionamento espacial do que o tempo de processamento para a execução do algoritmo de *view-frustum culling*.

6.2 Trabalhos Futuros

O código atual do VisMobile pode ser otimizado de modo a diminuir o uso da memória, assim como do processador. Como trabalhos futuros, pode-se realizar o armazenamento e acesso otimizado da estrutura de particionamento espacial na memória. Ou seja, realizando o armazenando seqüencial dos triângulos para diminuir o número de acessos a memória do dispositivo, bem como, o número de chamadas ao método **glDrawArrays**. Alguns dispositivos móveis não apresentam suporte a ponto flutuante e, com isso, durante a execução da aplicação, suas variáveis são convertidas para ponto fixo, fazendo uso do processador. Com isso, faz-se necessário a conversão do VisMobile para a utilização apenas de ponto fixo. Além disso, pode-se diminuir o tamanho dos arquivos dos mundos particionados salvando suas informações de forma binária.

Como extensão ao VisMobile, é interessante a implementação de obstáculos virtuais para uso no método de *occlusion culling* implementado. Adicionalmente, otimizações pode ser realizadas no teste de oclusão. Posteriormente, outros métodos de *occlusion culling* podem ser implementados para comparar seu desempenho em diferentes dispositivos.

Quanto às estruturas de particionamento, os desempenhos das estruturas *BSP-Tree* e Portal *Octree* podem ser melhorados, realizando a implementação de um PVS e de um algoritmo de células e portais, respectivamente, para estas estruturas. Além das estruturas de dados espaciais presentes no VisMobile, é recomendado também a implementação de outras estruturas como, por exemplo, a *Kd-Tree*.

Nos testes realizados em diferentes ambientes, ainda podem ser explorados novos cenários como, por exemplo, os ambientes híbridos (contendo áreas internas e externas), bem como, em ambientes dinâmicos. Outra variação seria a realização de testes utilizando diferentes valores do FOV da câmera.

Finalmente, para a geração de um ambiente ainda mais realista, pode ser adicionado ao VisMobile suporte à textura, utilização de *mipmaps* e níveis de detalhamento (LOD). A inserção de suporte à detecção de colisão também poderá aumentar o realismo quando a câmera se locomover pelo ambiente, guiada pelo usuário da aplicação.

Referências Bibliográficas

[1] Spinor GmbH. Shark3d. Disponível em http://www.spinor.com/?select=shark3d/render/index. Último acesso em 8/5/2008.

[2] James T. Klosowski and Cláudio T. Silva. The Prioritized-Layered Projection Algorithm for Visible Set Estimation. In *IEEE Transactions on Visualization and Computer Graphics*, pages 108–123, 2000.

[3] Daniel Cohen-Or, Yiorgos Chrysanthou, Cláudio T. Silva, and Frédo Durand. A Survey of Visibility for Walkthrough Applications. In IEEE, editor, *IEEE Transactions on Visualization and Computer Graphics*, volume 9, pages 412–431, July 2003.

[4] David Luebke and Chris Georges. Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets. In *Proceedings of Symposium on Interactive 3D Graphics*. ACM Press, 1995.

[5] Seth J. Teller and Carlos H. Sequin. Visibility Preprocessing for Interactive Walkthroughs. In *SIGGRAPH'91: Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*, pages 61–69, July 1991.

[6] Vladlen Koltun, Yiorgos Chrysanthou, and Daniel Cohen-Or. Virtual Occluders: An Efficient Intermediate PVS Representation. In *Proceedings of the 11th Eurographics Workshop on Rendering*, pages 59–70. Eurographics, 2000.

[7] Jane Hwang, Jaehoon Jung, and Gerard Jounghyun Kim. Hand-held Virtual Reality: a Feasibility Study. In *VRST'06: Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, pages 356–363, New York, NY, USA, 2006. ACM Press.

[8] Antti Nurminen. m-LOMA - a Mobile 3D City Map. In *Web3D'06: Proceedings of the 11th International Conference on 3D Web Technology*, pages 7–18, New York, NY, USA, 2006. ACM Press.

[9] Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering*. A. K. Peters, 2nd edition, 2002.

[10] Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker. A Characterization of Ten Hidden-Surface Algorithms. *ACM Computing Surveys (CSUR)*, 6(1):1–55, 1974.

[11] Jiří Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful. *Computer Graphics Forum (Proceedings of Eurographics'04)*, 23(3):615–624, September 2004.

[12] Daniel Cohen-Or, Eran Rich, Uri Lerner, and Victor Shenkar. A Real-time Photo-realistic Visual Flythrough. *IEEE Transactions on Visualization and Computer Graphics*, 2:255–264, 1996.

[13] Satyan R. Coorg and Seth J. Teller. Temporally Coherent Conservative Visibility (Extended Abstract). In *Symposium on Computational Geometry*, pages 78–87, 1996.

[14] Tom Hudson, Dinesh Manocha, Jonathan Cohen, Ming C. Lin, Kenneth E. Hoff III, and Hansong Zhang. Accelerated Occlusion Culling Using Shadow Frusta. *Proceedings of ACM Symposium on Computational Geometry*, pages 1–10, 1997.

[15] Jiří Bittner, Vlastimil Havran, and Pavel Slavík. Hierarchical Visibility Culling with Occlusion Trees. *Proceedings of Computer Graphics International'98 (CGI'98)*, pages 207–219, 1998.

[16] Ned Greene and Michael Kass. Error-bounded Antialiased Rendering of Complex Environments. *SIGGRAPH'94: Proceedings of the 21th Annual Conference on Computer Graphics and Interactive Techniques*.

[17] Dirk Bartz, Michael Meißner, and Tobias Hüttner. Extending Graphics Hardware for Occlusion Queries in OpenGL. In *HWWS'98: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 97–104., New York, NY, USA, 1998. ACM.

[18] James T. Klosowski and Cláudio T. Silva. Efficient Conservative Visibility Culling Using the Prioritized-Layered Projection Algorithm. In *IEEE Transactions on Visualization and Computer Graphics*, volume 7, pages 365–379, 2001.

[19] Gernot Schaufler, Julie Dorsey, Xavier Décoret, and François Sillion. Conservative Volumetric Visibility with Occluder Fusion. In *SIGGRAPH'00: Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pages 229–238, July 2000.

[20] Frédo Durand, George Drettakis, Joelle Thollot, and Claude Puech. Conservative Visibility Preprocessing using Extended Projections. In *SIGGRAPH'00: Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pages 239–248, July 2000.

[21] Peter Wonka, M. Wimmer, and D. Schmalstieg. Visibility Preprocessing with Occluder Fusion for Urban Walkthroughs. In *Proceedings of the 11th Eurographics Workshop on Rendering*, pages 71–82. Eurographics, 2000.

[22] Vladlen Koltun, Yiorgos Chrysanthou, and Daniel Cohen-Or. Hardware-accelerated From-region Visibility Using a Dual Ray Space. *Eurographics Workshop on Rendering*, pages 205–216, 2001.

[23] Daniel Cohen-Or, Yiorgos Chrysanthou, Cláudio T. Silva, and Frédo Durand. A Survey of Visibility for Walkthrough Applications. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):412–431, 2002.

[24] Fábio O. Moreira. *Smart Visible Sets for Networked Virtual Environments*. Tese de Doutorado, Instituto de Informática - Universidade Federal do Rio Grande do Sul, 2003.

[25] John M. Airey, John H. Rohlf, and Frederick P. Brooks Jr. Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments. In *Proceedings of Symposium on Interactive 3D Graphics*, pages 41–49, 1990.

[26] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical Z-buffer Visibility. In *SIGGRAPH'93: Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, pages 231–238, New York, NY, USA, 1993. ACM.

[27] Hansong Zhang, Dinesh Manocha, Thomas Hudson, and Kenneth E. Hoff III. Visibility Culling Using Hierarchical Occlusion Maps. *SIGGRAPH'97: Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, 31:77–88, 1997.

[28] Peter Wonka and Dieter Schmalstieg. Occluder Shadows for Fast Walkthroughs of Urban Environments. In P. Brunet and R. Scopigno, editors, *Computer Graphics Forum (Proceedings of Eurographics'99)*, volume 18(3), pages 51–60. The Eurographics Association, 1999.

[29] Fausto Bernardini, James T. Klosowski, and Jihad El-Sana. Directional Discretized Occluders for Accelerated Occlusion Culling. *Computer Graphics Forum (Proceedings of Eurographics'00)*, 19(3), 2000.

[30] Shaun Nirenstein. *Fast and Accurate Visible Preprocessing*. PhD Thesis, Faculty of Science at the University of Cape Town, South Africa, 2003.

[31] Tommer Leyvand, Olga Sorkine, and Daniel Cohen-Or. Ray Space Factorization for From-Region Visibility. *ACM Transactions on Graphics*, 22(3):595–604, 2003.

[32] Frédéric Mora, Lilian Aveneau, and Michel Mériaux. Coherent and Exact Polygon-to-Polygon Visibility. In *Proceedings of Winter School on Computer Graphics'05*, 2005.

[33] Denis Haumont, Olivier Debeir, and FranÃ§ois Sillion. Volumetric Cell-and-Portal Generation. *Computer Graphics Forum (Proceedings of Eurographics'03)*, 22(3):303–312, September 2003.

[34] Samuli Laine. A General Algorithm for Output-Sensitive Visibility Preprocessing. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 31–39. ACM Press, 2005.

[35] Mikko Laakso. *Potentially Visible Set*. Technical Report, Helsinki University of Technology - Telecommunications Software and Multimedia Laboratory, 2003.

[36] Cláudio T. Silva, Yi-Jen Chiang, Jihad El-Sana, and Peter Lindstrom. Out-of-core Algorithms for Scientific Visualization and Computer Graphics. In *IEEE Visualization'02*, 2002. Course Notes for Tutorial 4.

[37] Shaun Nirenstein, Edwin Blake, and James Gain. Exact From Region Visibility Culling. In *Proceedings of the 13th Eurographics Workshop on Rendering*, pages 191–202, Pisa, Italy, 2002.

[38] Zhi Zheng and Tony K. Y. Chan. Optimized Neighbour Prefetch and Cache for Client-server Based Walkthrough. In *CW'03: Proceedings of the 2003 International Conference on Cyberworlds*, page 143, Washington, DC, USA, 2003. IEEE Computer Society.

[39] Robert D. Schiffenbauer. *A Survey of Aspect Graphs*. Technical Report, Polytechnic University - Department of Computer and Information Science, 2001.

[40] Seth J. Teller. *Visibility Computations in Densely Occluded Polyhedral Environments*. PhD Thesis, University of California at Berkeley, 1992.

[41] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[42] C. Tzafestas and P. Coiffet. Real-Time Collision Detection Using Spherical Octrees : VR Application. In *Proceedings of IEEE International Workshop on Robot and Human Communication*, pages 500–506, Tsukuba, Japan, 1996.

[43] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On Visible Surface Generation by a priori Tree Structures. In *SIGGRAPH'80: Proceedings of the 7^{th} Annual Conference on Computer Graphics and Interactive Techniques*, pages 124–133, New York, NY, USA, 1980. ACM.

[44] Bruce Naylor, John Amanatides, and William Thibault. Merging BSP Trees yields Polyhedral Set Operations. In *SIGGRAPH'90: Proceedings of the 17^{th} Annual Conference on Computer Graphics and Interactive Techniques*, volume 24, pages 115–124, New York, NY, USA, 1990. ACM.

[45] The Khronos Group. OpenGL ES - Embedded System. Disponível em http://www.khronos.org/opengles/. Último acesso em 01/6/2008.

[46] M. Segal and K. Akeley. *The Design of the OpenGL Graphics Interface*. Technical Report, Silicon Graphics. Disponível em http://www.sun.com/software/graphics/opengl/OpenGLdesign.pdf. Último acesso em 09/08/2006.

[47] Maria Andréia F. Rodrigues, Rafael Garcia Barbosa, and Wendel Bezerra Silva. Desenvolvimento de Aplicações 3D para dispositivos Móveis utilizando as APIs M3G e OpenGL ES. *Revista de Informática Teórica e Aplicada - RITA*, 13(2):69–102, 2006.

[48] Maria Andréia F. Rodrigues, Rafael Garcia Barbosa, and Wendel Bezerra Silva. Desenvolvimento de Aplicações 3D para dispositivos Móveis utilizando as APIs M3G e OpenGL ES. In *SIBGRAPI'06: Proceedings of the 19^{th} Brazilian Symposium on Computer Graphics and Image Processing*, 2006. Course Notes for Tutorial.

[49] K. Pulli. APIs for Mobile Graphics. *SPIE Electronic Imaging 2006 Multimedia on Mobile Devices II*, pages 1–13, 2006.

[50] M. Segal and K. Akeley. *The OpenGL Graphic System: A Specification (Version 1.3)*. Silicon Graphics. Disponível em http://www.opengl.org/documentation/specs/version1.3/glspec13.pdf. Último acesso em 09/08/2006.

[51] Rasteroid NVIDIA Hybrid Graphics. *NVIDIA Corporation to Acquire Hybrid Graphics*. Disponível em http://www.nvidia.com/object/IO_30518.html. Último acesso em 01/06/2008.

[52] Hans-Martin Will. Vincent 3D Rendering Library - Open Source Graphics Libraries for Mobile and Embedded Devices. Disponível em http://www.vincent3d.com/Vincent3D/index.html. Último acesso em 29/5/2008.

[53] Lighthouse 3D. *OpenGL Shading Language*. Disponível em http://www.lighthouse3d.com/opengl/glsl/index.php. Último acesso em 01/06/2008.

[54] André Luiz Battaiola, Rodrigo de G. Domingues, Bruno Feijó, Dilza Scwarcman, Esteban Walter G. Clua, Lauro Eduardo Kosovitz, Marcelo Dreux, Carlos André Pessoa, and Geber Ramalho. Desenvolvimento de Jogos em Computadores e Celulares. *Revista de Informática Teórica e Aplicada - RITA*, 8(2):7–46, 2001.

[55] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, August 1996.

[56] Samuel Ranta-Eskola. *BSP Trees: Theory and Implementation*. Disponível em http://www.devmaster.net/articles/bsp-trees/. Último acesso em 16/06/2008.

[57] Tommer Leyvand. *CityGen - A Procedural Model Generator*. Disponível em http://www.cs.tau.ac.il/ tommer/citygen/. Último acesso em 16/06/2008.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo