FUNDAÇÃO EDSON QUEIROZ

UNIVERSIDADE DE FORTALEZA - UNIFOR

Um Sistema para Navegação e Visualização Interativa em Ambientes Virtuais utilizando Celulares

Rafael Garcia Barbosa

Fortaleza

2007

FUNDAÇÃO EDSON QUEIROZ

UNIVERSIDADE DE FORTALEZA - UNIFOR

Um Sistema para Navegação e Visualização Interativa em Ambientes Virtuais utilizando Celulares

Rafael Garcia Barbosa

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: Profa. Dra. Maria Andréia Formico Rodrigues

Fortaleza

2007

Um Sistema para Navegação e Visualização Interativa em Ambientes Virtuais utilizando Celulares

Rafael Garcia Barbosa

Área de Concentração: Computação Gráfica

Linha de Pesquisa: Ambientes Virtuais para Dispositivos Móveis

Data de Aprovação: _____

Banca Examinadora:

_____

Profa. Dra. Maria Andréia Formico Rodrigues

_____

Prof. Dr. Geber Lisboa Ramalho

_____

Prof. Dr. Joaquim Bento Cavalcante Neto

AGRADECIMENTOS

Em primeiro lugar quero agradecer a Deus por me dar saúde, força e capacidade para desenvolver este trabalho.

Quero também agradecer a meus pais por me darem todo o apoio (financeiro e emocional) para que eu concluísse o mestrado, e a meus irmãos que também foram de grande ajuda neste processo.

Quero ainda agradecer a todos os meus amigos que, de alguma forma, contribuíram durante esses anos, auxiliando diretamente no desenvolvimento da dissertação ou, indiretamente, dando-me apoio para que eu conseguisse finalizá-la.

Em especial, quero agradecer imensamente à minha orientadora Maria Andréia Formico Rodrigues, sem a qual eu não teria conseguido chegar até aqui. Meu muito obrigado pelas noite perdidas produzindo artigos, corrigindo textos e fazendo pesquisa. Como você já está cansada de saber, você não é apenas uma grande orientadora, mas também uma segunda mãe e uma grande amiga.

Resumo

O aumento progressivo da capacidade de processamento gráfico e de armazenamento dos dispositivos móveis, especialmente telefones celulares e PDAs, bem como da capacidade de transmissão de dados das redes sem fio, apontam para um novo cenário de interação onde os usuários desses dispositivos poderão visualizar e explorar ambientes gráficos 3D independentemente de sua localização física. Exemplos de aplicações que podem se beneficiar com essas tecnologias são os guias virtuais e os jogos. Neste trabalho, nós usamos e estendemos uma API para o desenvolvimento de aplicações gráficas 3D em dispositivos móveis. Em particular, a API M3G, voltada à plataforma Java, foi explorada e estendida, adicionando-se novos recursos gráficos que não estavam originalmente disponíveis (algoritmos para detecção de colisão e planejamento de trajetórias). Esses recursos gráficos visam a visualização e navegação realista em ambientes virtuais interativos, utilizando-se celulares. Tais recursos foram agrupados em um ambiente gráfico, baseado na arquitetura proposta neste trabalho. Adicionalmente, três aplicações foram criadas para avaliar o desempenho do sistema em três plataformas diferentes: um emulador e dois telefones celulares (modelos w300i e w600i). Em particular, um número considerável de testes foi conduzido, baseado em uma das aplicações desenvolvidas. Os resultados obtidos mostram que os novos recursos foram implementados com sucesso, particularmente, o algoritmo para detecção de colisão, que apresentou um ótimo desempenho. Entre as plataformas analisadas, o modelo de celular w300i foi o que apresentou os melhores resultados em termos de desempenho. Finalmente, também provamos que os resultados obtidos com o emulador não são confiáveis.

Abstract

The continuous progress in graphics processing and memory capabilities of mobile devices, especially cell phones and PDAs, as well as in wireless networks data transmission capacity, lead to a new interactive graphics scenario where mobile users will be able to visualize and explore 3D environments, no matter their physical location. Examples of applications which can benefit from these technologies are virtual guides and games. We used and extended an API for the development of 3D applications using mobile devices. In particular, the M3G API, used by the Java plataform, was explored and extended, by adding new graphical resources that were not originally available (collision detection and path planning algorithms). These graphical resources aim at the realistic visualization and navigation through interactive virtual environments, using mobile phones. Such resources were gathered into a graphical system, which was based on an architecture proposed in this work. Additionally, three applications were created in order to evaluate the system performance in three different platforms: an emulator and two mobile phones (w300i and w600i models). In particular, a considerable number of experiments were conducted, based on one of the applications developed. The obtained results show that the new graphical resources were implemented with success, particularly, the collision detection algorithm, which presented excellent performance. Among the analysed platforms, the w300i cell phone model was the one that presented the best results in terms of performance. Finally, we also proved that the emulator results are untrustable.

Lista de Figuras

Sumário

Lista de Tabelas

Capítulo 1

Introdução

O aumento progressivo da capacidade de processamento gráfico e de armazenamento dos dispositivos móveis, especialmente telefones celulares e PDAs, bem como da capacidade de transmissão de dados das redes sem fio, apontam para um novo cenário de interação onde os usuários desses dispositivos poderão visualizar e explorar ambientes gráficos 3D, de forma individual ou colaborativa. Tais dispositivos e cenários são projetados para aumentar a eficiência e produtividade de profissionais em trânsito [1]. Jogos 3D, guias turísticos virtuais (em museus, *shoppings* e universidades) e sistemas para monitoramento remoto de pacientes são apenas alguns dos exemplos de aplicações que poderão se beneficiar desse novo cenário.

Apesar dos avanços tecnológicos significativos na área de mobilidade, os dispositivos móveis atuais ainda apresentam limitações importantes quando comparados aos tradicionais computadores pessoais: são tipicamente lentos; não são confiáveis como única fonte de armazenamento de dados; e possuem severas restrições com relação a mecanismos de interação com o usuário [2]. Em particular, um problema crítico enfrentado durante o desenvolvimento de aplicações gráficas 3D para dispositivos móveis é conseguir oferecer recursos sofisticados de visualização e navegação de forma eficiente, levando-se em conta a usabilidade e a portabilidade das mesmas, além dos limites de memória e de processamento dos diferentes modelos de dispositivo móvel. Além disso, restrições no consumo de energia (e a própria natureza intermitente do sinal das redes sem fio) impõem, aos usuários desses dispositivos, um modelo de operação onde a expectativa de perda das conexões de rede está sempre presente.

Embora as APIs para processamento gráfico 3D em dispositivos móveis sejam relativa-

mente recentes, os usuários de dispositivos portáteis têm exigido recursos gráficos cada vez mais sofisticados como, por exemplo, serviços 3D interativos. Adicionalmente, as versões iniciais das APIs gráficas voltadas para dispositivos móveis possuem ainda recursos computacionais limitados, às vezes, não oferecendo elementos básicos comumente utilizados em Computação Gráfica.

O desenvolvimento de aplicações gráficas 3D para dispositivos móveis, que levem em conta as restrições e os problemas apresentados anteriormente, é uma área de pesquisa recente e ainda pouco explorada por pesquisadores e desenvolvedores de *software*. Nesse contexto, há uma evidente demanda pelo desenvolvimento de aplicações que façam uso eficiente dos diferentes recursos tecnológicos disponíveis em cada tipo de dispositivo móvel, de tal forma a garantir a geração de implementações compactas e otimizadas.

Ambientes 3D interativos podem ser acessados, visualizados e navegados por usuários de dispositivos móveis em trânsito e, devido à própria natureza móvel destes dispositivos, estes ambientes já estão podendo ser obtidos a partir de servidores de alta capacidade. Diante deste cenário, a exploração e a análise das APIs gráficas 3D existentes é importante para proceder com a identificação da real capacidade dos recursos gráficos presentes nestas APIs (realismo tanto na visualização, quanto na qualidade da navegação no mundo 3D). A partir da análise detalhada destes recursos, modificações e extensões específicas na API M3G podem ser implementadas, de forma a possibilitar a criação de aplicações 3D interativas em dispositivos móveis, por exemplo, em telefones celulares.

Este trabalho tem como objetivo geral o desenvolvimento de um ambiente gráfico para navegação e visualização interativa em celulares. Com o foco voltado à plataforma Java, isto inclui a exploração de uma JSR (*Java Specification Request*) para manipulação de elementos gráficos 3D, a API M3G [3]. Por ser relativamente recente, a API M3G ainda apresenta limitações, como a falta de determinados recursos gráficos básicos para cálculos em Computação Gráfica. Possíves pontos de extensão da API M3G serão então definidos, com o objetivo de desenvolver melhorias e criar novas funcionalidades, colaborando para a geração de cenas mais realistas.

Vários objetivos específicos também podem ser especificados. Dentre eles, destacam-se:

- A implementação de classes com base na API M3G, criadas para disponibilizar novas funcionalidades;

- O projeto de uma arquitetura, para auxiliar na modularização dos componentes do sistema;
- A criação de um arcabouço (*framework*) para reunir esse conjunto de novos recursos e para ser reutilizado por novas aplicações gráficas;
- O desenvolvimento de aplicações, que visam testar e analisar os recursos atuais da API M3G, bem como os novos métodos implementados; e
- A execução e a análise de desempenho de uma das aplicações desenvolvidas em emuladores e em modelos de telefones celulares (w300i e w600i da Sony Ericsson).

O restante deste trabalho é organizado como se segue. No Capítulo 2 são apresentados os trabalhos existentes na área. Conceitos básicos são introduzidos no Capítulo 3. A API M3G é descrita genericamente no Capítulo 4. O sistema implementado é detalhado no Capítulo 5. No Capítulo 6, são apresentadas as aplicações desenvolvidas. No Capítulo 7 são conduzidos os experimentos e realizada a análise de desempenho do sistema. Conclusões e trabalhos futuros são apresentados no Capítulo 8.

Capítulo 2

Trabalhos Existentes

Passeios virtuais por museus, *shoppings*, universidades ou mesmo cidades têm demandado o desenvolvimento de aplicações gráficas bastante realistas, que permitam aos usuários navegar e explorar interativamente o ambiente virtual onde se encontram. Por exemplo, para adquirir informações detalhadas sobre itens de interesse presentes nestes cenários e selecionados pelos usuários. Embora diversos sistemas já tenham sido propostos para computadores pessoais tradicionais, tais sistemas ainda não foram amplamente desenvolvidos em dispositivos móveis (como celulares). Nestes dispositivos, os ambientes atualmente existentes não apresentam o uso de uma API 3D para celulares recente, como a M3G, ou não possuem a implementação de recursos comumente utilizados em ambientes de navegação, como o planejamento de trajetórias. A proposta deste trabalho é desenvolver um sistema para celulares baseado na API M3G, que disponibilize recursos de visualização e navegação em ambientes virtuais, reunindo métodos presentes na literatura para a implementação de tais recursos.

Em termos de posicionamento, dois tipos de ambientes virtuais podem ser definidos: os internos [4, 5, 6] e os externos [4, 7, 8, 9]. Os ambientes internos são representados por espaços fechados, contidos dentro de uma construção, como as salas de um museu ou o andar de um edifício. Por sua vez, os ambientes externos são representados por espaços abertos, como o campus de uma universidade ou o mapa de uma cidade. Neste trabalho, foi desenvolvido um sistema para navegação e visualização interativa em ambientes internos.

Considerando a navegação como o ato de se deslocar dentro de um ambiente virtual, o processo de familiarização do usuário com este ambiente é denominado de *wayfinding* [10]. O

wayfinding auxilia o usuário na construção do mapa cognitivo do ambiente virtual, através da utilização de uma ou mais técnicas (como mapas, placas, bússolas e rastro). Trombetta *et al*. [10] apresentam o uso dessas quatro técnicas, as quais são utilizadas da seguinte maneira: os **mapas** ilustram as salas e as portas que as conectam, além da posição atual do usuário e o seu objetivo; as **placas** indicam o caminho a ser seguido para se chegar ao objetivo; a **bússola** é representada por uma seta que sempre aponta para o objetivo; e o **rastro** é representado por setas que indicam por onde o usuário já passou.

Em sua maior parte, os trabalhos relacionados a serviços de navegação e visualização, tanto a nível bidimensional (2D), quanto tridimensional (3D), propõem o uso de mapas 2D que simulam a região a ser explorada. Em particular, Aboud *et al*. [4] propõem a visualização de mapas 2D, textos e páginas html. Mais especificamente, estes autores desenvolveram uma arquitetura para a visualização e a navegação em ambientes virtuais internos e externos. Diferentes modelos de dispositivos (PDAs e *pen-based* PCs) podem ser utilizados para aplicações de turismo, auxiliando o usuário (turista) na resposta a dúvidas específicas de busca por informações sobre o ambiente e na interação com o mesmo e com outras pessoas. O sistema, como um todo, funciona como um guia turístico, possuindo quatro funções extensíveis: o **cartógrafo** (representado pelo mapa do ambiente); o **bibliotecário** (que contém as informações sobre o ambiente); o **navegador** (responsável pelo mapeamento da posição física do usuário no ambiente 3D); e o **mensageiro** (que corresponde aos componentes de comunicação). Essa modularização permite a troca de implementação dos componentes com o mínimo impacto no resto do sistema. Um problema apresentado pelo autor é a necessidade de toda a informação estar armazenada no dispositivo, não podendo ser obtida remotamente.

Nos sistemas que utilizam mapas 2D, o nível de interatividade é geralmente mais restrito do que naqueles que utilizam mapas 3D, tendo em vista que os primeiros não oferecem suporte à criação de ambientes 3D sofisticados, nem à visualização ou exploração dos mesmos [7, 8]. Além disso, os mapas 2D são geralmente insuficientes no processo de familiarização do usuário com o cenário, uma vez que não incluem representações de objetos do mundo real que possam facilmente ser reconhecidas. Tais representações são importantes por serem tipicamente utilizadas como forma de orientação, por exemplo, quando se leva em consideração objetos com características visuais únicas. O uso de representações 3D do ambiente real supre esta insuficiência do mapas 2D e provê, aos visitantes de uma área não familiar, informações específicas sobre o local visitado e suas redondezas. Estas representações 3D, em sistemas de navegação, costumam ser mais próximas da realidade do que as 2D, tendendo a ser uma forma mais intuitiva de manipulação e entendimento do mapa. Exemplos são os trabalhos, apresentados em [11, 12], que oferecem suporte a cenas 3D realistas, tipicamente encontrados em ambientes virtuais disponíveis para computadores pessoais tradicionais.

Já o trabalho apresentado por Laakso *et al.* [13] permite que os usuários dos dispositivos móveis (também no papel de turistas virtuais) utilizem um mapa 3D para navegar, explorar e obter informações turísticas a partir do ambiente que eles estão visitando (uma cidade virtual). O usuário pode interagir com o sistema através de 4 setas: duas delas movem o modelo 3D para frente ou para trás e outras duas rotacionam o modelo 3D para esquerda ou para direita. Toda a informação necessária é armazenada localmente no dispositivo e uma seta principal mostra um destino buscado pelo usuário. O artigo é baseado em um projeto de pesquisa chamado TellMaris, cujo principal objetivo é prover aos cidadãos europeus um novo conjunto de serviços de informação através do uso de mapas 3D, avaliando sua usabilidade e determinando como estes mapas podem auxiliar turistas durante suas viagens e planejamento de feriados (pontos de interesse, como restaurantes e atrações podem também ser encontrados). Embora tenham desenvolvido mapas 3D, os autores não utilizaram tecnologias 3D recentes, como a API M3G, para a manipulação dos mesmos.

Segundo Nurminen [9], existem poucos estudos envolvendo mapas 3D reais em dispositivos móveis. Este autor desenvolve uma aplicação específica para dispositivos móveis com um rico conjunto de informações que podem ser obtidas do ambiente ou adicionadas a ele. Nesta aplicação, o usuário pode navegar por um espaço 3D (uma cidade), por exemplo, para reconhecer o cenário de um local previamente visitado ou que esteja visitando no momento presente. Além de evitar colisão com as construções, a aplicação possui ainda uma qualidade de renderização fotorealista e com taxas de renderização interativas. Para tal, o autor utiliza diversas técnicas de *culling* (descarte), além do uso de níveis de detalhes aplicados a texturas. Como o foco principal deste autor é possibilitar uma navegação manual realista pelo ambiente virtual, nenhum enfoque foi dado à navegação automática.

Já Bleschmied *et al.* desenvolveram uma tecnologia chamada CityServer3D para implementar novos conceitos e mecanismos para sistemas de informação geográfica 3D [14]. Baseado em um modelo de três camadas, o sistema provê fontes de dados que podem ser acessadas por diversas interfaces. Os modelos gerenciados são armazenados em um banco de dados, processados por componentes do servidor e distribuídos a diferentes clientes. Dentre estes clientes, encontra-se o Mobile3D Viewer, que permite a visualização de mapas 3D em dispositivos móveis. Adicionalmente, o Mobile3D Viewer possibilita que o usuário navegue livremente pelo ambiente, suportando objetos representados com diferentes níveis de detalhamento e provendo mecanismos de interação com o mapa. Embora o Mobile3D Viewer utilize a API M3G nos clientes (dispositivos móveis), o processamento mais custoso (como cálculos de rota entre a posição corrente do usuário e o destino desejado) é realizado do lado do servidor, utilizando o CityServer3D. Assim, apenas os resultados já processados pelo servidor são carregados e visualizados no dispositivo.

Usuários que navegam por ambientes desconhecidos ou mesmo muito amplos podem necessitar de auxílio para encontrarem determinados locais do ambiente que sejam de seu interesse. Assim, além de disponibilizar o método de navegação manual por ambientes virtuais (também conhecida como "modo de navegação manual", onde o usuário pode navegar livremente pelo ambiente virtual, explorando-o e, possivelmente, modificando-o), alguns autores oferecem um segundo método que auxilia neste processo de busca: uma função de planejamento de trajetórias (conhecida como "modo de navegação automática") [15, 16], geralmente controlada pelo sistema, com base em parâmetros que podem ser passados pelo usuário (como, por exemplo, os pontos de origem e de destino de uma determinada trajetória).

Segundo Lamarche e Donikian [17], os métodos utilizados para o planejamento de trajetórias são classificados em três tipos: o mapa de estradas, que contém um conjunto de trajetórias padronizadas do tipo linha/curva livre de obstáculos [18, 19, 16]; a decomposição de células, que consiste em decompor espaços livres em células, gerando um grafo de conectividade (onde os nós correspondem às células e as arestas correspondem às adjacências das células) [20]; e campos potenciais, que discretizam o ambiente em uma grade (*grid*) regular e associam um potencial a cada célula, cujo valor corresponde à soma do potencial repulsivo (gerado pelos obstáculos) e do potencial atrativo (gerado pelo objetivo) [21]. Juntamente com essas técnicas, outros autores utilizam também a técnica de *skeletonization* para extrair o "esqueleto"de um objeto 3D (ou ambiente) [22].

No trabalho apresentado por Li *et al*., que utiliza o mapa de estradas, o usuário pode especificar locais de interesse em um mapa 2D e, assim, permitir que o sistema gere automaticamente a animação do *tour* virtual [18]. Uma vez que os ambientes utilizados por Li *et al*. são geralmente construções arquitetônicas, os objetos que compõem a cena podem ser projetados em um espaço 2D. Para auxiliar no processo de geração de trajetórias, a geometria do *avatar* guia é aproximada por um volume envoltório esférico. O processo começa na posição inicial definida pelo usuário e prossegue até que todos os pontos de interesse tenham sido visitados pelo algoritmo, gerando assim a trajetória. Um ponto negativo é a utilização de ambientes virtuais relativamente simples. Embora também utilize um mapa de estradas [16] para criar um caminho livre de colisões entre dois pontos especificados pelo usuário, o trabalho apresentado por Salomon *et al* foca no uso de ambientes 3D complexos, como grandes fábricas compostas de diversos objetos em seu interior.

Andújar *et al*. utilizaram a decomposição de células para também gerar um caminho livre de colisões a partir de um ponto de origem, passando por pontos relevantes [20]. Uma diferença apresentada pelos autores, quando comparado aos trabalhos que também utilizam decomposição de células, consiste no fato de que os pontos relevantes não precisam ser selecionados pelo

usuário, uma vez que são definidos através do uso de uma métrica. Além disso, embora a busca por um caminho seja realizada de forma similar a outras já citadas (utilizando uma projeção 2D do ambiente), um grafo de células e portais é construído automaticamente, de forma a extrair a informação geométrica do cenário. Neste grafo, os nós (ou células) representam as regiões de espaço aberto, como salas presentes em uma construção; e as arestas (ou portais) representam os elementos que conectam essas regiões, geralmente definido como portas ou espaços abertos. Stout propôs a utilização do algoritmo A* para encontrar uma boa rota neste grafo [23]. Neste algoritmo, duas informações são levadas em consideração: o quanto já foi percorrido e uma heurística de quanto falta para chegar ao objeto.

O trabalho apresentado por Li *et al*. [21] descreve um método de campos potenciais, utilizando sensores que auxiliam usuários (pessoas, robôs) a encontrarem um caminho que evite áreas de risco (interpretadas como obstáculos). A idéia é movimentar o usuário através de forças de atração (que aproxima o objeto do objetivo) e de repulsão (que afasta o objeto do objetivo), gerando um caminho livre de obstáculos.

A maioria dos métodos propostos na literatura para extração do esqueleto de um elemento 3D são divididos em três classes: *boundary peeling* [24], *distance coding* [25] e diagramas de Voronoi [26]. Na técnica de *boundary peeling*, as camadas mais externas da imagem são extraídas interativamente, de forma que esta remoção não afete a topologia do objeto. Ao final do processo, são produzidos pontos conectados que definem o esqueleto do objeto. A segunda técnica extrai diretamente os pontos que formam o esqueleto do objeto, através de testes na vizinhança do objeto, com base na transformada da distância (uma aproximação da distância Euclidiana). Finalmente, a última técnica calcula o diagrama de Voronoi com base nos pontos que compõem a borda do objeto.

Para a realização de cálculos mais complexos (como o menor caminho em grafos) ou mesmo para obtenção de elementos 3D presentes nos ambientes virtuais a serem navegados, pode ser necessário estabelecer uma comunicação entre os celulares e um computador pessoal que funcione como servidor. Inúmeras abordagens têm sido propostas na literatura para transmissão de cenas 3D utilizando uma arquitetura cliente-servidor, como apresentado em [7, 8]. Apesar de alguns autores reconhecerem que a informação 3D deve ser provida sob demanda e relativa à posição/orientação do usuário, muitos deles não possuem uma infraestrutura de comunicação cliente-servidor, sendo que a aplicação 3D é executada diretamente no dispositivo [4, 13]. Neste trabalho, foi implementado um esquema bastante simples para solicitar a um servidor remoto, arquivos contendo a informação de cenas 3D ou mesmo cálculos mais complexos. Essa solicitação é feita sob demanda, à medida que o usuário necessita da informação solicitada. Qualquer protocolo de comunicação pode ser utilizado para transmissão destas in-

formações, como por exemplo o Wi-Fi, bastando estender o módulo de comunicação presente no sistema.

No próximo capítulo serão apresentados alguns conceitos básicos, fundamentais para a área de navegação e detecção de colisão.

Capítulo 3

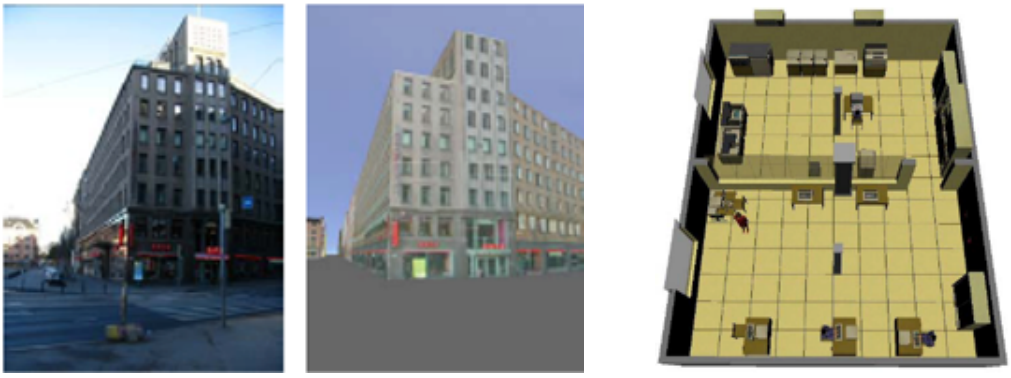
Conceitos Básicos: Navegação e Detecção de Colisão

Tendo em vista que o foco deste trabalho é a navegação e a visualização interativa em ambientes virtuais, é importante estudar conceitos básicos relacionados ao processo de navegação em si e, para aumentar o grau de realismo, ao processo de detecção de colisão entre objetos. As seções a seguir descrevem estes conceitos.

3.1 Navegação

Como visto anteriormente, para ambos os tipos de navegação, manual e automática, pode-se ainda classificar os ambientes visitados em: **internos**, que correspondem a interiores de contruções; e os **externos**, que correspondem a regiões exteriores, como cidades inteiras [27]. A Figura 3.1 exemplifica estes dois tipos de ambientes. Cada tipo de navegação e de ambiente pode possuir diferentes abordagens utilizadas para auxiliar o usuário durante seu processo de navegação.

Durante a navegação manual, o usuário costuma explorar o mundo 3D livremente, por qualquer lugar que ele desejar. Contudo, devido ao fato deste processo depender exclusivamente do usuário, isso pode ocasionar problemas, pois este pode encontrar-se sem referência de posição, em algum lugar do mapa que estava explorando. Uma técnica existente para evitar este tipo de



(a) (b)

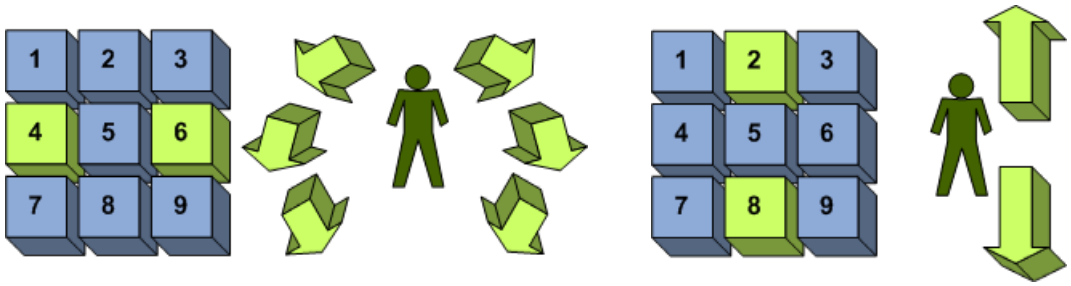
Figura 3.1: Em (a), exemplos de ambientes externos (extraído de [9]). Em (b), um exemplo de ambiente interno (extraído de [6]).

problema, utilizada em aplicações para computadores pessoais tradicionais, corresponde à utilização de marcos conhecidos, como objetos ou construções do mundo real, que são facilmente reconhecidos pelo usuário. Essa técnica é comumente utilizada em ambientes externos, pois, em geral, os usuários costumam conhecer pontos importantes que estão presentes nos lugares visitados, como estádios, *shoppings* e praças de cidades. Contudo, esta técnica também pode ser utilizada para interiores, caso o usuário tenha um conhecimento prévio do cenário no mundo real por onde ele está navegando virtualmente.

Técnicas de navegação por ambientes virtuais já vêm sendo aplicadas há bastante tempo nos computadores pessoais e, mais recentemente, também nos dispositivos móveis, que agora possuem suporte a APIs para processamento gráfico 3D. Com isso, o usuário pode navegar simultaneamente pelo ambiente virtual (utilizando o dispositivo móvel) e pelo ambiente real (caminhando pelo mesmo) [13]. Dependendo da semelhança entre os mundos virtual e real, este processo pode auxiliar o usuário no reconhecimento de elementos presentes no cenário, facilitando o seu uso em ambientes internos, que necessitariam de um conhecimento prévio dos mesmos para evitar que o usuário se encontrasse perdido.

Para realizar a navegação em ambientes virtuais, tanto em computadores pessoais quanto em celulares, é necessário mapear teclas específicas desses dispositivos para determinados comandos de navegação. Os tipos de comandos de navegação mais comuns são aqueles que utilizam as transformações geométricas de translação e rotação, movendo a representação virtual do usuário (*avatar*) pelo ambiente. Independentemente do dispositivo utilizado, é comum usar os dígitos de um teclado numérico para esta finalidade. Como exemplo, as teclas laterais (4 e 6) poderiam ser utilizadas para rotacionar o *avatar*, enquanto que as teclas superior e inferior (2 e 8, respectivamente) aplicariam uma translação para frente ou para trás, como pode ser visualizado na Figura 3.2. Assim, considerando que o *avatar* se move apenas em um plano

formado por dois eixos coordenados (por exemplo, *xz*), ele teria liberdade para se locomover para qualquer lugar do ambiente virtual.



(a) (b)

Figura 3.2: Em (a), um exemplo de rotação. Em (b), um exemplo de translação.

Dos métodos existentes para a navegação automática, os que usam mapas de estradas consistem no cálculo de uma rede de caminhos padronizados (linhas, curvas) que passam por espaços abertos [18, 16]. Diferentes abordagens podem ser utilizadas para calcular esta rede, como a criação de um grafo de visibilidade [16] ou o cálculo de um diagrama de Voronoi dentro dos espaços vazios [18].

Nos métodos que utilizam campos potenciais, o espaço vazio é discretizado em uma malha regular e um valor potencial é associado a cada célula. A função potencial é tipicamente definida como a soma de um potencial de repulsão (gerado por obstáculos presentes no ambiente) e um potencial de atração (gerado pelo objetivo). Campos potenciais podem ser combinados com técnicas de busca em grafos, de forma a definir um caminho até o objetivo [28, 29].

Por sua vez, os métodos que realizam a decomposição de células dividem os espaços vazios em células e geram um grafo de conectividade. Para realizar a decomposição do espaço, pode-se utilizar: a **decomposição exata de células**, de forma que a sua união corresponda exatamente ao espaço livre (triangulação de *Delaunay*, polígonos convexos) [28]; e a **decomposição aproximada de células**, que consiste em usar modelos predefinidos, cuja união está estritamente incluída no espaço vazio (*quadtrees*) [28, 30, 31, 32].

Em particular, o método que realiza a decomposição de células é composto por dois passos principais: o particionamento do espaço em células e a geração do grafo de conectividade. A Figura 3.3 ilustra um exemplo de particionamento de um ambiente e seu respectivo grafo de conectividade.

No grafo da Figura 3.3b, um nó representa uma célula convexa e uma aresta representa um segmento livre compartilhado por duas células adjacentes, com uma tamanho maior que a

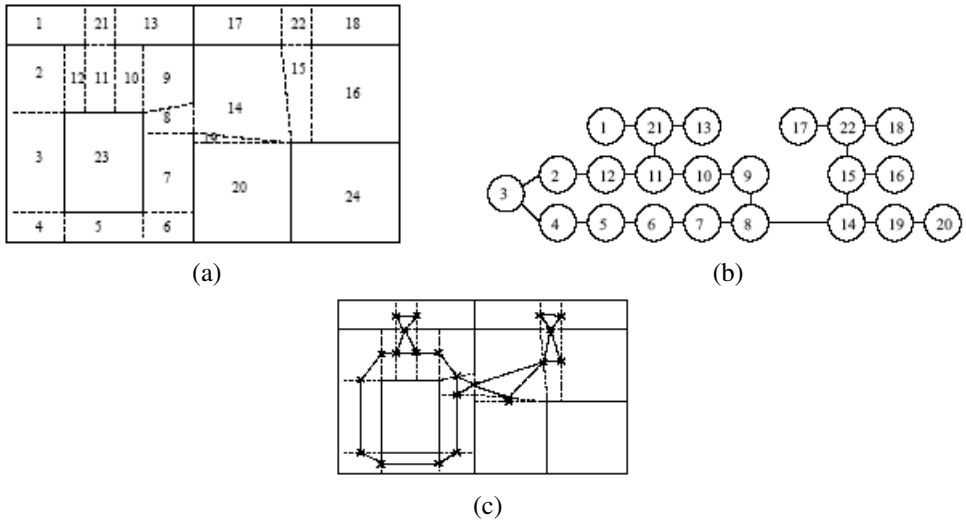


Figura 3.3: Em (a), um exemplo de particionamento de um ambiente. Em (b), seu respectivo grafo de conectividade e, em (c) o conjunto de segmentos que conectam as células (extraído de [17]).

largura do *avatar*. Considerando que não há obstáculos no ambiente e que as células são convexas, pode-se sempre gerar segmentos lineares que passem pelas células, conectando-as. Para cada célula, pontos-chave são gerados em arestas livres e são conectados através de segmentos lineares, como ilustra a Figura 3.3c. Supondo que k pontos-chave são gerados em cada um dos n segmentos livres pertencentes aos limites de uma dada célula c, o número de caminhos gerados dentro desta célula é dado por $n(n-1)k^2$. Isso pode ser demonstrado tomando-se uma célula da Figura 3.3c que possua três arestas livres. Neste caso, o valor de n é 3 e, como visualizado na figura, o valor de k é 1 (somente um ponto-chave foi gerado para cada aresta livre). Assim, o número de caminhos gerados é $3*(2)*1^2 = 6$, sendo que cada aresta livre possui dois caminhos que conectam-se às outras arestas, totalizando o valor de 6 caminhos, como obtido pela fórmula.

Para finalizar, o Algoritmo 1 pode ser utilizado para obter o caminho resultante, fornecido como uma sequência de segmentos livres [17]. Para isso, supõe-se que C_e é a célula que contém o *avatar*, C_g é a célula que contém o objetivo e S é a célula abstrata de mais alto nível que contém C_e e C_g. O Algoritmo inicia numa célula abstrata de mais alto nível, descendo em suas células de mais baixo nível (linhas 4, 6 e 11), até chegar nas células C_e e C_g (linha 8).

Algoritmo 1 Criação de caminhos que conectam o *avatar* e o objetivo

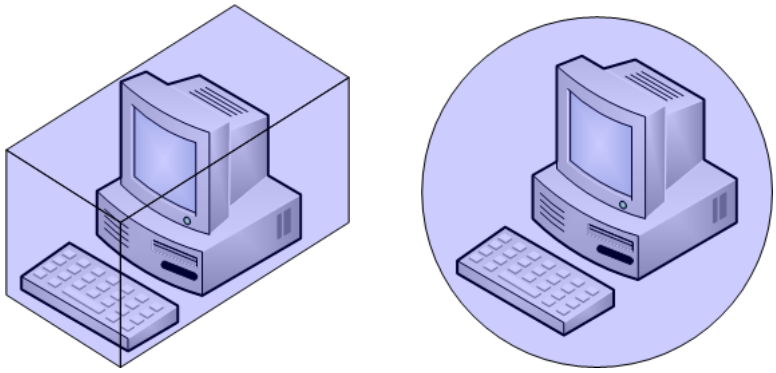
Entrada: Célula abstrata de mais alto nível S
Saída: Células necessárias para o planejamento de trajetórias $Cels$
void path(S)
1: $Cels \leftarrow Temp \leftarrow 0$
2: **while** $S \neq 0$ **do**
3: **while** $S \neq 0$ **do**
4: $c \leftarrow pickElement(S)$ // remove um elemento do conjunto S
5: **if** $(C_e \in c \wedge C_e \neq c) \vee (C_g \in c \wedge C_g \neq c)$ **then** // Identifica se chegou na célula de mais baixo nível
6: $Temp \leftarrow Temp \cup split(c)$ // O método *split* retorna um conjunto contendo as células de mais baixo nível que compõem a célula c
7: **else**
8: $Cels \leftarrow Cels \cup \{c\}$
9: **end if**
10: **end while**
11: $S \leftarrow Temp$
12: $Temp \leftarrow 0$
13: **end while**
14: **return** $Cels$

3.2 Detecção de Colisão

Ao se trabalhar com detecção de colisões, a primeira consideração a ser feita está relacionada ao tipo de representação geométrica das cenas e de seus objetos constituintes. Malhas poligonais, por exemplo, oferecem uma maior quantidade de informações sobre os objetos, quando comparadas a um conjunto de polígonos sem informações de conexão entre eles (conhecido como *polygon soup*) [33]. Estas informações podem ser utilizadas para melhorar o desempenho dos algoritmos de detecção de colisão. Por sua vez, o uso de geometria sólida construtiva (CSG) pode facilitar os testes de inclusão de um objeto em outro, porém, encapsula o acesso direto aos vértices, arestas e faces das primitivas geométricas que constituem os objetos. Após definida a representação geométrica, pode-se selecionar os algoritmos de detecção de colisão mais apropriados.

Outra consideração importante está relacionada à informação geométrica que será utilizada durante o processo de detecção de colisão. Embora seja possível passar a geometria de um objeto diretamente para o sistema de colisão, esta pode ser complexa demais para este processo. Além disso, geralmente, existe um limite de quão precisas as colisões devem ser. Assim, para minimizar os custos em testes de intersecção, é comum substituir a geometria do objeto por uma versão mais simplificada da mesma. Por exemplo, volumes envoltórios (como caixas e esferas) podem ser utilizados para encapsular a geometria de um objeto, como ilustrado na Figura 3.4. Um volume envoltório corresponde a um volume único que encapsula um ou mais objetos de natureza mais complexa.

Nem todos os objetos geométricos são apropriados para serem utilizados como volumes envoltórios. Para tal, os mesmo devem apresentar algumas propriedades desejáveis, como:

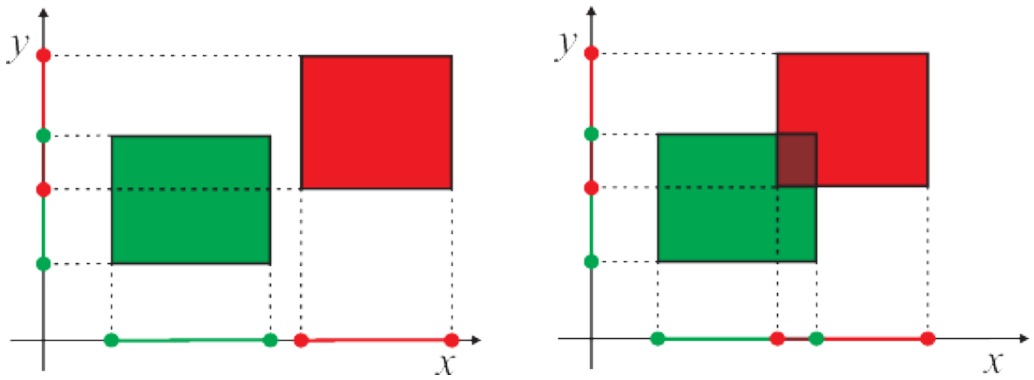


(a) (b)

Figura 3.4: Exemplos de volumes envoltórios cúbico (a) e esférico (b).

processamento e testes de intersecção rápidos, bom ajuste à forma geométrica do objeto, fácil aplicação de transformações geométricas e uso de pouca memória. Essas características em geral tendem a reduzir o tempo de processamento durante os testes de inclusão e intersecção, utilizando pouca memória para isso. A criação dos volumes envoltórios não costuma impactar no desempenho do sistema, uma vez que é realizada em tempo de pré-processamento. Um exemplo de volume envoltório são as caixas alinhadas com os eixos (AABBs) [33].

As caixas alinhadas com os eixos são um dos volumes envoltórios mais utilizados. Elas correspondes a caixas retangulares (de seis lados em 3D e quatro em 2D), cujas faces são orientadas de forma tal que as normais das faces são sempre paralelas aos eixos de um dado sistema de coordenadas. A melhor característica de uma AABB é a sua rápida checagem de intersecção, que envolve simplesmente uma comparação direta de valores individuais de coordenadas. A Figura 3.5a ilustra a não-ocorrência de intersecção, enquanto que a Figura 3.5b ilustra a ocorrência da mesma. A checagem analisa apenas os extremos de cada objeto, para cada eixo coordenado, e a intersecção só ocorre no caso dos objetos transpassarem em todos os valores coordenados.



(a) (b)

Figura 3.5: Intersecção entre caixas alinhadas. Em (a) a intersecção não ocorre porque apenas os valores do eixo y dos objetos foi transpassado. Em (b) a intersecção ocorre porque ambos os valores dos eixos x e y foram transpassados.

A Figura 3.6 mostra as três representações mais comuns para as caixas alinhadas com os eixos. Na Figura 3.6a, os valores de coordenadas máximo e mínimo são definidos, já na Figura 3.6b, o valor de coordenada mínimo e o diâmetro. Na Figura 3.6c, o ponto central e o raio são definidos. Em termos de espaço requerido em memória, a primeira representação (Figura 3.6a) tem desempenho inferior, pois necessita armazenar 6 valores de coordenadas, enquanto que as outras duas representações (Figuras 3.6b e 3.6c) necessitam apenas de 4 valores. A terceira representação (Figura 3.6c) tem desempenho superior à segunda (Figura 3.6b), tendo em vista que o armazenamento do raio exige um número menor de *bits*, do que o armazenamento exigido na representação do diâmetro. No caso do objeto sofrer apenas translações, o desempenho da segunda e terceira representações (Figuras 3.6b e 3.6c) também é melhor, uma vez que somente é necessário atualizar 3 valores (x, y e z do ponto utilizado como coordenada mínima ou como ponto central).

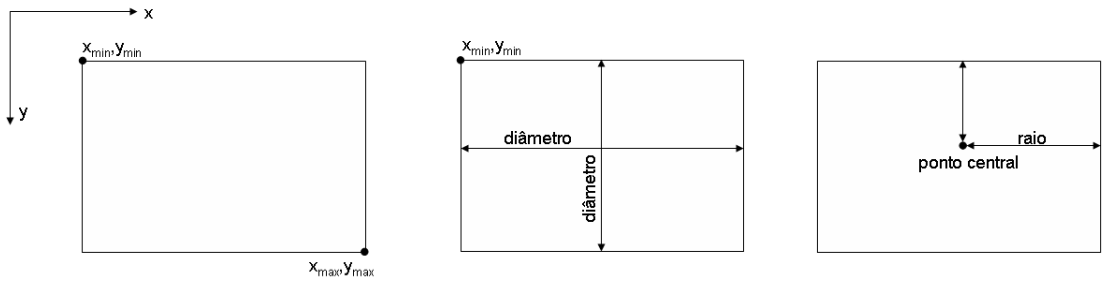


Figura 3.6: Três representações de caixas alinhadas aos eixos.

Os testes de intersecção utilizados pelo sistema de detecção de colisões dependem do tipo de representação da caixa alinhada escolhida. Embora a representação de máximo e mínimo seja a que mais ocupa espaço em memória, é a que apresenta a menor quantidade de operações durante os testes de intersecção. O Algoritmo 2 mostra o código utilizado para o teste [33].

Algoritmo 2 Teste de Intersecção de uma AABB: recebe duas AABBs e checa se houve intersecção entre elas

Entrada: As AABBs a e b
Saída: Indicação se houve intersecção
boolean interseccaoAABB(a, b)
1: **if** $a.max[0] < b.min[0] || a.min[0] > b.max[0]$ **then** // Checa se algum dos eixos não possui intersecção
2: **return false**
3: **end if**
4: **if** $a.max[1] < b.min[1] || a.min[1] > b.max[1]$ **then**
5: **return false**
6: **end if**
7: **if** $a.max[2] < b.min[2] || a.min[2] > b.max[2]$ **then**
8: **return false**
9: **end if**
10: **return true** // Se houve sobreposição em todos os eixos, então houve intersecção

As linhas 1, 4 e 7 do Algoritmo 2 checam, respectivamente, se não há intersecção entre dois objetos nos eixos x, y e z. Caso haja intersecção nos três eixos, a linha 10 retorna a indicação da ocorrência da colisão entre os objetos.

Apesar da facilidade na sua criação e da simplicidade de seus testes de intersecção, uma caixa alianhada aos eixos precisa ser realinhada após a aplicação de uma rotação à mesma, ocorrida durante movimentos no objeto que originou a caixa envoltória (isso acontece porque a caixa alinhada aos eixos fica desalinhada após a rotação).

Existem quatros estratégias mais comuns para realinhar a caixa envoltória: utilizar uma caixa alinhada aos eixos de volume suficiente para sempre envolver o objeto (contudo, uma abordagem similar e melhor seria utilizar uma esfera envoltória); calcular uma caixa alinhada aos eixos reconstruída dinamicamente a partir do conjunto de pontos originais (esta estratégia pode requerer muito poder de processamento e memória); calcular uma caixa alinhada aos eixos reconstruída dinamicamente usando a estratégia de *hill climbing* [33] (que necessita de objetos convexos para funcionar e vai sempre em busca do próximo vértice a partir de um inicial); e, por último, calcular uma caixa alinhada aos eixos aproximada, reconstruída dinamicamente a partir da caixa alinhada aos eixos que foi rotacionada (esta estratégia simplesmente envolve a caixa alinhada aos eixos rotacionada em uma nova caixa alinhada aos eixos). Esta última abordagem é a mais comumente utilizada, gerando uma caixa alinhada aos eixos aproximada, maior do que a original.

Algoritmo 3 Calcula uma nova AABB, a partir de uma AABB rotacionada

Entrada: A AABB a e a matriz de rotação m
Saída: A nova AABB b
AABB atualizacaoAABB(a, m)

```
1:  for i = 0 to 3 do
2:      for j = 0 to 3 do
3:          float e ← m[i][j] * a.min[j]
4:          float f ← m[i][j] * a.max[j]
5:          if e < f then
6:              b.min[i] ← b.min[i] + e
7:              b.max[i] ← b.min[i] + f
8:          else
9:              b.min[i] ← b.min[i] + f
10:             b.max[i] ← b.min[i] + e
11:         end if
12:         return b
13:     end for
14: end for
```

O Algoritmo 3 detalha como calcular uma nova caixa alinhada aos eixos, a partir de uma caixa alinhada aos eixos rotacionada. Inicialmente, aplica-se a rotação aos extremos da caixa alinhada aos eixos, como mostrado nas linhas 1 a 4. Para cada novo valor obtido, inicialmente deve-se encontrar o maior e o menor valor. Depois disso, somá-los aos valores de máximo e mínimo da nova caixa alinhada aos eixos a ser gerada (linhas 5 a 10). Ao final, a nova caixa alinhada aos eixos é retornada (linha 12).

No próximo capítulo serão introduzidas as principais características da API M3G, incluindo algumas de suas limitações e possibilidades de extensão.

Capítulo 4

A API *Mobile 3D Graphics* (M3G)

A API M3G (*Mobile 3D Graphics*), também conhecida como JSR-184 (*Java Specification Request*) [3], é uma API de alto-nível, escalável e que exige pouco espaço em memória para a sua execução. Sendo proposta para a plataforma Java, corresponde a um pacote gráfico 3D opcional, que deve ser utilizado em conjunto com a plataforma Java Micro Edition (JME) para dispositivos móveis [34]. A configuração mínima necessária exigida pela API para seu correto funcionamento é a CLDC 1.1 [35], podendo utilizar quaisquer dos perfis atualmente existentes: MIDP 1.0 ou 2.0 [36, 37]. Esta API é voltada para dispositivos que tipicamente possuem pouca memória, poder de processamento reduzido, e sem suporte em *hardware* para processamento gráfico 3D ou aritmética de ponto-flutuante como, por exemplo, telefones celulares e PDAs. Entretanto, foi desenvolvida de forma a se beneficiar de dispositivos que apresentem unidade de ponto flutuante ou mesmo *hardware* gráfico 3D especializado.

A versão 1.0 da especificação da API M3G (sob a JSR 184) e a implementação de referência, lançadas em 2003 e 2004 [3, 38], respectivamente, passaram a ser utilizadas pelos desenvolvedores na geração de suas aplicações gráficas. Em 2005, uma versão atualizada da especificação (versão 1.1) foi lançada, juntamente com a sua implementação de referência, permitindo a execução de operações que não estavam disponíveis na primeira versão da API como, por exemplo, obter os vértices das malhas dos objetos gerados [38]. Contudo, esta versão ainda não encontra-se disponível nos dispositivos atuais. Atualmente, encontra-se em desenvolvimento a versão 2.0 da especificação da API M3G, sob a JSR-297, com previsão de lançamento para o segundo semestre de 2007 [39]. O grande foco no desenvolvimento desta versão é o uso de aceleradores gráficos 3D programáveis e a melhoria no desempenho geral das aplicações.

A Figura 4.1 mostra as principais camadas de uma arquitetura típica, disponível em dispositivos móveis que têm suporte à API M3G [40]. Como pode ser observado neste modelo de arquitetura, a API M3G foi desenvolvida de tal forma a se beneficiar da implementação da API OpenGL ES [41], com a qual deve ser compatível, permitindo que ambas utilizem o mesmo motor de renderização. A API M3G disponibiliza ainda uma implementação própria da API OpenGL ES, para o caso do dispositivo não possuir uma implementação da mesma disponível.

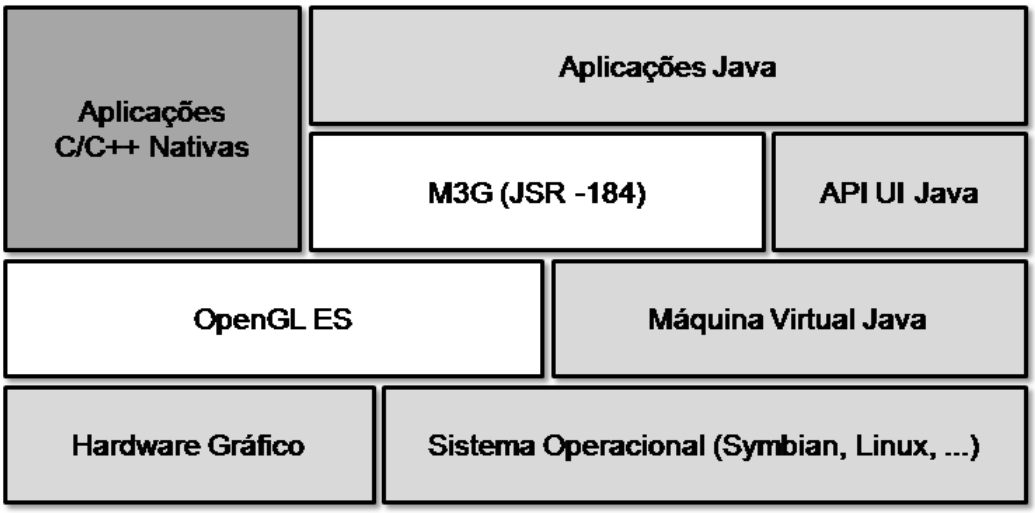


Figura 4.1: Arquitetura em dispositivos móveis com suporte à API M3G (extraído de [39]).

Como pode ser visto na arquitetura, aplicações desenvolvidas através do uso da tecnologia Java devem utilizar M3G para usufruir dos recursos 3D disponibilizados nesta API. Em particular, o uso da plataforma Java permite que uma mesma aplicação possa executar em dispositivos e sistemas operacionais diferentes. A API JME, que trata da interface com o usuário (UI API), pode ser utilizada para prover recursos específicos para a aplicação, tais como, a definição de um local onde desenhar uma cena 3D, ou mesmo, a inserção de elementos gráficos 2D na cena. A API M3G pode ainda fazer acesso indireto ao sistema operacional e/ou ao *hardware* gráfico, caso disponível, utilizando a OpenGL ES.

A API M3G é composta por 30 classes, que possibilitam a utilização das principais operações realizadas em Computação Gráfica para a manipulação e visualização de objetos 3D. Dentre as funcionalidades presentes na mesma, destacam-se: o suporte à criação e à manipulação de animações, a capacidade de utilização de cálculos de iluminação, a definição de tipos de materiais, texturas e parâmetros de visualização, o disparo de raios de intersecção, e o carregamento de arquivos contendo informações geométricas de objetos e cenas 3D [42, 3].

Sendo uma das classes mais importantes, a Graphics3D é responsável pela renderização geral das cenas, encapsulando o *pipeline* de renderização dentro dela. Com o objetivo de prover

funcionalidades para os diferentes tipos de aplicações gráficas, a API M3G disponibiliza dois modos de renderização: ***retained mode*** (que renderiza um grafo de cena completo) e ***immediate mode*** (que permite renderizar grupos de objetos separadamente). Em particular, o ***immediate mode*** foi desenvolvido para funcionar de forma similar à renderização do OpenGL [43], enquanto que o ***retained mode*** funciona de forma similar à renderização do Java3D [44]. Contudo, nada impede que estes dois modos de renderização possam ser usados conjuntamente.

Através do uso de um grafo de cena, a API permite representar objetos e seus atributos de visualização. A utilização deste grafo permite ao usuário definir toda a estrutura do ambiente gerado, apenas agrupando o conjunto de objetos a serem visualizados, bem como os seus estados correntes. Tal grafo pode ser projetado e exportado para um formato de arquivo conhecido, disponibilizado pela própria API M3G. Este formato, chamado m3g, encapsula toda a informação do ambiente virtual criado em uma ferramenta de modelagem geométrica que tenha tal recurso, como por exemplo o 3ds [45].

A despeito do fato de ser chamado de grafo, o grafo de cena é, na verdade, uma estrutura do tipo árvore. Isso indica que um nó só pode pertencer a, no máximo, um grupo por vez, e não deve haver ciclos no mesmo. A Figura 4.2 exibe um exemplo de grafo de cena contendo algumas das principais classes da API M3G.

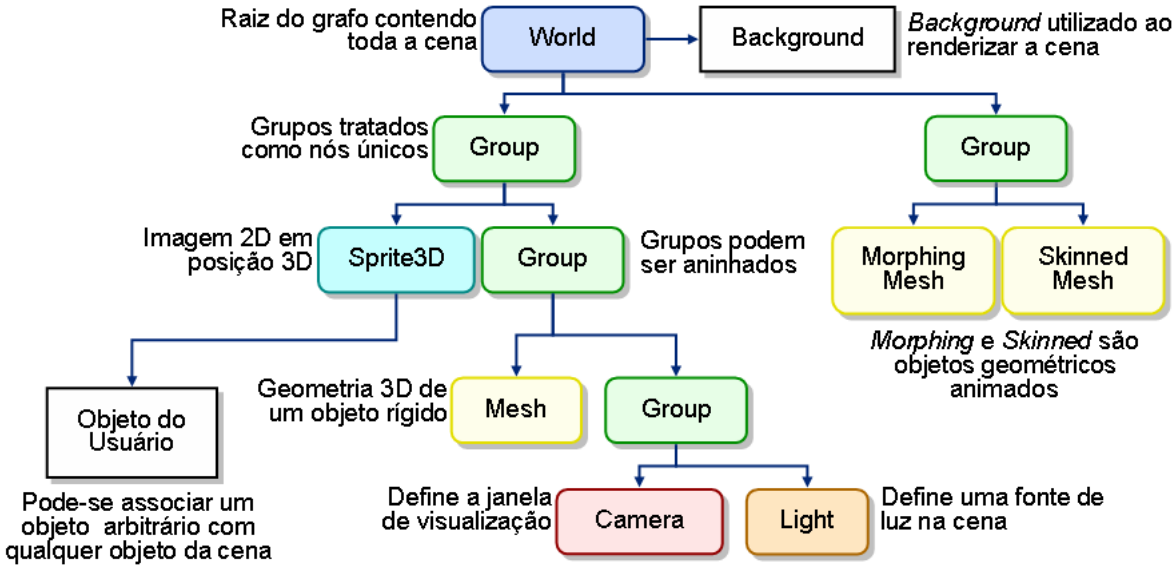


Figura 4.2: Exemplo de grafo de cena da API M3G.

A raiz do grafo é representada pela classe World, que pode ser passada diretamente ao método de renderização da classe Graphics3D (***retained mode***). As classes Node, Mesh, MorphingMesh e SkinnedMesh representam os elementos gráficos que compõem uma cena. Estes objetos são definidos por vértices, utilizando-se as classes VertexBuffer, VertexArray,

IndexBuffer e TriangleStripArray, que indicam além dos vértices, a forma de entrada dos mesmos e o modo como eles são conectados. A primeira versão da API M3G (1.0) não disponibiliza o acesso direto aos vértices, e uma vez que eles tenham sido armazenados nos objetos acima citados, eles não podem mais ser obtidos diretamente. Embora a versão 1.1 da API M3G disponibilize tais vértices, ela ainda não encontra-se disponível nos dispositivos móveis atuais.

A classe Group permite agrupar diversos elementos gráficos em um só, possibilitando, por exemplo, a aplicação de transformações ao grupo inteiro, ao invés de aplicá-las individualmente a cada objeto. Entretanto, as transformações geométricas aplicadadas a um grupo ficam armazenadas no mesmo e só são propagadas para seus filhos durante a renderização. Assim, a transformação geométrica de um objeto (Mesh) corresponde, na verdade, à composição da sua transformação geométrica e das transformações geométricas armazenadas nos grupos que estão acima dele, fazendo parte da hierarquia deste objeto.

As transformações que podem ser aplicadas aos grupos e objetos individuais são controladas pelas classes Transform e Transformable. A primeira representa uma matriz de transformação que pode ser composta de várias transformações geométricas aplicadas de uma vez a um objeto. Já a segunda faz parte da hierarquia de classes dos objetos (sendo uma super-classe da classe Mesh), indicando que eles podem sofrer transformações geométricas diretamente. As classes que representam as transformações geométricas são fundamentais para permitir a **navegação manual** por ambientes virtuais, pois é através das mesmas que podemos locomover o *avatar* pelo cenário 3D. Não há classes atualmente disponíveis para auxiliar na **navegação automática** nos processos de particionamento de ambientes virtuais e geração do grafo de conectividade.

Como mostra a Figura 4.2 (nó folha mais inferior à esquerda), objetos do usuário podem também ser associados a qualquer objeto geométrico presente no grafo de cena. Isso é útil, por exemplo, em jogos que necessitam associar informações ao *avatar* (como itens disponíveis, quantidade de vida, etc).

Pode-se ainda modificar características dos objetos como, por exemplo, a transparência e a ordem de entrada dos vértices, utilizando-se as classes CompositingMode e PolygonMode. Texturas e materiais podem ser utilizados através das classes Texture2D e Material, respectivamente. Todas estas classes que afetam a aparência dos objetos podem ser aplicadas ao elemento Appearance dos mesmos. A iluminação é controlada pela classe Light, que permite o uso de luzes do tipo ambiente, *spot*, direcional e omni.

Cenas podem ser carregadas a partir da leitura de um arquivo m3g através da classe Loader, responsável também pela obtenção dos dados armazenados neste arquivo, como as malhas geométricas dos objetos e seus atributos, texturas, hierarquias da cena, propriedades dos materiais, quadros-chave das animações, etc. Através da classe Camera, os parâmetros de visualização de uma cena podem ser controlados (por exemplo, os planos próximo e distante, a razão de aspecto, o ângulo de visualização e o tipo de projeção da câmera sintética - paralela ou perspectiva).

Recursos adicionais para o controle de animações são disponibilizados através de três classes básicas: AnimationController, AnimationTrack e KeyFrameSequence. Estas classes de animação podem ser úteis para a navegação automática, exibindo uma animação que leva o *avatar* do seu ponto de origem ao ponto de destino. Entretanto, para que isso ocorra é necessário que o caminho a ser percorrido pelo *avatar* tenha sido previamente gerado, por exemplo, através do uso de um grafo de conectividade, inexistente na versão atual da API M3G.

Existem ainda classes específicas para a geração de planos de fundo (classe Background), efeitos especiais do tipo fumaça (Fog), e imagens 2D no mundo 3D (Sprite3D e Image2D). A primeira pode ser aplicada diretamente ao mundo virtual (classe World), a segunda faz parte da aparência dos objetos (classe Appearance) e a terceira representa um objeto que pode ser adicionado diretamente ao grafo de cena.

Adicionalmente, raios de intersecção podem ser calculados para auxiliar o controle do movimento da câmera e na detecção de colisão através da classe RayIntersection. Contudo, considerando-se uma cena onde objetos em movimento colidem entre si ou com objetos estáticos da cena, o uso de raios de intersecção, para a detecção de colisões em tempo de execução, pode tornar-se inviável. Isso ocorre porque cada objeto dinâmico pode precisar disparar não somente um, mas vários raios de interseção, em diversas direções, uma vez que as colisões entre os objetos podem acontecer em qualquer um dos eixos coordenados x, y e z. O uso de raios de interseção apresenta ainda outra limitação: estes podem falhar ao detectar um possível ponto de colisão, devido ao número limitado de raios que podem ser disparados, a partir de cada objeto. Uma implementação de envoltórios, inexistente na API M3G, melhoraria o desempenho no cálculo da colisão.

No próximo capítulo serão apresentados os detalhes do sistema proposto e implementado neste trabalho.

Capítulo 5

O Sistema Proposto e Implementado

Neste capítulo será detalhado o sistema que foi proposto e implementado neste trabalho para visualização e navegação em ambientes virtuais, utilizando pequenos dispositivos móveis, mais especificamente, telefones celulares. As próximas seções apresentam a arquitetura do sistema, os métodos implementados para navegação e detecção de colisão, e o esquema utilizado para a transmissão de dados.

5.1 Arquitetura do Sistema

Neste trabalho, o sistema foi desenvolvido tendo como base a arquitetura mostrada na Figura 5.1. Nesta arquitetura, quatro camadas principais podem ser destacadas: a **Camada de Comunicação**, o **Modelo M3G**, o **Controlador do Dispositivo** e as **Visões** [46, 47]. Esta divisão foi realizada com base no padrão arquitetural MVC (*Model-View-Controller*), que separa os dados, os meios de visualização destes dados e a forma de interação com o sistema [48]. Esta separação é importante para que futuras extensões possam, por exemplo, adicionar novas formas de visualização dos dados sem necessitar que o restante do sistema seja modificado, bastando incluí-las na camada de visões. Além disso, esta organização facilita a troca ou atualização de determinados componentes do sistema, por exemplo, o mecanismo de detecção de colisões pode ser alterado modificando-se apenas o **Modelo M3G**. Embora não faça parte do padrão MVC, neste trabalho, a **Camada de Comunicação** foi incluída na arquitetura para separar o restante da aplicação do protocolo de comunicação utilizado.

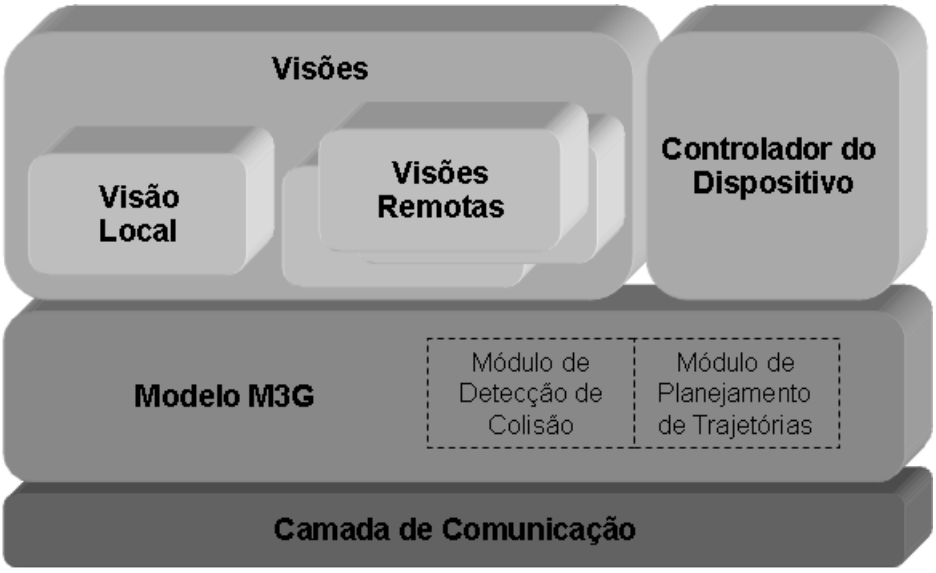


Figura 5.1: Arquitetura do sistema baseada no padrão de *software* MVC.

Uma das funções do **Modelo M3G** é armazenar todos os dados utilizados em tempo de execução. Assim, durante o processo de inicialização do sistema, o **Modelo M3G** é acessado para definir a configuração inicial de variáveis e objetos geométricos a serem utilizados. O cenário 3D inicial, por exemplo, é obtido a partir de um servidor remoto por meio da **Camada de Comunicação**, que foi ativada pelo **Modelo M3G**. Cada cenário pertecente ao mundo virtual pode ser solicitado separadamente ao servidor remoto. Uma vez obtido o cenário inicial, a camada **Visão** é então ativada para exibí-lo.

Ao efetuar interações com o sistema (para transladar o avatar ou solicitar a criação de uma trajetória até um dado ponto), o usuário ativa o **Controlador do Dispositivo**. O **Controlador do Dispositivo** é responsável por traduzir todo tipo de interação em chamadas de métodos presentes no **Modelo M3G**. Qualquer modificação no estado do **Modelo M3G** irá disparar uma atualização na **Visão**, de forma a sempre mantê-la em sincronia com o estado do **Modelo M3G**. Isso é garantido através do uso de outro padrão de projeto [48], chamado Observador, que faz com que a visão fique atenta a qualquer alteração no **Modelo M3G**.

A camada **Visão** é composta de duas outras camadas: a **Visão Local** e as **Visões Remotas**. A primeira representa a tela do dispositivo que é visualizada pelo usuário local, enquanto que a segunda representa imagens do ambiente local visualizadas remotamente. As **Visões Remotas** podem ser úteis, por exemplo, em extensões no sistema que façam uso de colaboração, mostrando a usuários remotos as atualizações no cenário local.

O **Modelo M3G** é ainda responsável por toda a lógica da aplicação, abrigando, por exem-

plo, os módulos de detecção de colisão e planejamento de trajetórias. Através do módulo de detecção de colisão, o **Modelo M3G** impede que aconteçam colisões com os objetos do ambiente 3D. Assim, caso um movimento de navegação viole restrições impostas pelo mecanismo de detecção de colisões, o **Modelo** não irá atualizar nem seu estado, nem as **Visões**. Utilizando o módulo de planejamento de trajetórias, o **Modelo M3G** disponibiliza um caminho que leve o usuário da sua posição original até um ponto específico escolhido pelo mesmo. Por ser um cálculo que demanda tempo, e recursos computacionais, ele é efetuado em um servidor remoto. Para isso, o **Modulo M3G** solicita a geração deste caminho à **Camada de Comunicação**, que, por sua vez, acessa o servidor remoto responsável por este processamento. Uma vez gerado o caminho, o servidor o retorna à **Camada de Comunicação**, que o repassa para o **Modelo M3G**. De posse do caminho, o **Modelo** pode gerar uma sequência de quadros (animação) a ser exibida pela **Visão**, representando a navegação automática.

A **Camada de Comunicação** realiza basicamente a troca de informações entre os dispositivos e o servidor remoto, que contém os cenários 3D e a lógica para a geração de trajetórias. Esta camada foi criada para esconder do **Modelo M3G** os detalhes específicos da tecnologia de rede utilizada. Da forma como foi modelada, esta camada é flexível ao uso de novas tecnologias para transmissão de dados, fazendo com que o protocolo de comunicação possa ser facilmente substituído por outro que venha a emergir futuramente.

5.2 Métodos Implementados

Nas próximas seções serão apresentados os métodos implementados neste trabalho para a navegação manual pelo ambiente virtual, a navegação automática (através da geração de planejamento de trajetórias) e o tratamento de colisões com objetos do cenário 3D [46, 49, 50].

5.2.1 Navegação Manual

Para efetuar movimentos manuais, o usuário deve interagir com as teclas do dispositivo móvel (neste trabalho, telefones celulares). Como sugerido no Capítulo 3, as teclas 2 e 8 movimentam o *avatar* para frente e para trás, respectivamente. Enquanto que as teclas 4 e 6 rotacionam o *avatar* para a esquerda e para a direita, respectivamente.

Para efetuar estes movimentos de translação e rotação, foi necessário sobrescrever métodos atualmente presentes na classe Canvas (por exemplo, os métodos keyPressed e keyReleased). Em particular, a classe Canvas (parte integrante da API JME) é responsável, tanto pela renderização do cenário virtual, quanto pela resposta a eventos disparados quando o usuário interage com o dispositivo. No sistema implementado, a classe Renderer herda da classe Canvas e sobrescreve tais métodos. Estes métodos são detalhados nos Algoritmos 4 e 5, respectivamente.

Algoritmo 4 Método chamado quando o usuário pressiona uma tecla

Entrada: O código da tecla pressionada *keyCode*
void keyPressed(keyCode)
1: $this.keyCode \leftarrow keyCode$ // Armazena a tecla pressionada
2: $keyRepeated \leftarrow$ **true** // Ativa o estado de repetição
3: $keyRepeatedThread.wakeup()$ // Ativa a *thread* que movimenta o avatar

O Algoritmo 4 é responsável por realizar a movimentação do *avatar*, que pode se manter contínua, enquanto o usuário pressionar a tecla. Inicialmente, a tecla pressionada é armazenada para ser posteriormente utilizada (linha 1). Em seguida, uma chave é ativada para indicar que a tecla ainda encontra-se pressionada (linha 2). Enquanto esta tecla estiver pressionada, o movimento continuará a ser executado. Por último, a *thread* responsável pelo movimento do avatar é então ativada, de forma que a transformação geométrica correspondente é aplicada ao mesmo.

Algoritmo 5 Método chamado quando o usuário solta uma tecla que havia pressionado

Entrada: O código da tecla que foi solta *keyCode*
void keyReleased(keyCode)
1: $keyRepeated \leftarrow$ **false** // Desativa o estado de repetição

O Algoritmo 5 é responsável por parar o movimento iniciado pelo Algoritmo 4. A linha 1 deste algoritmo desativa a chave anteriormente ativada pelo Algoritmo 4, indicando que o movimento não deve mais ser continuado.

O Algoritmo 6 apresenta a *thread* responsável pela execução da transformação geométrica. O método principal de execução desta *thread* possui um laço infinito que efetua uma sequência de passos detalhados a seguir. Inicialmente, a *thread* é desativada para esperar por uma interação por parte do usuário (linha 3). Quando o usuário interage com o dispositivo, o sistema chama o Algoritmo 4 que reativa a *thread*. Em seguida, a transformação geométrica solicitada pelo usuário é executada (linha 5 do Algoritmo 6). Por último, a tela do dispositivo é atualizada para refletir as mudanças decorrentes da transformação geométrica (linha 6).

A camada de controle utiliza a chave da tecla pressionada para definir qual transformação geométrica deve ser aplicada, como detalhado no Algoritmo 7. Caso a tecla pressionada seja a 2 (*UP*) ou 8 (*DOWN*), o avatar deve sofrer uma translação. Caso a tecla pressionada seja a 4 (*LEFT*) ou a 6 (*RIGHT*), o avatar deve sofrer uma rotação.

Algoritmo 6 Método principal da *thread* que realiza movimentos no *avatar*

```
void run()
1: while true do
2:     if !keyRepeated then
3:         this.wait()
4:     end if
5:     control.keyEvent(keyCode) // Chama a camada de controle responsável por qualquer interação
6:     repaint() // Atualiza a tela do dispositivo
7: end while
```

Algoritmo 7 Método que define qual transformação geométrica deve ser aplicada

Entrada: O código da tecla pressionada *keyCode*

```
void keyEvent(keyCode)
1: if keyCode = Canvas.UP or keyCode = Canvas.DOWN then
2:     model.translateCamera(gameAction) // Aplica uma translação
3: end if
4: if keyCode = Canvas.LEFT or keyCode = Canvas.RIGHT then
5:     model.rotateCamera(gameAction) // Aplica uma rotação
6: end if
```

O último passo para a aplicação de transformações geométricas ao *avatar* corresponde à chamada dos métodos que, efetivamente, aplicarão estas transformações. A forma como as transformações geométricas são aplicadas depende do sistema de coordenadas utilizado. A Figura 5.2 exibe o sistema de coordenadas utilizado pela API M3G e o plano xz pelo qual o *avatar* pode se locomover. Imaginando que o avatar encontra-se no centro dos eixos de coordenadas, olhando em direção ao eixo z negativo, as rotações em torno do eixo y acontecem positivamente para a esquerda e negativamente para a direita. Assim, movimentos de translação para frente deverão ocorrer com valores negativos sendo aplicados aos eixos x e z, enquanto que movimentos para trás deverão ocorrer com valores positivos em x e z (Algoritmo 8). Para provar isso, basta analisar o quadrante formado por $-z$ e $-x$, na Figura 5.2. Neste caso, o ângulo é positivo, o seno é aplicado ao eixo x e o cosseno é aplicado ao eixo z. Como os valores do seno e do cosseno de um ângulo positivo são também positivos, as translações para frente (linha 2) devem acrescentar valores negativos em z e/ou em x. O mesmo ocorre para os outros quadrantes do eixo de coordenadas.

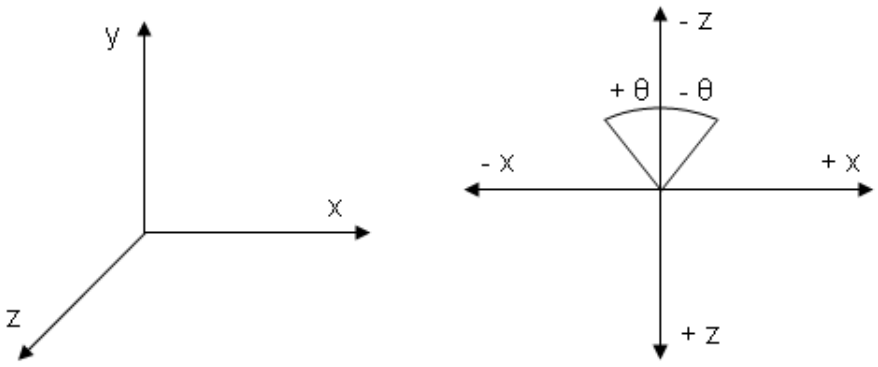


Figura 5.2: Sistema de coordenadas e plano xz visto de cima.

Para permitir que a translação aconteça sempre para frente ou para trás de onde o *avatar* está olhando, é necessário armazenar o ângulo de rotação do *avatar*. Isso pode ser observado nas

Algoritmo 8 Método que translada o avatar

Entrada: O código da tecla pressionada *keyCode*
void translate(keyCode)
1: **if** keyCode = Canvas.UP **then** // Translação para frente
2: $avatar.translate(-x * sin\Theta, 0, -z * cos\Theta)$
3: **end if**
4: **if** keyCode = Canvas.DOWN **then** // Translação para trás
5: $avatar.translate(x * sin\Theta, 0, z * cos\Theta)$
6: **end if**

linhas 2 e 6 do Algoritmo 9. Sempre que o usuário pressionar a tecla que representa a translação para frente, o *avatar* deve rotacionar no sentido positivo de rotação do eixo *y*, somando um valor ao ângulo de rotação. No caso inverso, ao pressionar a tecla que representa a translação para trás, um valor é subtraído do ângulo de rotação. Além desses valores utilizados para calcular o ângulo atual em que se encontra o *avatar*, é necessário armazenar também o ângulo que ele acabou de rotacionar (linhas 3 e 7 do Algoritmo 9). Esse ângulo é utilizado pelo algoritmo para efetivamente aplicar a rotação ao *avatar* (linha 9). O restante do Algoritmo 9 calcula os valores do seno e do cosseno do ângulo de rotação atual do *avatar*.

Algoritmo 9 Método que rotaciona o avatar

Entrada: O código da tecla pressionada *keyCode*
void rotate(keyCode)
1: **if** keyCode = Canvas.LEFT **then** // Rotação para a esquerda
2: $newAngle = newAngle + y$ // Armazena o valor utilizado na translação
3: $rotAngle = y$ // Define o valor fixo de rotação para a direita
4: **end if**
5: **if** keyCode = Canvas.RIGHT **then** // Rotação para a direita
6: $newAngle = newAngle - y$ // Armazena o valor utilizado na translação
7: $rotAngle = -y$ // Define o valor fixo de rotação para a esquerda
8: **end if**
9: $avatar.postRotate(rotAngle, 0, 1, 0)$ // Aplica a rotação em torno do eixo *y*
10: double $rads = Math.toRadians(newAngle)$ // Calcula o valor em radianos do ângulo atual
11: $sin\Theta = Math.sin(rads)$ // Calcula o seno do ângulo atual
12: $cos\Theta = Math.cos(rads)$ // Calcula o cosseno do ângulo atual

5.2.2 Navegação Automática

Neste trabalho, foi também implementado um algoritmo de planejamento de trajetórias (*path planning*) para permitir que o usuário navegue de forma automática por ambientes virtuais. Por não ser o foco deste trabalho, o particionamento dos ambientes virtuais apresentados será efetuado de forma manual. Entretanto, serão abordados detalhadamente os conceitos básicos relacionados à geração do grafo de conectividade.

A abordagem utilizada para geração do grafo de conectividade é similar àquela que usa um mapa do ambiente virtual e que gera um grafo baseado neste mapa, introduzida no Capítulo 3. Contudo, para simplificar o trabalho de geração do grafo, o mesmo foi criado utilizando-se os recursos disponíveis na API M3G. Por exemplo, com os nós representando os portais, as arestas do grafo poderiam ser obtidas através do disparo de raios de intersecção entre os mesmos, o que

motivou o uso de uma abordagem alternativa para a construção do grafo. Nesta abordagem alternativa, os nós do grafo correspondem aos portais que conectam as células e as arestas correspondem ao espaço vazio entre os portais, ou seja, as células em si. Com o grafo definido desta forma, pode-se disparar raios de intersecção entre as portas (nós do grafo) e conectá-las quando possivel (através das arestas). No caso de diferentes salas de uma mesma construção serem carregadas, cada uma possuirá um sub-grafo. A Figura 5.3 mostra um exemplo deste grafo, na qual, os pontos de interesse do usuário também fazem parte do grafo, na forma de novos nós.

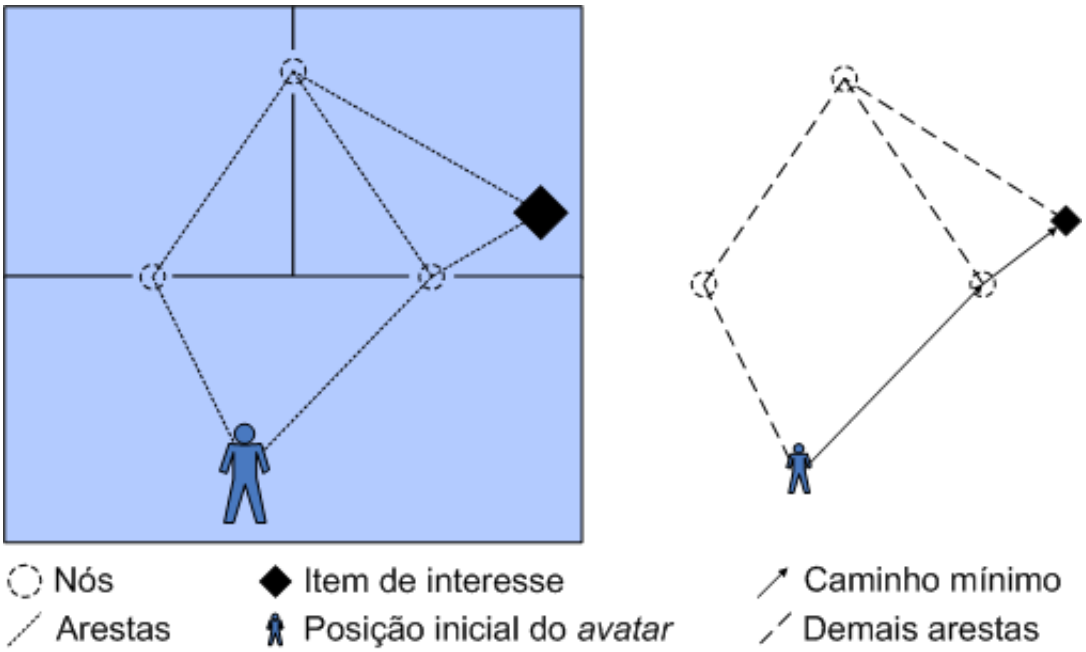


Figura 5.3: Grafo de células interconectadas.

O primeiro passo na construção do grafo de conectividade é a identificação dos portais presentes no mundo virtual. Além de representarem os nós do grafo, são também utilizados para encontrar as arestas que ligam os portais, dois a dois. Estes portais são facilmente obtidos através de identificadores, associados previamente aos mesmos durante a modelagem geométrica dos cenários (por exemplo, realizada no 3ds [45]). Uma vez carregado um cenário, seus portais podem ser obtidos navegando-se pela estrutura do grafo de cena na qual a API M3G é baseada.

O Algoritmo 10 ilustra o processo desenvolvido para a construção do grafo, que inicia-se com o disparo de raios partindo de cada portal, em direção aos demais. No caso do raio atingir seu portal alvo, uma nova aresta é criada no grafo, conectando os dois portais (linha 7). O peso da aresta que une dois portais é relativo à distância euclidiana entre os mesmos. Se um raio atinge outro elemento que não seja o portal alvo, por exemplo, um obstáculo, uma busca é realizada para encontrar um caminho que conecte os dois portais (linha 9). Ao final do processo,

o menor caminho pode ser obtido através da aplicação de um algoritmo de otimização (neste trabalho, não foi realizada uma pesquisa extensiva dos algoritmos de otimização, tendo sido escolhido o algoritmo de Dijkstra [29], cuja complexidade é $O([m+n]logn)$). Em particular, o "menor caminho" corresponde ao problema de encontrar um caminho entre dois vértices, de forma que a soma dos pesos de seus vértices seja a menor possível. Ao final deste processo, o algoritmo implementado neste trabalho gera o grafo contendo o menor caminho entre dois portais, livre de colisões.

Algoritmo 10 Criação do grafo de conectividade

Entrada: Um vetor de portais *portals*
Saída: Um grafo de conectividade
Graph createGraph(portals)
1: Graph $graph = new$ Graph()
2: **for** $i = 0$ to $portals.length$ **do**
3: **for** $j = 0$ to $portals.length$ **do**
4: **if** $isNotLinked(portals[i], portals[j])$ **then** // Checa a conexão entre os portais
5: Element $hit \Leftarrow castRay(portals[i], portals[j])$ // Dispara um raio de um portal a outro
6: **if** $hit == portals[j]$ **then** // Verifica se o raio atingiu seu portal alvo
7: $graph.add(portals.get(i), portals.get(j))$ // Adiciona uma aresta ao grafo
8: **else**
9: $findPath(graph, portals.get(i), portals.get(j), hit)$ // Busca um caminho alternativo
10: **end if**
11: **end if**
12: **end for**
13: **end for**
14: $setShortPath(graph)$ // Calcula o menor caminho
15: **return** *graph*

Para encontrar um caminho alternativo, que seja livre de obstáculos e que conecte dois portais, neste trabalho foi utilizada uma abordagem através da qual o *avatar* irá circundar o obstáculo. Para tal, pequenos cubos são adicionados nos arredores do obstáculo, sendo também inseridos no grafo de conectividade, como ilustrado na Figura 5.4. A menor dimensão dos cubos utilizada para permitir que o raio de intersecção o atingisse foi de valor 4 (este valor foi obtido empiricamente). Valores menores que este causaram falhas no cálculo do raio de intersecção, devido a erros de aproximação. O processo de criação dos cubos é detalhado no Algoritmo 11. Inicialmente, são criados os cubos na vizinhança do elemento atingido, o obstáculo (linha 1). Para cada cubo, dois novos raios são disparados: um do portal de origem, em direção ao cubo; e outro deste cubo em direção ao portal de destino (linhas 2 e 3, respectivamente). Caso ambos os raios atinjam seus alvos, duas novas arestas são adicionadas ao grafo (linhas 6 e 13). Se algum dos raios não atingir seu objetivo durante este processo, realiza-se uma recursão, até que sejam encontrados um ou mais caminhos entre os portais de origem e destino (linhas 8 e 15), respectivamente. Uma vez que todos as trajetórias possíveis tenham sido geradas, pode-se utilizar novamente o algoritmo de Dijkstra para obter o menor caminho que conecta os portais, como mostrado na Figura 5.4d. Este caminho será aquele que deve ser percorrido pelo *avatar*.

No sistema, os passos necessários para a criação do grafo de conectividade são efetuados em tempo pré-processamento, sendo necessária apenas uma pequena atualização do mesmo, em tempo de execução, caso o usuário precise adicionar novos pontos de interesse ao grafo (por

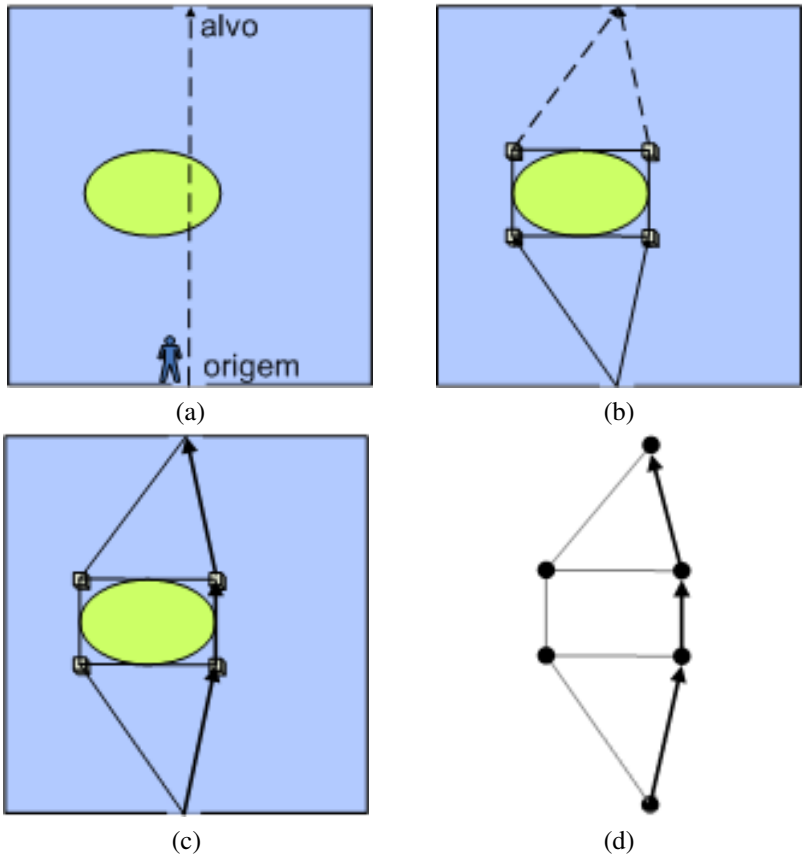


Figura 5.4: Busca por um caminho alternativo livre de colisão.

exemplo, sua posição corrente no mundo virtual). Estes novos pontos são incluídos de forma que apenas o sub-grafo da sala onde o avatar está contido precisa ser atualizado. Um problema existente atualmente nesta abordagem é o fato de que os pontos incluídos pelo usuário não são removidos do grafo. Assim, à medida que novas buscas são realizadas, novos pontos são incluídos no grafo, aumentando sua complexidade. Em trabalhos futuros, esses pontos deverão ser removidos.

5.2.3 Detecção de Colisão

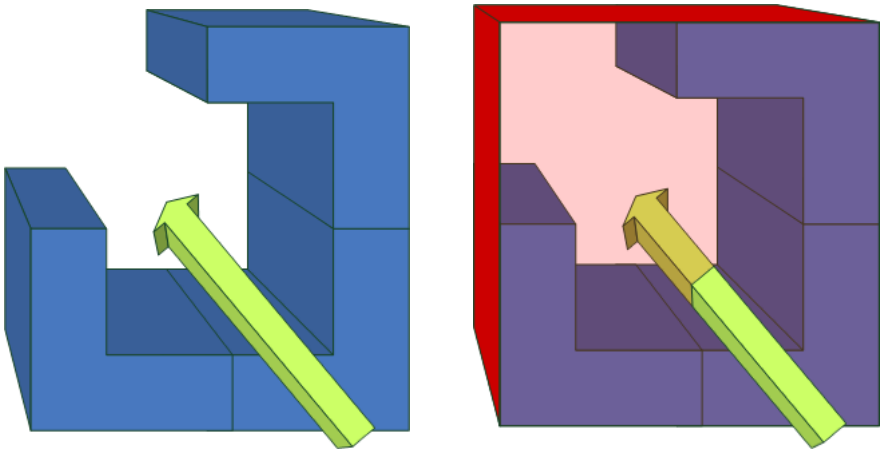
Apesar de ser um recurso extremamente importante para garantir o realismo da navegação em ambientes virtuais, a detecção de colisão entre objetos 3D não possui uma implementação direta na versão atualmente disponível da API M3G [3]. Contudo, a API disponibiliza uma implementação (pouco robusta) de raios de intersecção que pode ser utilizada, entre outras coisas, para detectar colisões bastante simples entre objetos. Nesse caso, para cada objeto em movimento no ambiente virtual, é necessário disparar um ou mais raios de intersecção, partindo dele próprio. Entretanto, cada objeto em movimento pode precisar disparar não somente um,

Algoritmo 11 Busca por um caminho alternativo livre de colisões

Entrada: Um grafo *graph*, o portal de origem *portalSrc*, o de destino *portalDest* e o obstáculo *obstacle*
void findPath(graph, portalSrc, portalDest, obstacle)

```
1:  Cube[] bricks ⇐ createNeighboorCubes(obstacle) // Cria pequenos cubos ao redor do obstáculo
2:  Element[] hitFromSrc ⇐ castRay(portalSrc, bricks) // Dispara um raio da origem aos cubos
3:  Element[] hitToDest ⇐ castRay(bricks, portalDest) // Dispara um raio dos cubos ao alvo
4:  for i = 0 to hitFromSrc.length do
5:      if hitFromSrc[i] == bricks[i] then // Verifica se os raios atingiram algum cubo
6:          graph.add(portalSrc, bricks[i])
7:      else
8:          findPath(graph, portalSrc, bricks[i], hitFromSrc[i])
9:      end if
10: end for
11: for i = 0 to hitToDest.length do
12:     if hitToDest[i] == portalDest[i] then // Verifica se os raios atingiram o alvo
13:         graph.add(bricks[i], portalDest)
14:     else
15:         findPath(graph, bricks[i], portalDest, hitToDest[i])
16:     end if
17: end for
18: setShortPath(graph) // Calcula o menor caminho
```

mas vários raios de intersecção (inclusive, muitas vezes, em diversas direções), uma vez que as colisões entre os objetos podem acontecer em qualquer um dos eixos coordenados x, y e z. Além do cálculo destes diversos raios ocasionar uma degradação no desempenho do sistema, esses raios podem falhar ao detectar pontos de colisão. Isso pode ocorrer, por exemplo, em casos onde os objetos sejam vazados (como ilustrado na Figura 5.5), ou em casos onde se deve limitar o número de raios que podem ser disparados a partir de cada objeto, com o objetivo de não degradar, além de um certo limiar, o desempenho do sistema. Como consequência destas limitações, torna-se importante a implementação de métodos na API M3G que permitam detectar colisões de forma mais robusta, garantindo um mínimo de desempenho do sistema e um maior realismo na navegação.



(a) (b)

Figura 5.5: Raio de intersecção disparado em objeto vazado. Em (a), o objeto não é atingido. Em (b), a caixa envoltória do objeto é atingida.

No sistema implementado, caixas envoltórias foram utilizadas para representar a geometria dos objetos 3D, por ser um método relativamente simples e possível de ser implementado em celulares (com limitações de memória e poder de processamento) [33]. Em particular, a

abordagem escolhida utiliza os valores extremos dos objetos para definir o volume envoltório (cúbico). Esta abordagem tem como principal vantagem o fato do teste de intersecção entre volumes envoltórios cúbicos, utilizado pelo algoritmo de detecção de colisão, ser rápido. Porém, este tipo de abordagem pode não ser recomendável para aproximar certas geometrias (por exemplo, aquelas que possuem elementos longos e pontiagudos, fazendo com que a caixa ocupe um grande espaço que não sofre colisão), sendo necessário o uso de outros métodos em conjunto com as caixas ou de outros envoltórios.

O primeiro passo para a utilização deste método, consiste na obtenção dos vértices do objeto a partir do qual será criado o volume envoltório cúbico. Contudo, uma séria limitação da versão 1.0 da API M3G, e que se tornou um problema clássico, é que os vértices dos objetos não estão disponíveis. Consequentemente, uma forma alternativa para obtenção dos mesmos foi proposta e implementada neste trabalho, como detalhado no Algoritmo 12. Atualmente, apesar da versão 1.1 da API M3G já disponibilizar estes vértices, esta ainda não se apresenta disponível nos pequenos dispositivos móveis atuais.

Algoritmo 12 Obtenção dos vértices de um objeto: recebe uma malha como entrada e retorna os seus vértices

Entrada: A malha da qual serão obtidos os vértices *mesh*
Saída: Vértices da malha
float[] getVertices(mesh)
1: VertexBuffer $vertexBuffer \leftarrow mesh.getVertexBuffer()$ // Obtém o VertexBuffer da malha geométrica do objeto
2: float[] *scaleBias* ← *new* float[4] // Vetor que irá armazenar os valores de *scale* e *bias*
3: VertexArray $positionArray \leftarrow vertexBuffer.getPositions(scaleBias)$ // Obtém os valores de scale e bias
4: float[] *out* ← *new* float[vertexBuffer.getVertexCount()*4]; // Vetor que irá armazenar os vértices do objeto
5: Transform *trans* ← *new* Transform()
6: $trans.setIdentity()$
7: $trans.postTranslate(scaleBias[1], scaleBias[2], scaleBias[3])$ // Aplica uma translação usando o valor de *bias* obtido
8: $trans.postScale(scaleBias[0], scaleBias[0], scaleBias[0])$ // Aplica uma escala usando o valor de *scale* obtido
9: $trans.transform(positionArray, out, true)$ // Aplica as transformações geométricas ao vetor resultado
10: **return** *out*

Na linha 1 do Algoritmo 12, o VertexBuffer que contém os vértices da malha é obtido. Uma vez que os vértices precisam ser definidos em valores inteiros, é necessário obter valores de *scale* e *bias* que posicionarão os vértices em coordenadas ponto-flutuante. Estes valores são obtidos na linha 3 e armazenados no vetor criado na linha 2. A linha 3 também obtém o objeto VertexArray, que contém os vértices da malha. Para se aplicar os valores de *scale* e *bias* ao vetor de vértices, deve-se criar um objeto Transform (linha 5), aplicar os valores de *bias* (linha 7) e *scale* (linha 8), e aplicar esta transformação ao vetor de vértices (linha 9). O vetor contendo os vértices nas suas coordenadas corretas é então retornado na linha 10.

Uma vez de posse dos vértices dos objetos, pode-se então calcular os valores extremos dos vértices de cada objeto, em cada um dos eixos coordenados x, y e z. Para isso, deve-se percorrer todos os vértices dos objetos, guardando sempre seus valores máximo e mínimo. O Algoritmo 13 detalha como este cálculo pode ser efetuado.

Algoritmo 13 Obtenção dos vértices extremos de uma malha: recebe os vértices da malha como entrada e obtém os valores extremos

Entrada: Os vértices de uma malha *vertices*
Saída: Vértices extremos da malha *lower* e *upper*
void obterExtremos(vertices)

```
1:  lower ← new Point()
2:  upper ← new Point() // Inicializa as variáveis máximo e mínimo com os primeiros valores do vetor de vértices
3:  lower.x ← upper.x ← out[0]
4:  lower.y ← upper.y ← out[1]
5:  lower.z ← upper.z ← out[2] // Percorre o vetor e obtém os extremos
6:  for n = 0 to vertices.length/4 do
7:      if upper.x < out[n*4] then // Analisa os maiores valores do vetor de vértices
8:          upper.x ← out[n*4]
9:      end if
10:     if upper.y < out[n*4+1] then
11:         upper.y ← out[n*4+1]
12:     end if
13:     if upper.z < out[n*4+2] then
14:         upper.z ← out[n*4+2]
15:     end if
16:     if lower.x > out[n*4] then // Analisa os menores valores do vetor de vértices
17:         lower.x ← out[n*4]
18:     end if
19:     if lower.y > out[n*4+1] then
20:         lower.y ← out[n*4+1]
21:     end if
22:     if lower.z > out[n*4+2] then
23:         lower.z ← out[n*4+2]
24:     end if
25: end for
```

Os valores obtidos dos vértices (máximo e mínimo) de um objeto serão utilizados para a construção da caixa envoltória deste objeto. Uma vez calculada a caixa envoltória para todos os objetos presentes no ambiente virtual, o sistema estará pronto para detectar possíveis colisões entre objetos. A checagem de ocorrência de colisões deve ser ativada sempre que houver algum tipo de transformação geométrica na cena 3D.

Tendo em vista que o sistema foi desenvolvido para permitir a navegação por ambientes virtuais interativos, a implementação atual permite que apenas o *avatar* movimente-se pelo cenário (portanto, os demais objetos são estáticos). Dessa forma, neste trabalho, a checagem da detecção de colisão ocorre durante os movimentos de rotação e translação que o usuário efetua no ambiente virtual. Neste caso, os valores máximo e mínimo armazenados na caixa envoltória do avatar são analisados para checar se estes ultrapassaram os valores máximo e mínimo do volume envoltório de outros elementos geométricos da cena. O método responsável por analisar se houve uma colisão entre os envoltórios de dois objetos é similar ao exibido no Algoritmo 2 e é detalhado no Algoritmo 14.

O Algoritmo 14 analisa se os extremos do avatar não ultrapassam os extremos de outro objeto (passado como parâmetro ao método). No caso da ocorrência de uma colisão, o tratamento é bastante simples: o avatar é retornado ao estado anterior ao da detecção da colisão.

Algoritmo 14 Análise de intersecção entre dois objetos: recebe os vértices da malha como entrada e obtém os valores extremos que serão utilizados para a geração do volume envoltório

Entrada: A caixa envoltória do objeto com o qual se deseja testar a colisão *bounding*
Saída: Indicação de ocorrência ou não de colisão *out*
boolean intersect(bounding)
1: **if** $this.calculatedUpper.x \geq bounding.calculatedLower.x$ & $this.calculatedLower.x \leq bounding.calculatedUpper.x$ & $this.calculatedUpper.y \geq bounding.calculatedLower.y$ & $this.calculatedLower.y \leq bounding.calculatedUpper.y$ & $this.calculatedUpper.z \geq bounding.calculatedLower.z$ & $this.calculatedLower.z \leq bounding.calculatedUpper.z$ **then** // Analisa se houve intersecção entre algum elemento geométrico, nos três eixos coordenados
2: **return true**
3: **else**
4: **return false**
5: **end if**

Uma vez que a detecção de colisão utilizando caixas envoltórias realiza apenas um teste dos valores extremos das coordenadas do objeto, esta geralmente caracteriza-se como um processo menos custoso do que o disparo de raios de intersecção (por exemplo, no caso de haver poucos elementos geométricos a serem testados na cena). Adicionalmente, esta estratégia é flexível o suficiente para detectar colisões quando o objeto em movimento (o *avatar*) se deslocar para frente, para os lados, ou mesmo para trás, ou seja, em qualquer um dos eixos coordenados.

Para permitir que os objetos que representam as malhas (Mesh) e os grupos (Groups) passassem a ter os seus volumes envoltórios associados a eles, neste trabalho foram implementadas novas classes: BoundedMesh e BoundedGroup (herdeiras de Mesh e Group), respectivamente. Ambas as classes BoundedMesh e BoundedGroup implementam uma interface em comum chamada BoundedElement, que define os métodos que devem ser implementados nas filhas. Os principais métodos existentes na interface BoundedElement são relacionados às transformações geométricas que o elemento pode sofrer. Assim, para cada transformação geométrica aplicada a um objeto, seu envoltório deve ser atualizado com a mesma transformação geométrica.

Em particular, a idéia básica de implementação da interface BoundedElement é sobrescrever os métodos de transformações geométricas herdados de Transformable. Assim, quando um objeto do tipo BoundedMesh sofrer, por exemplo, uma translação, ele irá executar o método translate implementado em BoundedMesh, que sobrescreve o mesmo método herdado de Transformable. Isto é necessário para que as transformações geométricas sofridas pela malha sejam repassadas ao seu volume envoltório. Para tal, o método implementado em BoundedMesh primeiro executa a translação convencional e, em seguida, atualiza o volume envoltório, como detalhado nas linhas 1 e 2 do Algoritmo 15, respectivamente.

Os volumes envoltórios disponíveis nas classes BoundedMesh e BoundedGroup correspondem a um atributo das mesmas e seu tipo é Bounds. A interface Bounds corresponde a uma interface genérica utilizada para representar um volume envoltório e pode possuir algumas

Algoritmo 15 Translação de um elemento BoundedMesh: aplica a translação geométrica na malha do objeto e depois no seu volume envoltório

Entrada: Valores de x, y e z para translação
void translate(x, y, z)
1: $super.translate(x, y, z)$ // executa a translação na malha
2: $bounding.translate(x, y, z)$; // executa a translação no volume envoltório

implementações diferentes como, por exemplo, uma caixa envoltória ou uma esfera envoltória. Atualmente, a implementação disponível neste trabalho é a da caixa envoltória, correspondendo à classe criada chamada BoundingBox, cujas características já foram descritas anteriormente no Capítulo 3.

5.2.4 Transmissão de Dados

Para permitir a realização de testes no sistema, neste trabalho foi implementada uma versão inicial da camada de comunicação que permite a troca de informações entre os dispositivos e o servidor remoto utilizando um protocolo de comunicação específico, o Bluetooth [51]. Devido ao fato do seu alcance ser pequeno (no máximo 100 metros [52]), os dispositivos móveis devem estar próximos ao servidor remoto. Entrentando, como foi explicado na Seção 5.1, a **Camada de Comunicação** é flexível o suficiente para permitir a inclusão e o uso de novas tecnologias de rede.

Cada cenário 3D presente no mundo virtual pode ser solicitado separadamente ao servidor. A idéia é que estes sejam obtidos sob demanda, à medida que a aplicação necessite dos mesmos. Uma vez que um determinado cenário é obtido (no formato de um arquivo m3g), este fica armazenado na memória física do celular, para posteriormente ser carregado na memória da aplicação. Além dos cenários 3D, o servidor remoto envia também trajetórias criadas pelo processo de navegação automática. Estas trajetórias são recebidas pelo dispositivo no formato de vetores de posições, indicando os caminhos por onde o *avatar* deve seguir.

O sistema prevê ainda o uso de colaboração (como ilustram as **Visões Remotas** da arquitetura apresentada na Seção 5.1), possibilitando que futuramente os dispositivos possam comunicar-se entre si para solicitar informações a celulares vizinhos, que estejam conectados em uma mesma rede [46].

No próximo capítulo serão detalhadas as aplicações desenvolvidas utilizando-se o ambiente gráfico implementado.

Capítulo 6

Aplicações Desenvolvidas

Neste capítulo são apresentadas as aplicações desenvolvidas com o intuito de analisar o desempenho do ambiente gráfico implementado [46, 49, 50]. São detalhados três exemplos de navegação interativa por ambientes virtuais: um museu virtual, uma aplicação de resgate e um *shopping* virtual. As aplicações foram testadas no emulador que acompanha o WTK da Sun Microsystems [34] e em dois modelos de telefones celulares Sony Ericsson, w600i e w300i. Estes telefones celulares possuem, respectivamente, 1 MB e 1,5 MB de *dynamic heap* disponíveis para executar a aplicação e, precisamente, 256 MB e 20 MB de *storage heap* disponíveis para armazenamento de arquivos. Os ambientes virtuais foram modelados no 3ds [45] e cada cenário (ou sala) presente nos ambientes foi exportado separadamente para o formato m3g.

As três aplicações implementadas executam com sucesso nos diferentes modelos de celulares e no emulador, evidenciando um bom projeto do ambiente gráfico. Um sistema mais amplo poderia ainda se beneficiar deste sistema, fazendo uso de outros recursos que não são o foco deste trabalho, como o GPS, o que auxiliaria o usuário do dispositivo na sua localização dentro do ambiente.

As hipóteses a serem validadas e as variáveis a serem monitoradas em cada aplicação são definidas abaixo:

1. Museu Virtual

   (a) É possível criar um ambiente virtual em um celular que simule um museu, utilizando

a API M3G.

(b) A navegação se mantém contínua entre salas que não estejam carregadas.

(c) O sistema detecta colisões entre o *avatar* e outros elementos presentes no cenário.

2. Resgate

(a) É possível criar um ambiente virtual que simule uma aplicação de resgate, utilizando a API M3G.

(b) O celular consegue obter cenas e trajetórias a partir do servidor remoto.

(c) É possível criar uma trajetória livre de obstáculos de um ponto até outro.

3. *Shopping* Virtual

(a) É possível criar um ambiente virtual que simule um *shopping center*, utilizando a API M3G.

(b) Os recursos implementados nas aplicações anteriores (1b, , ,) funcionam normalmente nesta aplicação.

(c) Quanto tempo é gasto com as operações de renderização, carregamento de cenas, detecção de colisão e transformações geométricas?

(d) O gerenciamento de memória está sendo efetuado corretamente de forma que diversas cenas possam ser carregadas em um ambiente com memória limitada?

(e) Qual o desempenho dos recursos da aplicação em diferentes modelos de dispositivos e com diferentes variações de objetos geométricos e texturas?

Para as variáveis dos itens 3c, 3d e 3e foram gerados gráficos os quais serão exibidos e analisados no Capítulo 7. As hipóteses foram verificadas utilizando-se análises subjetivas (com um número restrito de usuários, já que não era o foco deste trabalho) e objetivas (totalizando 45 testes, sendo 5 execuções em cada uma das 3 plataformas, para cada um dos 3 cenários).

6.1 Museu Virtual

Nesta seção é descrita uma aplicação que ilustra um museu virtual (que pode ser utilizada em um museu real ou fora dele), no qual usuários de celulares podem navegar interativamente, visualizando a arquitetura, as esculturas e as pinturas exibidas no ambiente 3D sintetizado [46].

Visitantes do museu virtual podem utilizar seus dispositivos móveis para navegar manualmente pelo museu ou ainda para criar uma rota automática até uma obra específica. Usuários podem também ter acesso a informações adicionais através da exploração interativa de objetos contidos no cenário, bem como podem solicitar informações sobre estes objetos, apontando para suas representações no mundo virtual. Adicionalmente, podem mudar o posicionamento e *zoom* de um ponto de vista em particular, por exemplo, para inspecionar uma obra famosa mais de perto.

A arquitetura do sistema implementado suportaria a extensão do uso desta aplicação para incluir a colaboração entre celulares que se encontram em uma vizinhança próxima. Neste caso, por exemplo, professores de posse de um dispositivo móvel mestre poderiam auxiliar estudantes durante uma excursão pelo museu, ou ainda, os estudantes com seus dispositivos móveis poderiam interagir uns com os outros para resolver uma tarefa em grupo (por exemplo, encontrar uma obra de um determinado artista e informar ou enviar informações textuais sobre esta obra para os outros participantes). Além disso, usuários poderiam tirar fotos de certas obras de arte e compartilhá-las com um grupo, colaborativamente, através de seu envio para os outros participantes, via rede.

Nesta aplicação, as células correspondem às salas do museu virtual e os portais correspondem às portas que conectam as salas. A Figura 6.1 mostra dois exemplos de salas do museu virtual.

(a) (b)

Figura 6.1: Salas do museu virtual. Em (a) há 197 vértices e 301 triângulos. Em (b) há 687 vértices e 1265 triângulos. Ambas utilizam texturas com resolução de 72 pixels por polegada.

6.2 Resgate

Embora vivenciar situações de emergência não seja desejável, a motivação para o desenvolvimento desta aplicação de resgate é poder colaborar em situações adversas, utilizando tecnologias de computação. No caso, como ferramenta auxiliar no resgate de uma vítima ou evacuação de um prédio, residência ou ambiente de trabalho [49], por exemplo, através da criação de uma rota de fuga para a vítima em si ou da criação de uma rota para o bombeiro, até a vitima em perigo (essa rota é criada a partir de uma requisição a um servidor remoto, utilizando-se o celular). Assim, foi desenvolvida uma aplicação de resgate, onde usuários podem realizar uma navegação guiada.

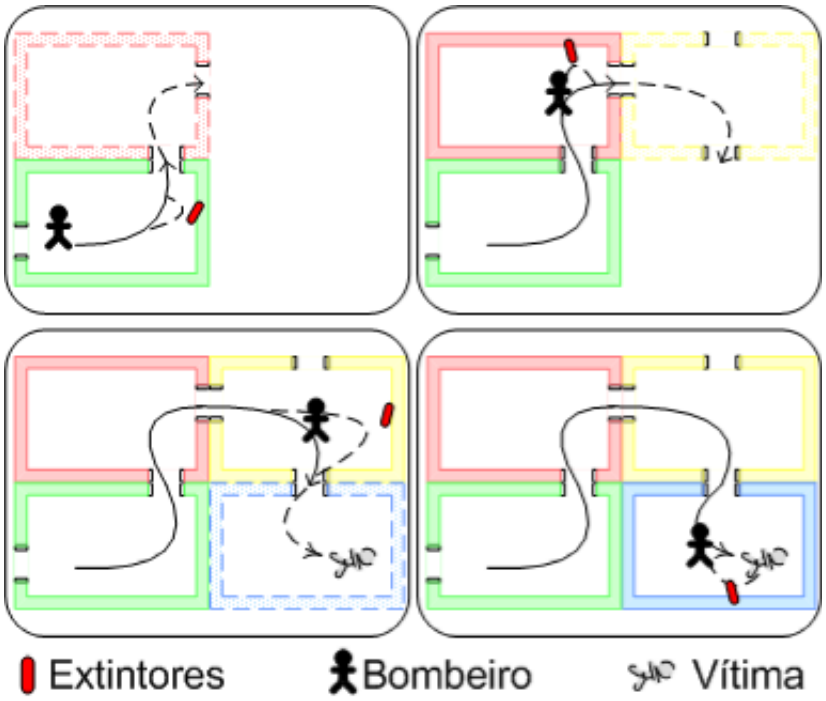


Figura 6.2: Criação do caminho mínimo até uma vítima, identificando diferentes tipos de extintores.

Nesta aplicação, os usuários de celulares (por exemplo, vítimas) podem solicitar uma rota de fuga da construção em perigo. No caso dos usuários serem bombeiros, eles podem usar os dispositivos móveis como uma maneira adicional para combater o fogo ou resgatar um ocupante em perigo (que esteja preso ou perdido) e levá-lo a uma área segura, seguindo instruções em um espaço 3D, obtidas de um computador servidor remoto durante uma situação de emergência. No caso do espaço em questão estar nas mesmas condições em que foi modelado previamente, esse procedimento é executado com sucesso. Trabalhos futuros poderiam abordar a alteração dos modelos 3D presentes no servidor, para inclusão de pontos de obstrução nas rotas, por parte dos bombeiros ou mesmo por parte das vítimas. Desta forma, esta navegação guiada poderia auxiliar em casos nos quais há pessoas perdidas ou confinadas em ambientes fechados, devido

à presença de obstáculos, enquanto simultaneamente permite que tenham algum controle direto e imediato sobre como alternativamente transitar pelo ambiente virtual.

(a) (b)

(c) (d)

Figura 6.3: A aplicação de resgate.

Por exemplo, os bombeiros podem controlar a posição e a orientação de uma câmera virtual, através da qual eles vêem o mundo 3D. Adicionalmente, os bombeiros podem descobrir automaticamente uma rota de saída para um andar específico de um edifício (sinais de saída devem ser naturalmente providos dentro da construção virtual, indicando os caminhos da porta de saída), ou mesmo uma rota de fuga desobstruída, e ainda encontrar a localização de extintores. Além disso, os bombeiros podem encontrar o extintor mais próximo durante a navegação no ambiente e determinar se ele é do tipo apropriado, através da leitura do código de classe do extintor e da comparação desta informação com o tipo de fogo. Os bombeiros podem também encontrar um caminho mínimo do local onde se encontram (a entrada de um edifício) até outro

local (a sala onde uma vítima está presa), como mostrado na Figura 6.2, bem como ter acesso a informações adicionais através da visualização de cenas virtuais que podem ser exploradas interativamente. Podem ainda solicitar informações sobre objetos de interesse (por exemplo, extintores) apontando para a sua localização no ambiente virtual.

A Figura 6.3 ilustra a aplicação de resgate implementada, exibindo quatro salas de uma construção. Em particular, a Figura 6.3a exibe a posição inicial do bombeiro, enquanto que a Figura 6.3d mostra o local aonde a vítima se encontra.

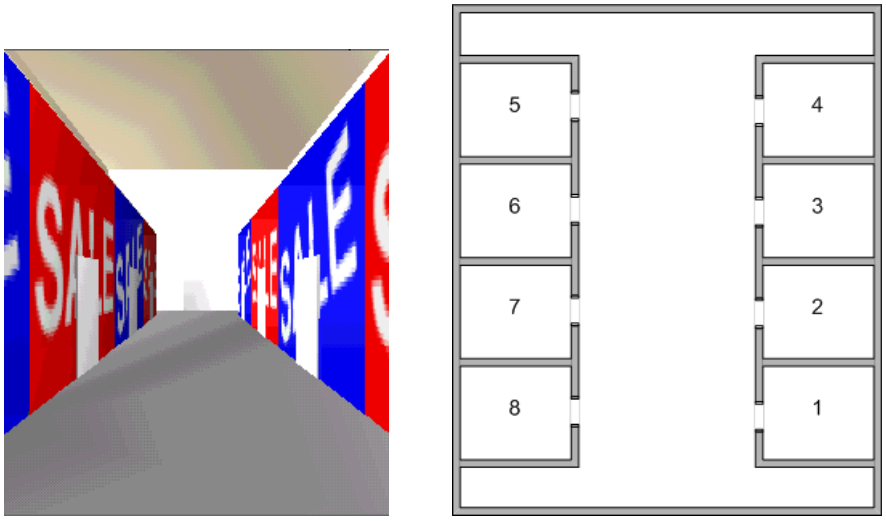
6.3 *Shopping* Virtual

Nesta seção é descrita uma aplicação que consiste em um *shopping center* virtual. Usando esta aplicação, o usuário do dispositivo pode, entre outras coisas, navegar livremente pelo ambiente do *shopping*, visualizar itens relativos a uma loja ou a um produto específico (através da seleção de sua representação 3D no ambiente virtual) e ainda obter um caminho mínimo no *shopping* até uma determinada loja. A motivação é que usuários de dispositivos móveis possam mover-se através do espaço e, interativamente, visualizar, navegar e planejar rotas através de um passeio virtual no ambiente.

O ambiente virtual é mostrado na Figura 6.4. Em particular, na Figura 6.4 (a), é exibida uma visão do corredor principal do *shopping* virtual, com lojas em ambos os lados. Essas lojas foram numeradas, iniciando na mais próxima à direita (mostrada na imagem) e terminando na mais próxima à esquerda. Esse esquema é ilustrado na Figura 6.4 (b), que mostra uma visão superior da imagem visualizada na Figura 6.4 (a).

Na Figura 6.5 são exibidas as salas do *shopping* virtual. As imagens exibidas correspondem à mesma seqüência mostrada na Figura 6.4 (b).

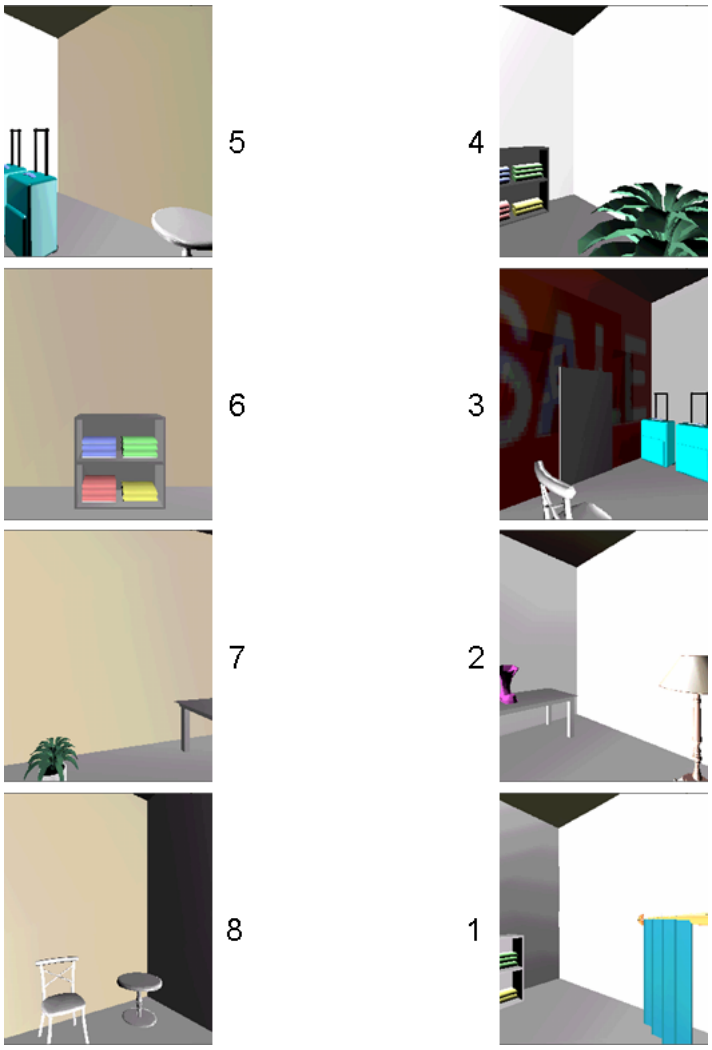
Nesta aplicação, os visitantes do *shopping center* podem usar seus celulares para navegar pela representação virtual do mesmo, procurando lojas ou produtos que sejam de interesse. De forma automática, através do seu telefone celular um usuário também pode descobrir o menor caminho (livre de obstáculos) até determinado local (por exemplo, uma loja ou uma saída do *shopping*, agilizando a chegada ao seu carro ou a um ponto de ônibus), como representado na Figura 6.6. O usuário pode ainda definir as características de um determinado item pelo qual procura, possibilitando que o sistema utilize o planejamento de trajetórias para gerar um



(a) (b)

Figura 6.4: A aplicação do *shopping center* virtual. Em (a) uma visão do corredor do *shopping* e, em (b), uma visão superior da estrutura do mesmo.

caminho até a loja onde este produto está disponível.



(a)

Figura 6.5: Salas do *shopping center* virtual.

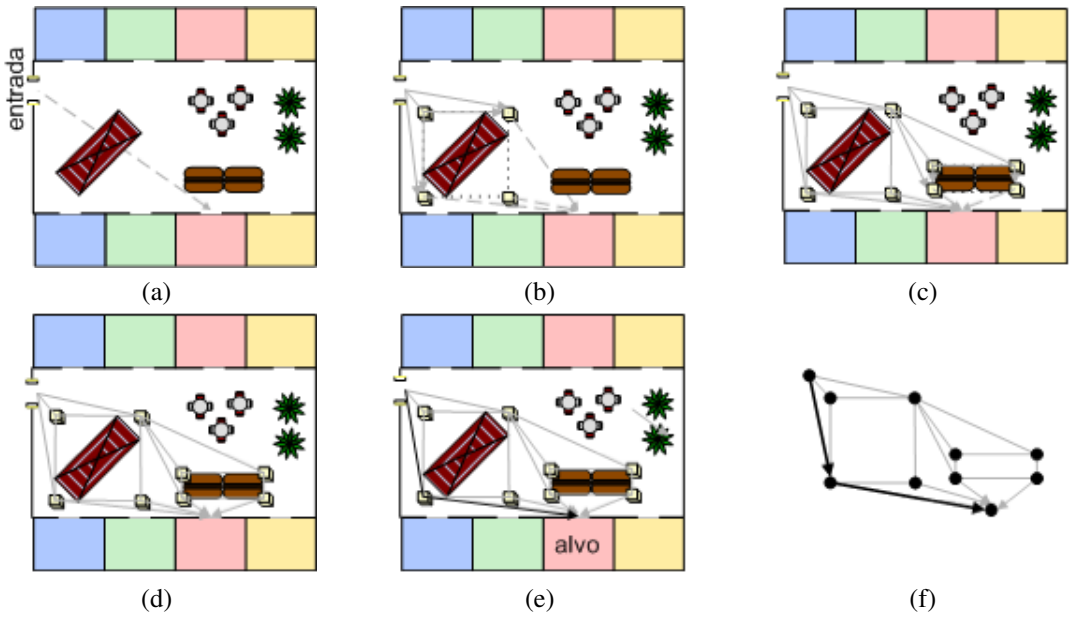


Figura 6.6: Em (a), (b), (c) e (d) são exibidos os caminhos livres de obstáculos até uma determinada loja. Em (f), é mostrado o grafo que representa o menor caminho, livre de obstáculos, da origem em que se encontra o usuário até um determinado ponto de interesse.

Capítulo 7

Análise de Desempenho

O *shopping* virtual apresentado na Seção 6.3 foi utilizado para analisar o desempenho do sistema. Os testes foram efetuados em três plataformas diferentes: em um PC tradicional, executando no emulador disponível no WTK da Sun Microsystems, e em dois telefones celulares da Sony Ericsson de modelos w300i e w600i (modelos disponíveis para teste). A configuração dos dispositivos encontra-se na Tabela 7.1.

Tabela 7.1: Configurações dos dispositivos utilizados nos testes.

	Dynamic heap	*Storage heap* (memória interna)	Processador
PC (Emulador)	800 KB	não foi limitado	AMD Turion 64 X2 - 768 MB de RAM
w600i	1000 KB	256 MB	Não disponibilizado
w300i	1500 KB	20 MB	Não disponibilizado

O valor do *dynamic heap* definido para o emulador foi limitado a 800 KB, como forma de tentar aproximar a sua configuração de memória com a dos celulares. O *storage heap* não foi limitado por não constituir um problema na implementação.

Para todos os cenários de execução da aplicação definidos, foram carregadas a estrutura do *shopping* e as respectivas salas por onde o observador navega. Cada sala é composta de objetos 3D e a estrutura do *shopping* possui texturas aplicadas à mesma. Os cenários de execução foram definidos da seguinte forma:

- **Cenário 1.** Os objetos 3D presentes nas salas não foram carregados e as texturas não foram utilizadas;
- **Cenário 2.** Os objetos 3D presentes nas salas foram carregados e as texturas não foram utilizadas;
- **Cenário 3.** Os objetos 3D presentes nas salas foram carregados e as texturas foram utilizadas.

O objetivo da definição destes cenários foi analisar o impacto que a adição de objetos e texturas causam na execução da aplicação, tanto no desempenho, quanto na memória disponível, contemplando as variáveis definidas no item 3e do Capítulo 6.

Uma trajetória que passa por seis, das oito salas do *shopping* virtual, foi previamente gravada (contendo 249 pontos). A Figura 7.1 ilustra a trajetória utilizada nos testes efetuados. O observador caminha pelas salas 1, 2, 7, 6, 4 e 5, nesta ordem. Essa trajetória foi escolhida aleatoriamente, de forma a percorrer o máximo de salas que os dispositivos fossem capazes de carregar. O tempo de execução e a memória disponível foram armazenadas para cada passo da execução. Foram realizadas cinco execuções consecutivas desta trajetória automática e a média destas execuções foi calculada e utilizada nos testes para a geração dos gráficos. O tempo total de cada execução foi separado nos tempos gastos com os principais processos da aplicação: renderização, aplicação de transformações geométricas de translação e rotação, cálculo da detecção de colisões e carregamento de cenas. As demais operações foram agrupadas e identificadas como "Outras operações". Os valores para o tempo de processamento foram obtidos em milisegundos. A detecção de colisão inclui testes com as caixas envoltórias, as quais englobam as portas e os objetos presentes nas salas. Em situações de detecção de colisão, o tempo gasto para aplicar a transformação geométrica, que retorna o objeto para a posição anterior ao momento da colisão, não foi utilizado para a geração dos gráficos. Todas essas definições estão relacionadas ao monitoramento das variáveis definidas nos itens 3c e 3d do Capítulo 6.

As hipóteses definidas nos itens 1 (a, b e c), 2 (a, b e c) e 3 (a e b) foram verificadas com sucesso. A navegação se manteve contínua, mesmo na presença de colisões entre o *avatar* e os elementos do cenário, embora, em algumas situações, tenham ocorrido pequenas pausas, causadas pelo carregamento das salas. A comunicação com o servidor também foi implementada com sucesso, sendo criada (no servidor) uma trajetória entre dois pontos e, posteriormente, enviada ao dispositivo.

Quanto às variáveis monitoradas no item 3c, 3d e 3e do Capítulo 6, as médias dos resul-

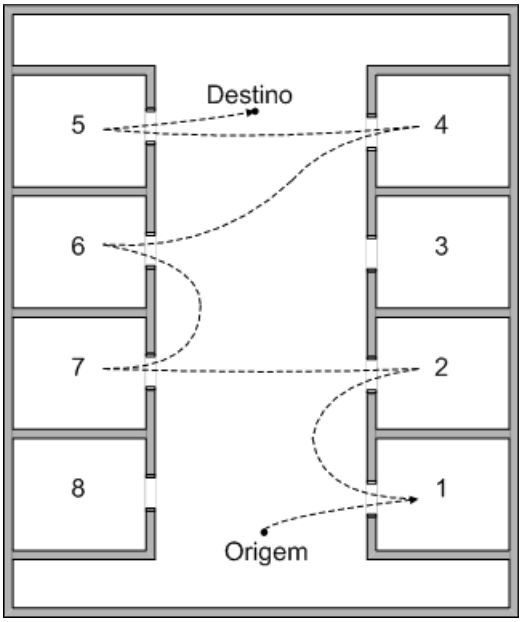


Figura 7.1: Trajetória percorrida automaticamente pelo observador.

tados obtidos estão especificadas nas Figuras 7.2, 7.3, 7.4 e 7.5. Como pode ser observado para as três plataformas, a inclusão de objetos geométricos às cenas causa um acréscimo nos tempos individuais dos processos de renderização, transformações geométricas e carregamento de cenas, resultando em um tempo total de execução maior. Como exemplo, para o w600i (Figura 7.2), o tempo total para execução da aplicação, sem o carregamento de objetos, foi de 43,98s, enquanto que com o acréscimo dos objetos o tempo total passou para 83,63s. Com a inclusão de texturas, o tempo total foi 86,47s. Nas Figuras 7.2, 7.3 e 7.5, pode-se observar que o processo de renderização consome muito tempo de processamento. Entretanto, nos celulares, o processo de transformação geométrica também consome muito tempo de processamento (os valores percentuais são sempre maiores que 36% do valor total do tempo de execução). Apesar de ser esperado que a renderização ocupasse grande parte deste tempo, as transformações geométricas, entretanto, poderiam consumir um tempo menor. Esse valor alto obtido para as transformações geométricas deve-se ao fato de que, além da aplicação da transformação ser efetuada, é ainda necessária a atualização das caixas envoltórias, o que envolve cálculos que depreciam o desempenho do processo, como a procura pelos novos valores extremos da caixa envoltória, onde todos os vértices do objeto que sofreu a transformação geométrica são percorridos. Métodos mais eficientes poderiam ser implementados para efetuar esta atualização. As Figuras 7.2, 7.3 e 7.5 também mostram que o tempo gasto com o cálculo da detecção de colisão é mínimo (por exemplo, sempre abaixo de 1% do valor total do tempo de execução nos celulares). Por sua vez, o processo de carregamento de cenas ocupa um tempo maior para os testes onde há objetos a serem carregados nas salas. Por exemplo, no w600i (Figura 7.2), o percentual de tempo gasto para carregar cenas vazias e cenas preenchidas com objetos foi, respectivamente, de 1,59% e de 9,01%. Devido ao aumento do tempo gasto com o carregamento

de cenas e com a renderização (na mudança do cenário 1 para o 2), o tempo gasto para efetuar outras operações diminui.

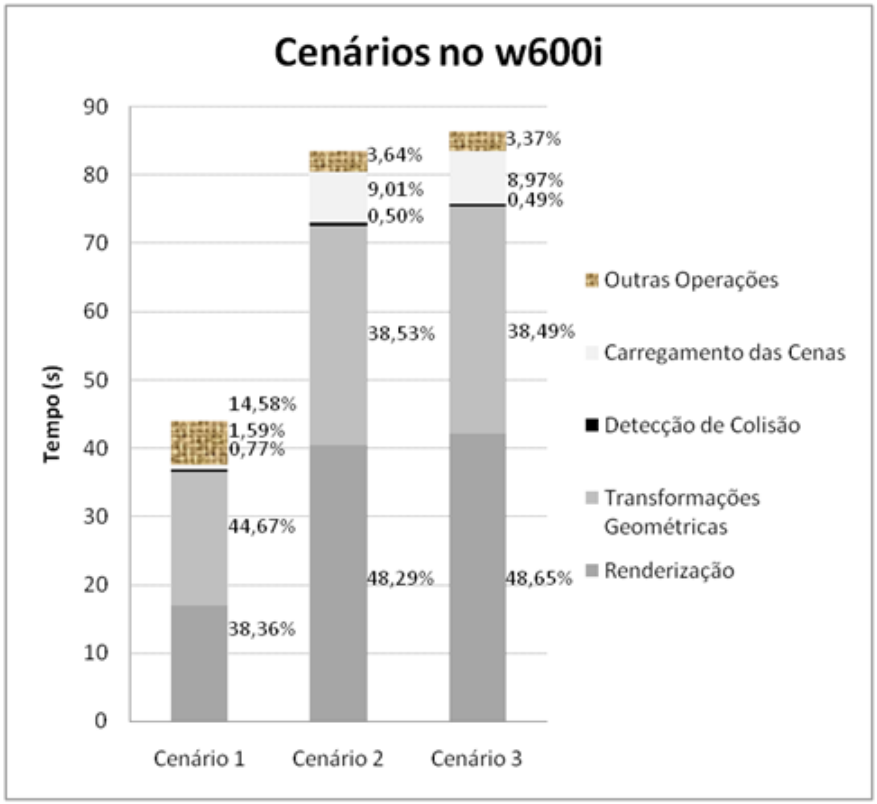


Figura 7.2: Médias de tempo utilizadas em cada processo, nos três cenários, para o celular w600i.

Para a geração dos próximos gráficos foram calculadas a média e a mediana de todos os valores obtidos nas mesmas cinco execuções. Por retratar com maior fidelidade e menor ruído a amostra de dados dos 5 experimentos, a mediana foi utilizada. Com base na mediana, os valores de desvio-padrão foram calculados e são também exibidos nos gráficos.

Nas Figuras 7.6, 7.7 e 7.8, os valores da mediana, relativos à memória disponível, são exibidos, respectivamente, para os Cenários 1, 2 e 3. Para cada figura, são ilustrados os resultados no celular w600i, no w300i e no emulador. Os valores de desvio padrão para a Figura 7.6 são 162,2245; 161,8267 e 162,4157KB para o w600i, w300i e emulador, respectivamente. Os valores de desvio padrão para a Figura 7.7 são 116,5996; 126,9870 e 147,8928KB para o w600i, w300i e emulador, respectivamente. Os valores de desvio padrão para a Figura 7.8 são 116,6416; 125,8097 e 147,9139KB para o w600i, w300i e emulador, respectivamente. O Cenário 1 precisa ser analisado separadamente dos outros dois pois a falta de objetos na cena com os quais colidir, faz com que o observador se desloque por um caminho um pouco diferente, apesar de seguir as mesmas transformações geométricas que foram aplicadas nos Cenários

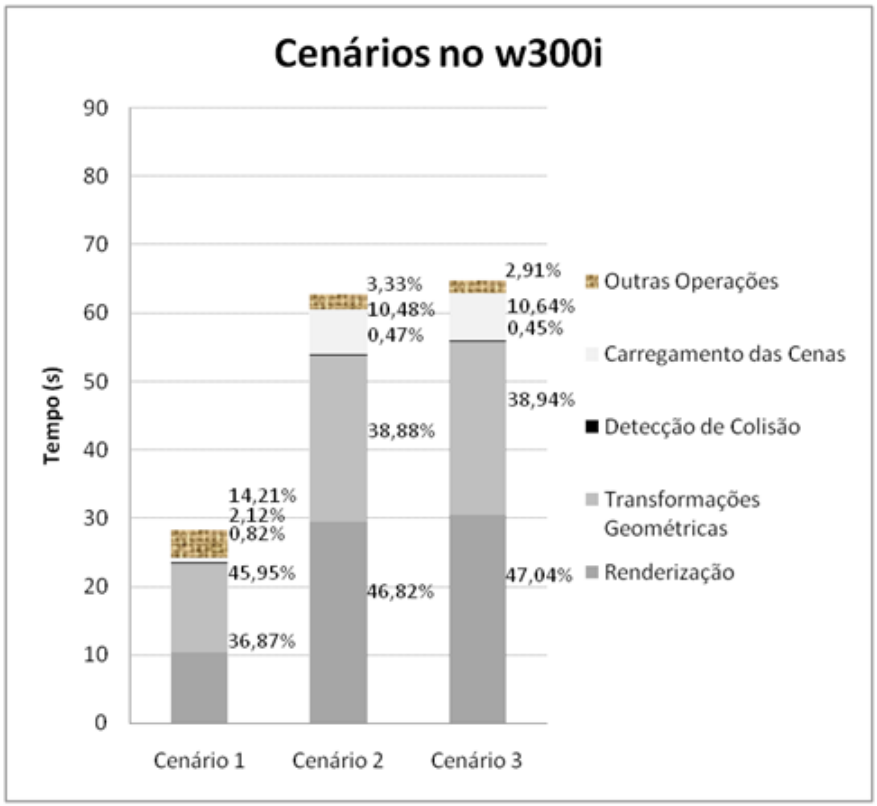


Figura 7.3: Médias de tempo utilizadas em cada processo, nos três cenários, para o celular w300i.

2 e 3. Na trajetória seguida no Cenário 1, o observador entra em apenas três salas, evidenciando a chamada explícita que é feita ao coletor de lixo do Java após o carregamento das mesmas. O coletor libera a memória não mais utilizada e aumenta a quantidade de memória disponível. Isso pode ser observado, para os celulares (em (a) e (b) da Figura 7.6), nos três primeiros saltos que ocorrem com os pontos da Figura 7.6, em particular, nos pontos 31, 187 e 300 do eixos das abscissas. O quarto salto ocorre, não pela entrada em uma sala, mas sim, pelo esgotamento da memória, fazendo com que a própria máquina virtual do Java chame o coletor de lixo. Para os Cenários 2 e 3, ambos os modelos de celulares se comportaram de forma muito similar, apresentando aumentos na memória disponível, exatamente após os pontos da trajetória 31, 167, 242, 323, 406 e 485 das Figuras 7.7 e 7.8, onde as salas são carregadas. Novamente, o coletor de lixo foi chamado nestes pontos e liberou a memória não mais utilizada. Apesar do celular w300i apresentar uma maior quantidade de memória para execução (coluna *dynamic heap* da Tabela 7.1), a similaridade entre os gráficos apresentados nas Figuras 7.7 e 7.8 deve-se ao fato da memória deste dispositivo ser liberada sob demanda, utilizando apenas a quantidade necessária para execução da aplicação. Por sua vez, o emulador apresenta gráficos diferenciados dos gráficos dos celulares, possivelmente, devido ao gerenciamento de memória do emulador (chamada ao coletor de lixo) ser efetuado de uma forma diferente, requerendo espaço em memória em

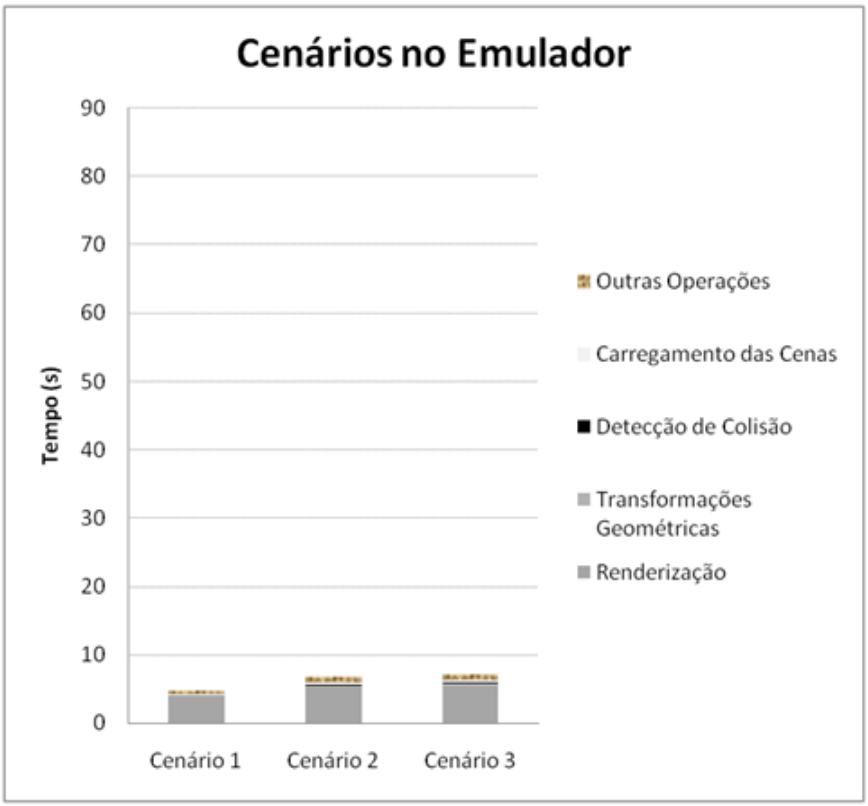


Figura 7.4: Médias de tempo utilizadas em cada processo, nos três cenários, para o emulador.

pontos da trajetória onde não existe uma chamada explícita ao coletor de lixo. Em particular, pode-se perceber que são efetuadas mais chamadas ao coletor, do que as presentes nos gráficos dos celulares.

A mediana do tempo de renderização gasto em cada ponto da trajetória é exibida nas Figuras 7.9, 7.10 e 7.11. Os valores de desvio padrão para a Figura 7.9 são 0,07363; 0,04596 e 0,0077s para o w600i, w300i e emulador, respectivamente. Os valores de desvio padrão para a Figura 7.10 são 0,1701; 0,1304 e 0,0105s para o w600i, w300i e emulador, respectivamente. Os valores de desvio padrão para a Figura 7.11 são 0,1741; 0,1327 e 0,01064s para o w600i, w300i e emulador, respectivamente. Através destes gráficos, pode-se observar que o emulador tem o melhor desempenho, por apresentar os menores tempos gastos com renderização. Prova esta de que o emulador não é confiável como parâmetro único indicativo para se avaliar o desempenho da aplicação em celulares, pois claramente utiliza as configurações da máquina onde executa. Portanto, torna-se mais interessante e útil comparar apenas os desempenhos dos celulares daqui para frente. Nesse caso, as imagens mostram que o w300i apresentou, em todos os cenários, um desempenho superior ao w600i, uma vez que precisou de um menor tempo para realizar a renderização de cada ponto da trajetória. Em particular, a Figura 7.12 permite a compara-

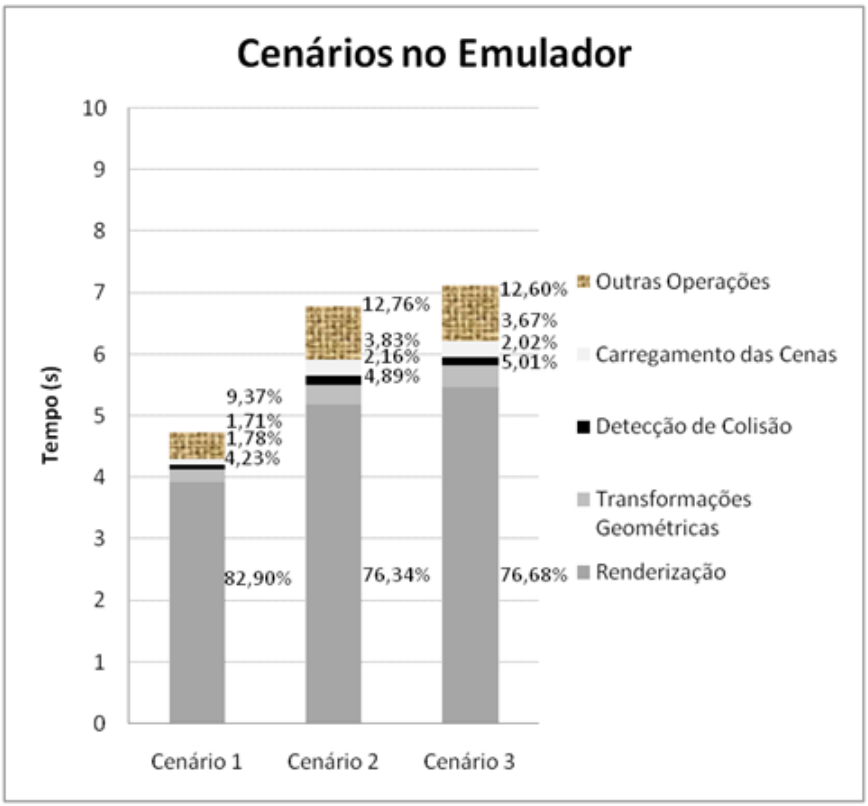


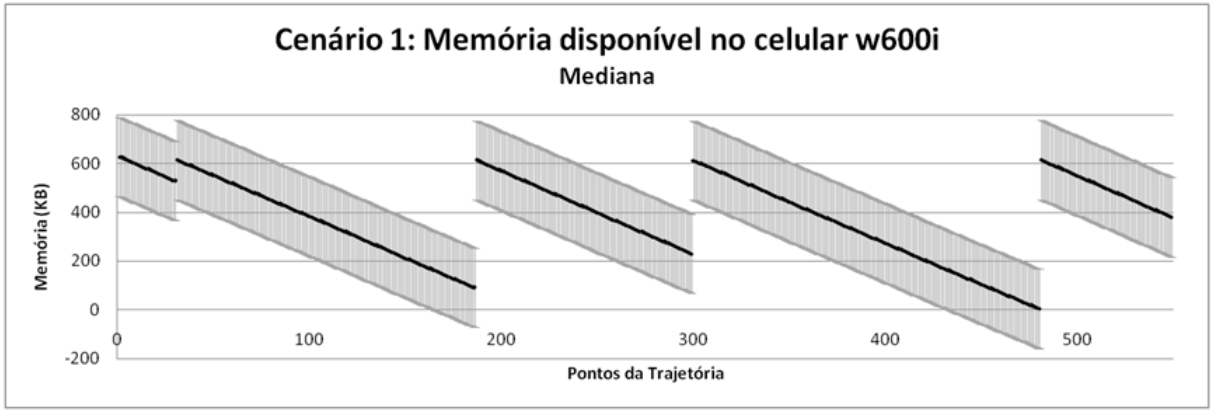
Figura 7.5: Visão maximizada da Figura 7.4.

ção dos tempos que ambos os celulares gastaram com a renderização, evidenciando o melhor desempenho do w300i.[1]

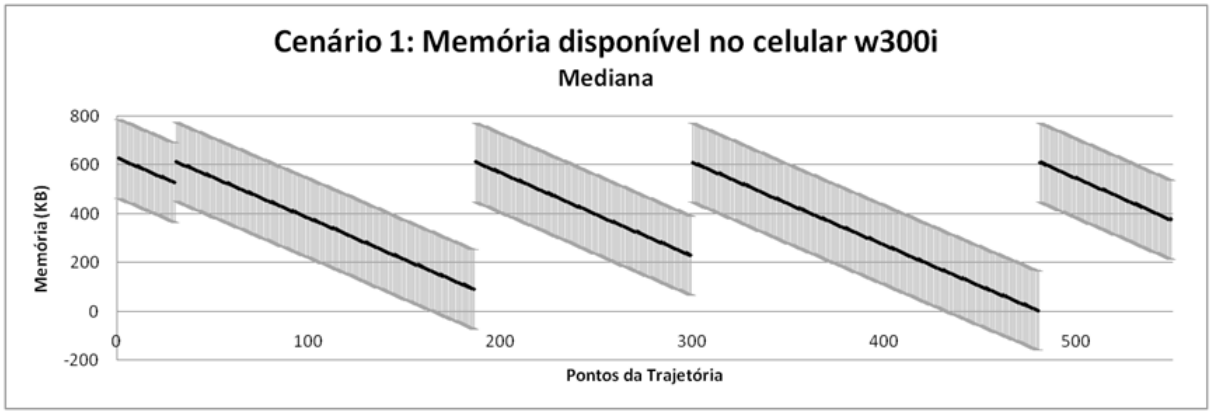
As medianas do tempo gastos com transformações geométricas em cada ponto da trajetória são exibidas nas Figuras 7.13, 7.14 e 7.15. Os valores de desvio padrão para a Figura 7.13 são 0,0882; 0,0583 e 0,0024s para o w600i, w300i e emulador, respectivamente. Os valores de desvio padrão para a Figura 7.14 são 0,1860; 0,1438 e 0,0039s para o w600i, w300i e emulador, respectivamente. Os valores de desvio padrão para a Figura 7.15 são 0,1904; 0,1469 e 0,0042s para o w600i, w300i e emulador, respectivamente. Assim como na renderização, o emulador tem o melhor desempenho, mas não é confiável. O w300i apresentou também, em todos os cenários, um desempenho superior ao w600i, uma vez que precisou de um menor tempo para aplicar transformações geométricas em cada ponto da trajetória. Em particular, a Figura 7.16 permite a comparação dos tempos que ambos os celulares gastaram com as transformações geométricas, evidenciando, novamente, o melhor desempenho do w300i.

[1] O site http://www.jbenchmark.com/index.jsp apresenta dados que reforçam os resultados obtidos, mostrando que o desempenho do w300i é superior ao do w600i.

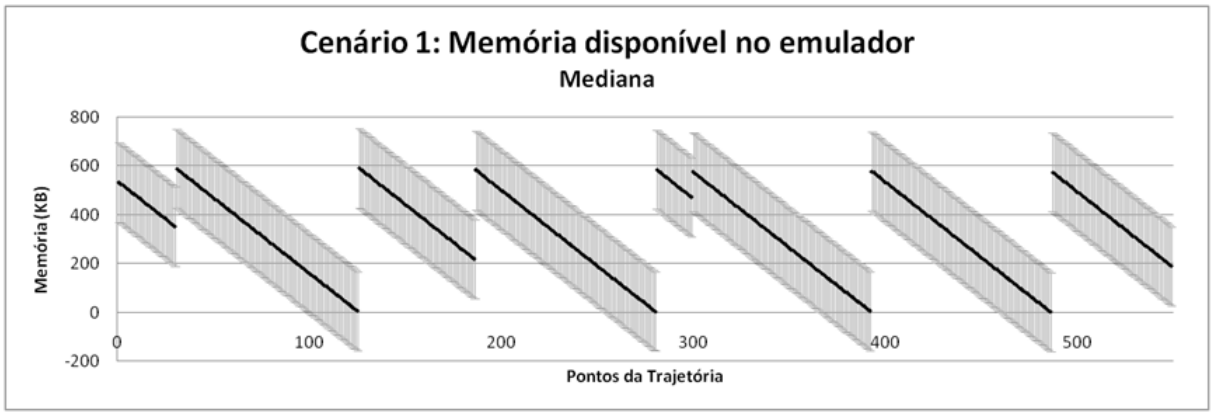
Quanto ao algoritmo de planejamento de trajetórias, atualmente ele está sendo executado no computador servidor e, no emulador, com sucesso. Vale ressaltar que o planejamento de trajetórias já otimiza a detecção de colisão, gerando um caminho livre de obstáculos. Nos cenários virtuais utilizados atualmente e, da forma como foram gerados os gráficos, o algoritmo de planejamento de trajetórias apresentaria resultados similares aos obtidos, uma vez que praticamente não há obstáculos dos quais desviar. Entretanto, em situações críticas, como ambientes muito povoados com obstáculos (por exemplo, o corredor do *shopping*), o percentual obtido para aplicação de transformações geométricas provavelmente seria menor que o atual, caso fosse considerado nos gráficos o tempo para posicionar o objeto colidente na posição anterior à da ocorrência da colisão.



(a)

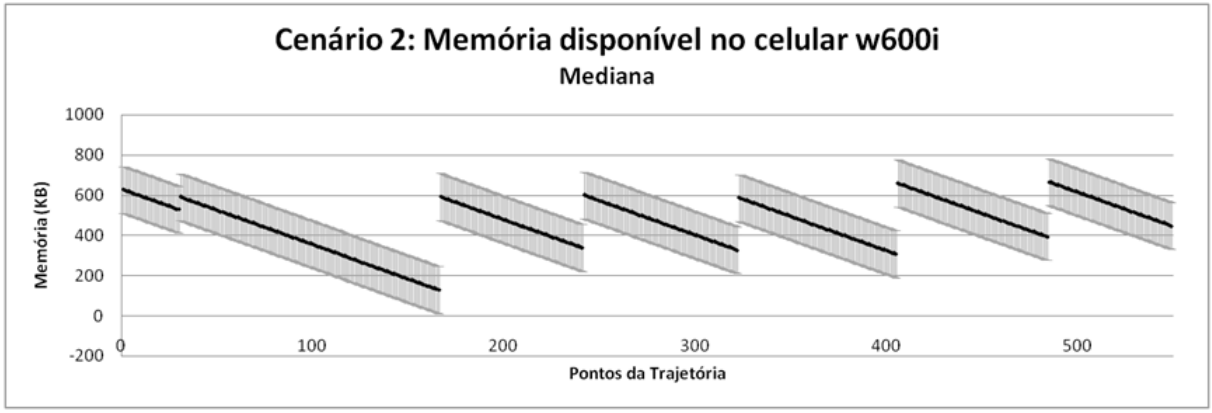


(b)

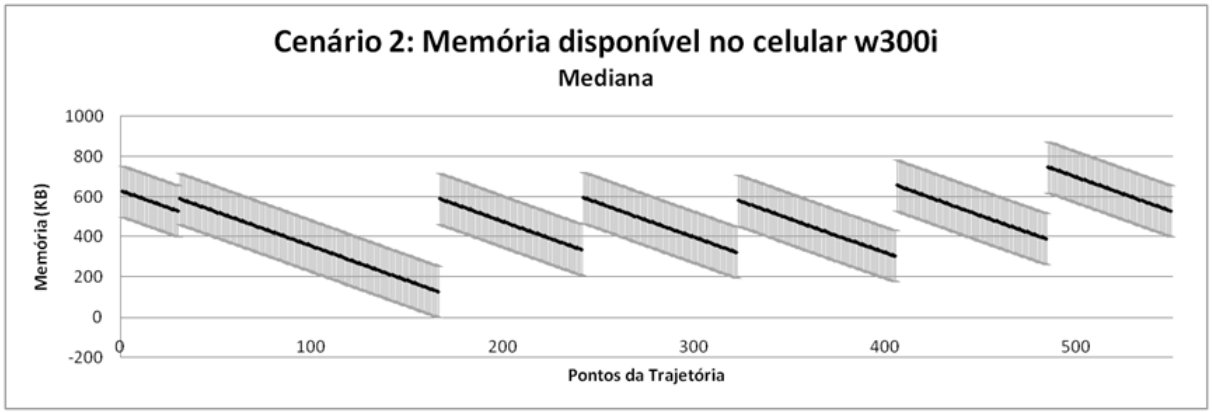


(c)

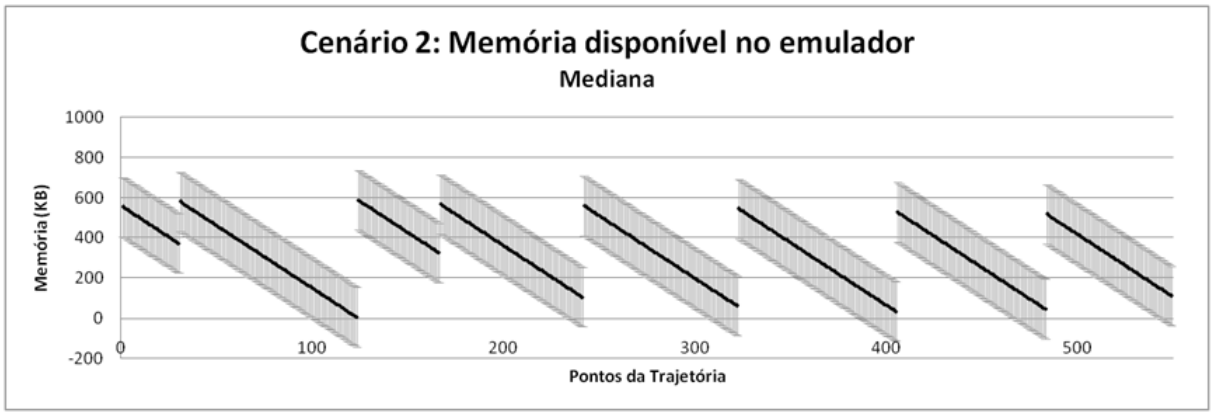
Figura 7.6: Valores de mediana da quantidade de memória disponível para o Cenário 1. Em (a), (b) e (c), o w600i, o w300i e o emulador, respectivamente. Os valores do desvio padrão em (a), (b) e (c) são 162,2245; 161,8267 e 162,4157KB, respectivamente.



(a)

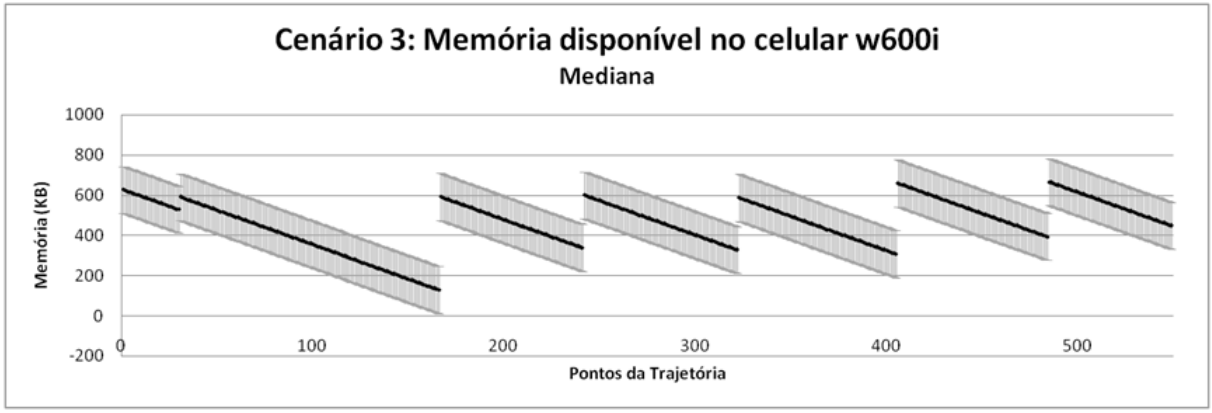


(b)

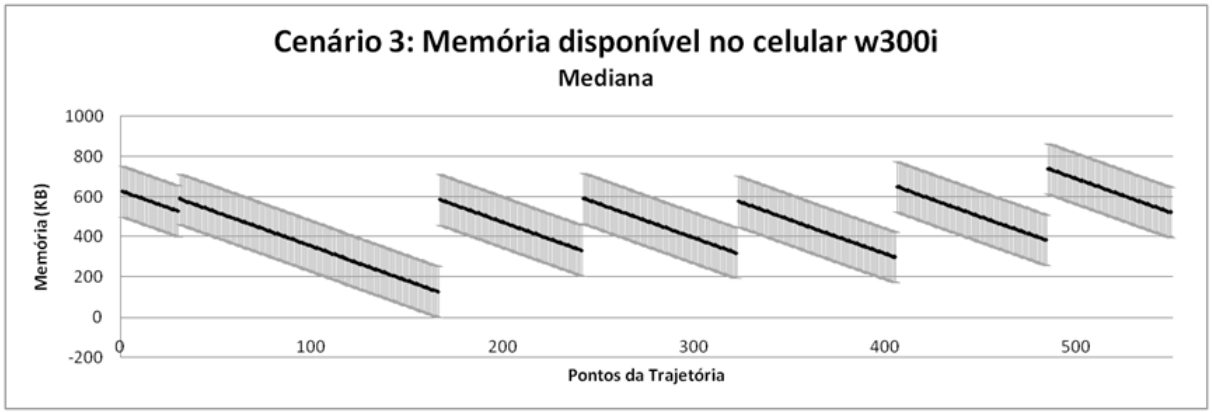


(c)

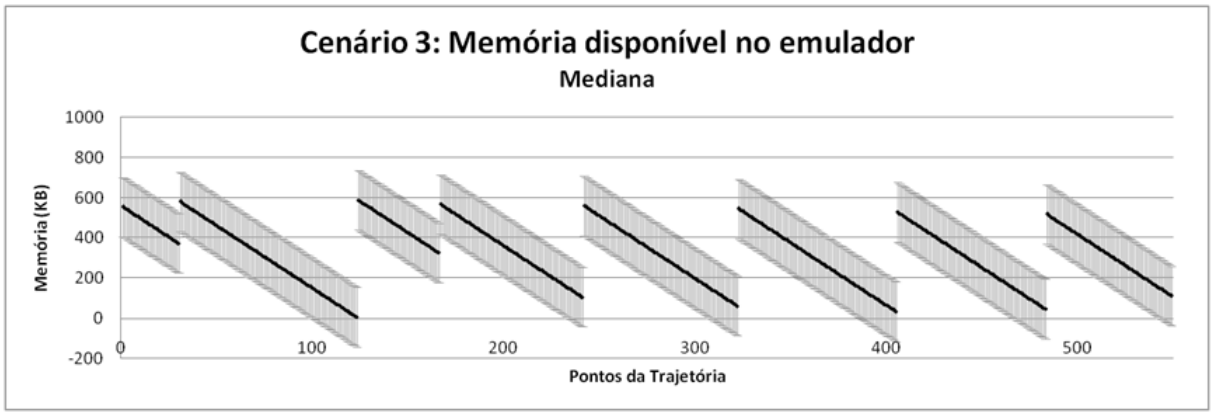
Figura 7.7: Valores de mediana da quantidade de memória disponível para o Cenário 2. Em (a), (b) e (c), o w600i, o w300i e o emulador, respectivamente. Os valores do desvio padrão em (a), (b) e (c) são 116,5996; 126,9870 e 147,8928KB, respectivamente.



(a)

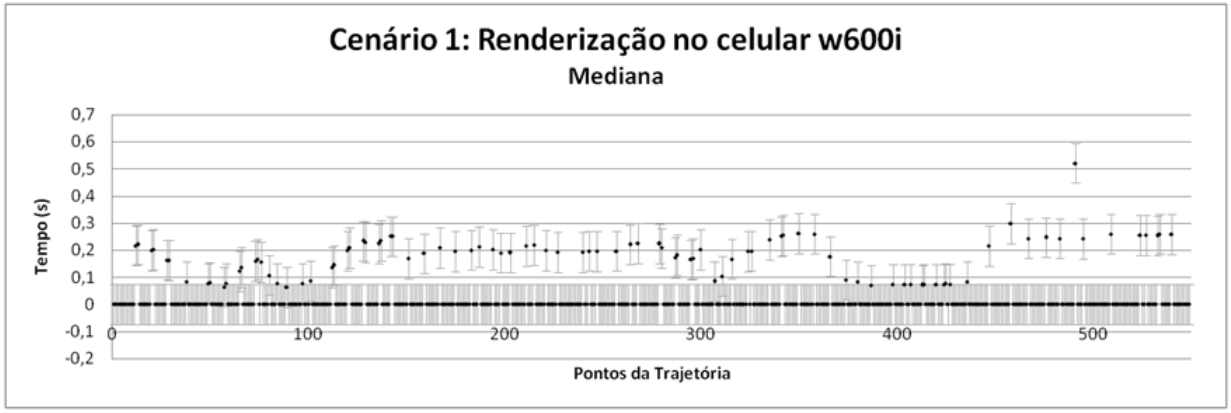


(b)

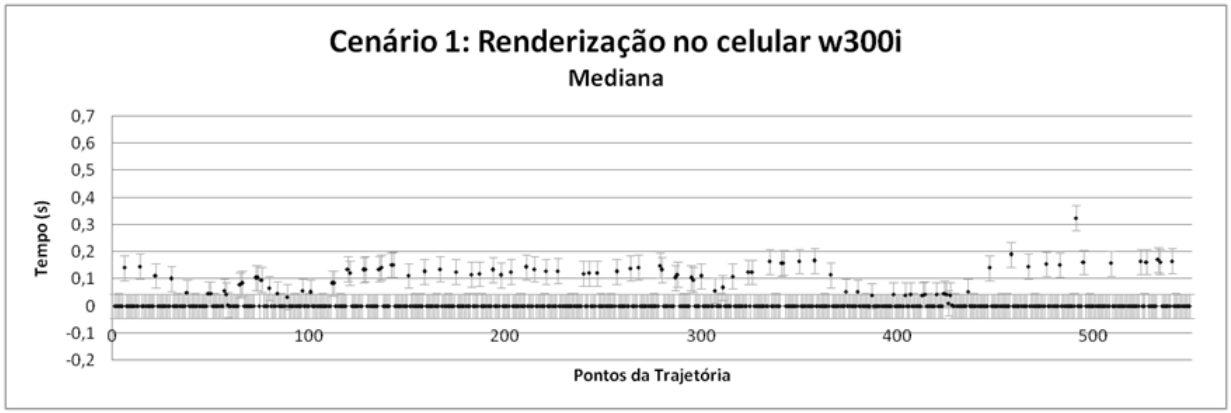


(c)

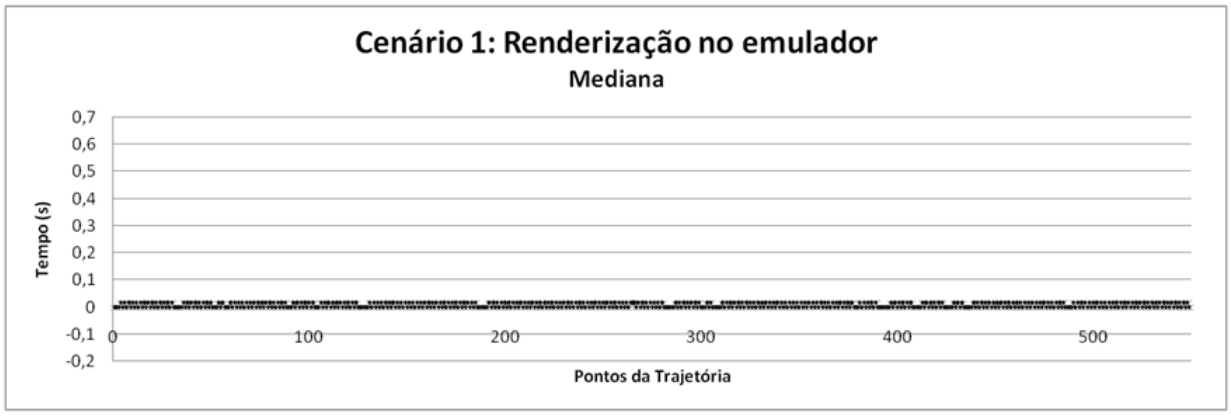
Figura 7.8: Valores de mediana da quantidade de memória disponível para o Cenário 3. Em (a), (b) e (c), o w600i, o w300i e o emulador, respectivamente. Os valores do desvio padrão em (a), (b) e (c) são 116,6416; 125,8097 e 147,9139KB, respectivamente.



(a)

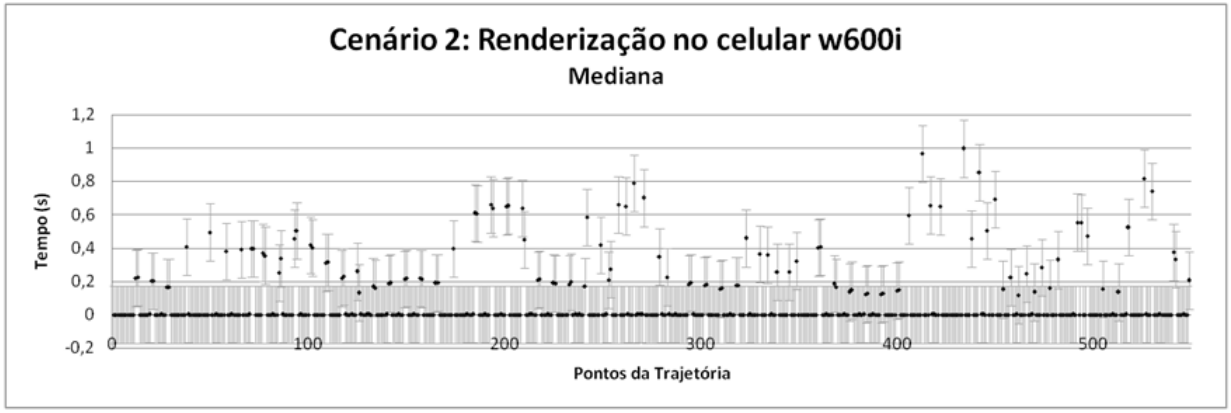


(b)

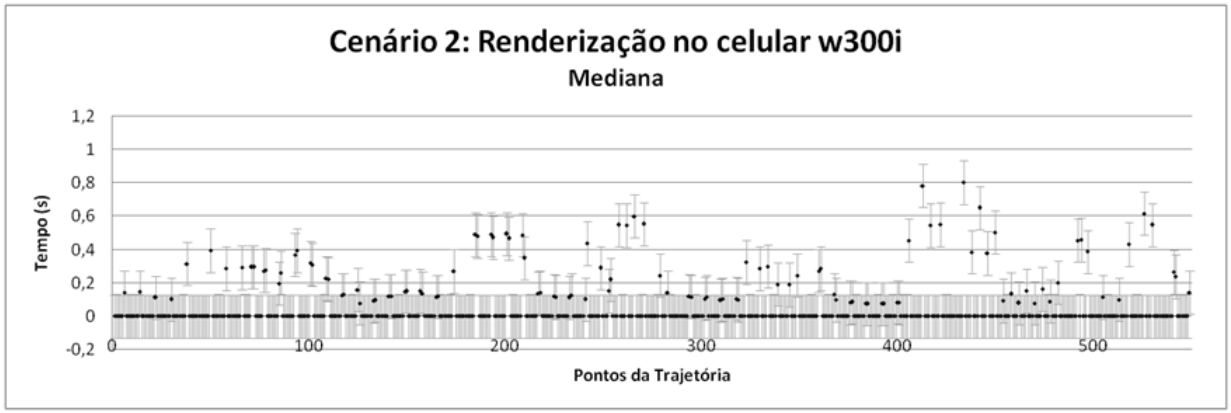


(c)

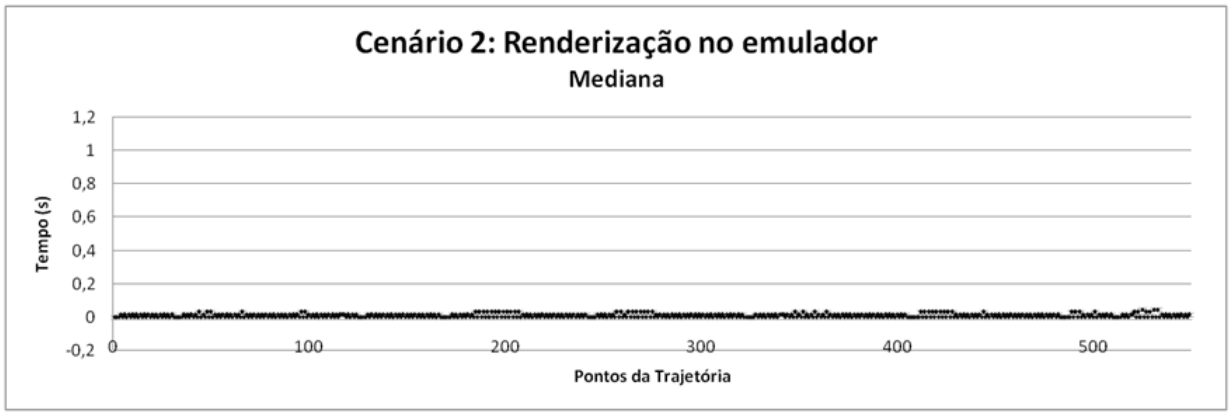
Figura 7.9: Valores de mediana do tempo de renderização para o Cenário 1. Em (a), (b) e (c), o w600i, o w300i e o emulador, respectivamente. Os valores do desvio padrão em (a), (b) e (c) são 0,07363; 0,04596 e 0,0077s, respectivamente.



(a)

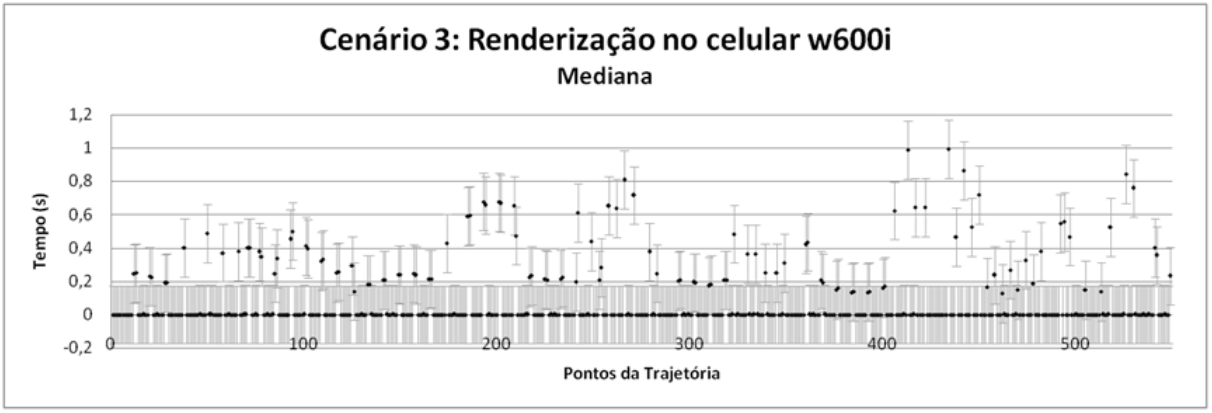


(b)

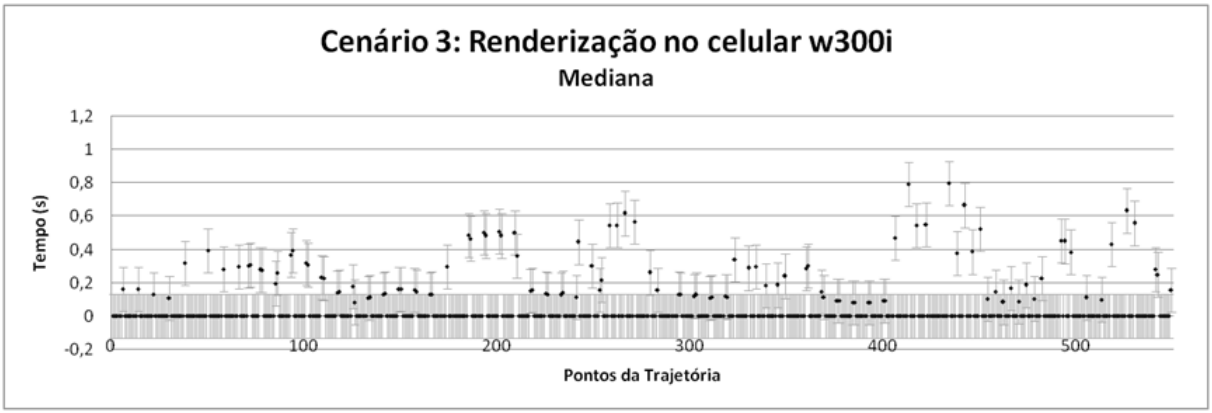


(c)

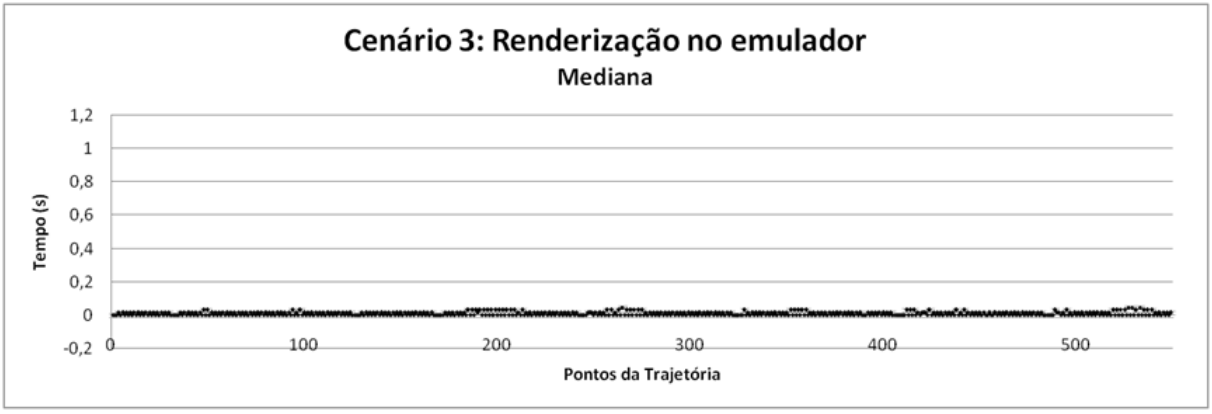
Figura 7.10: Valores de mediana do tempo de renderização para o Cenário 2. Em (a), (b) e (c), o w600i, o w300i e o emulador, respectivamente. Os valores do desvio padrão em (a), (b) e (c) são 0,1701; 0,1304 e 0,0105s, respectivamente.



(a)



(b)



(c)

Figura 7.11: Valores de mediana do tempo de renderização para o Cenário 3. Em (a), (b) e (c), o w600i, o w300i e o emulador, respectivamente. Os valores do desvio padrão em (a), (b) e (c) são 0,1741; 0,1327 e 0,01064s, respectivamente.

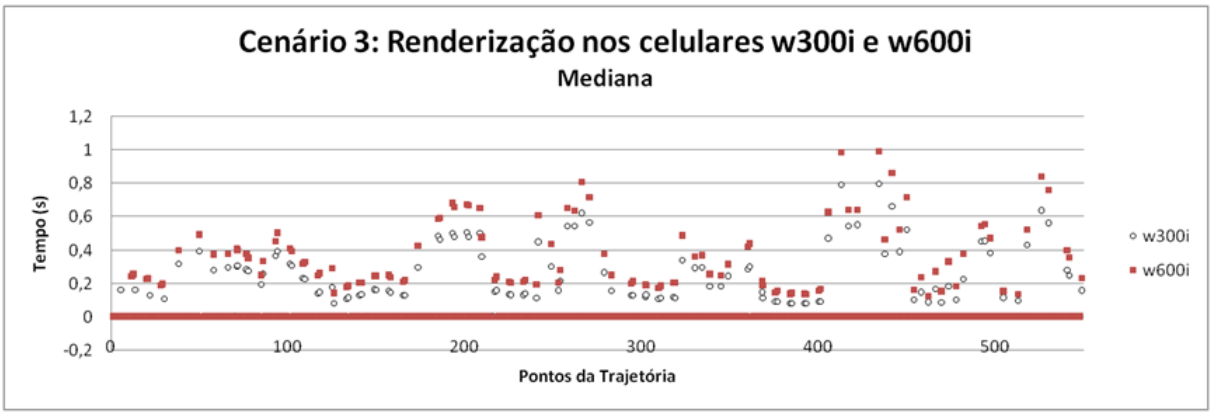
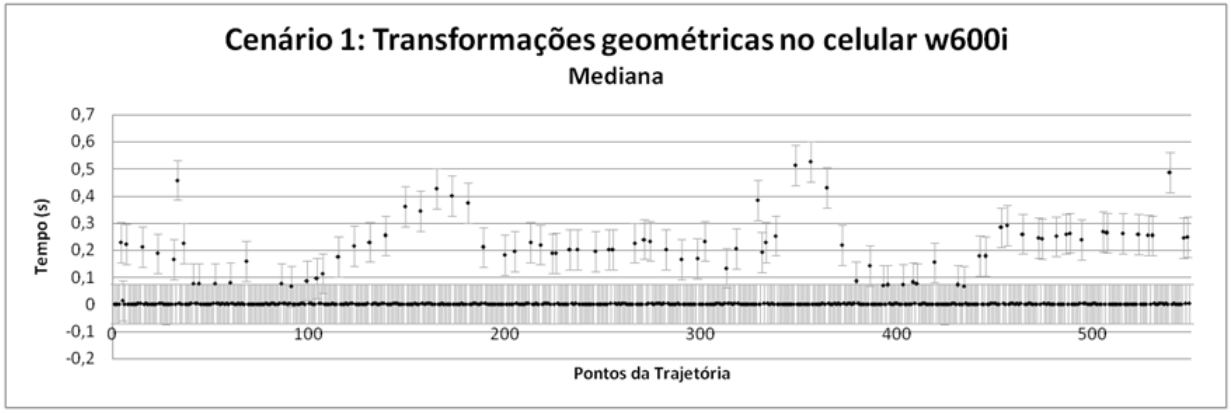
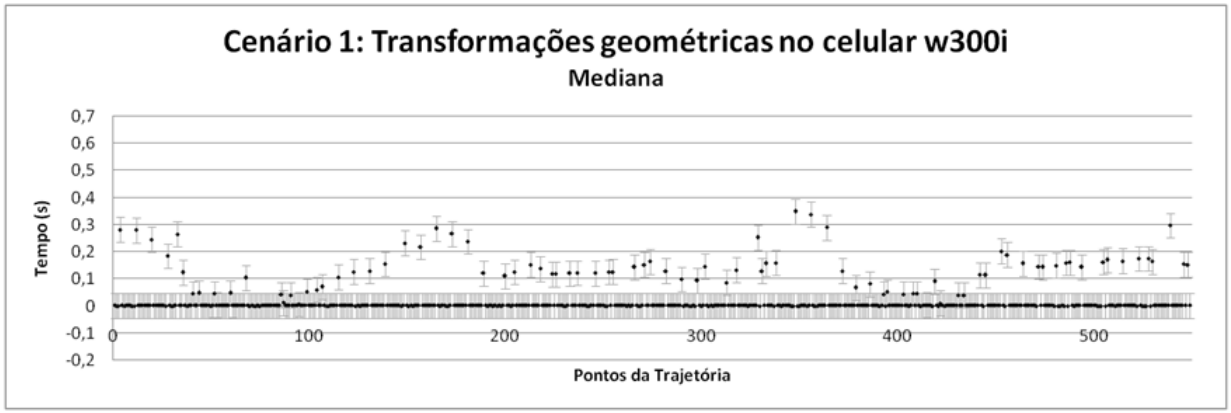


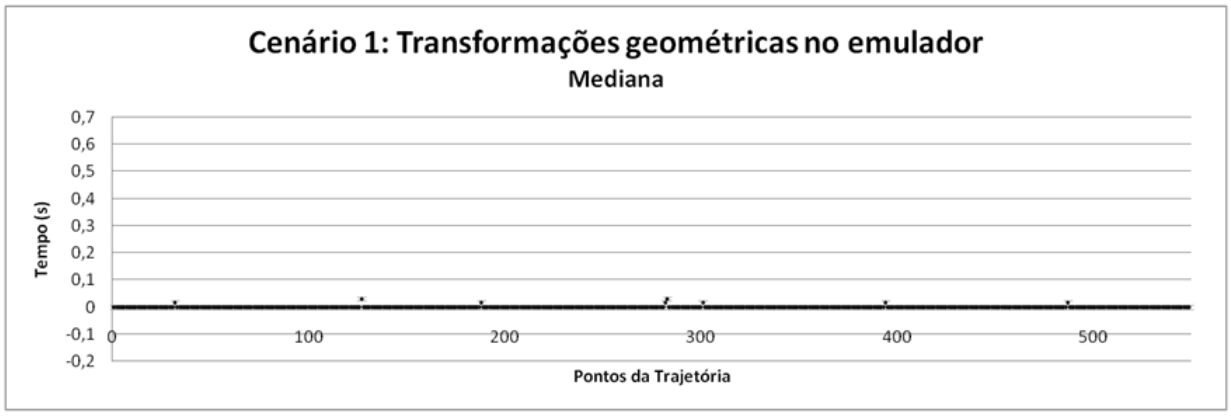
Figura 7.12: Comparação entre os tempos de renderização gastos pelos celulares no Cenário 3.



(a)

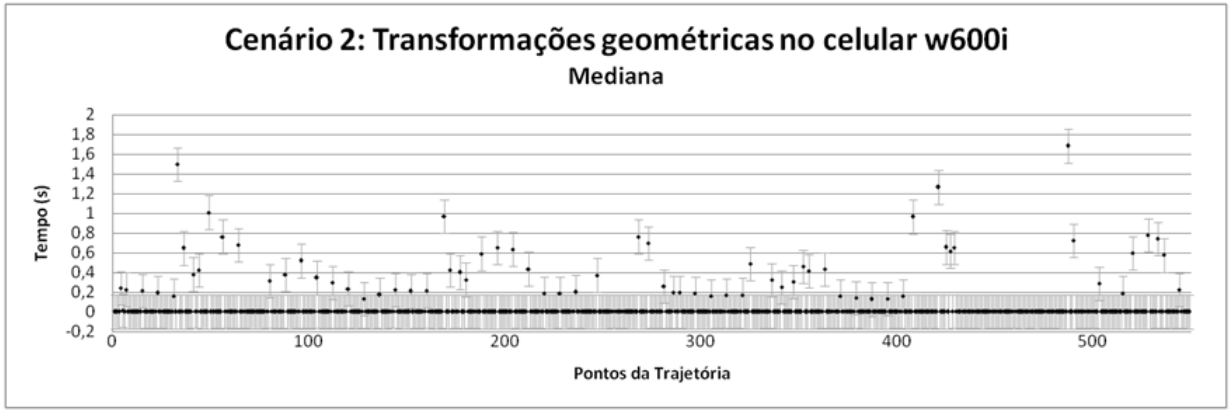


(b)

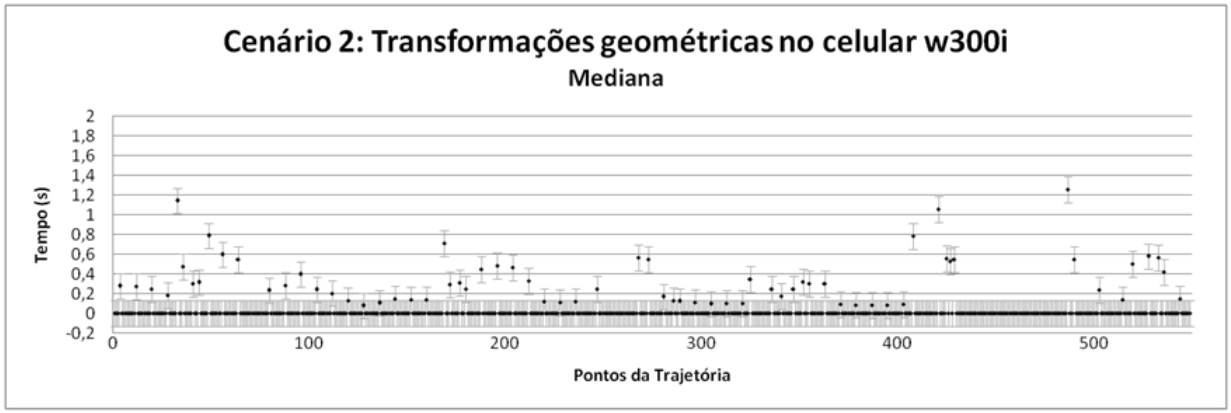


(c)

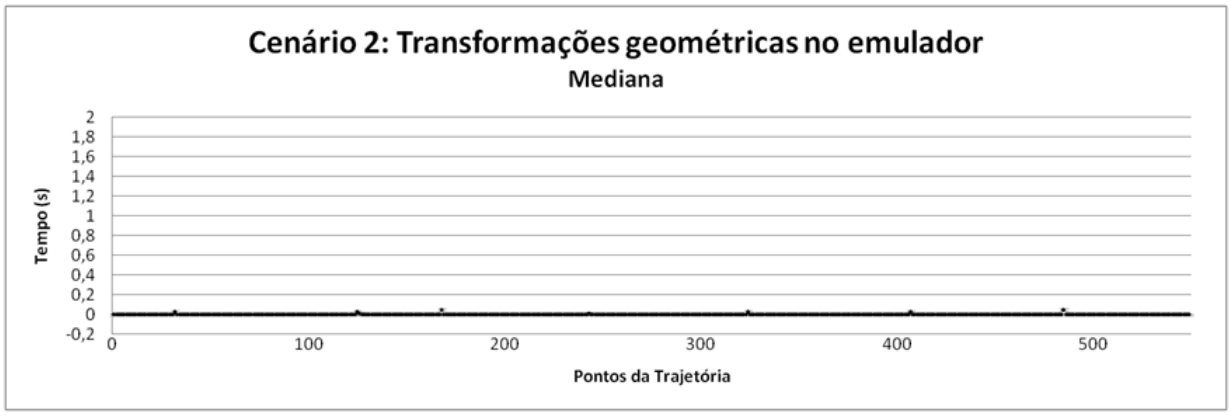
Figura 7.13: Valores de mediana do tempo para aplicação de transformações geométricas para o Cenário 1. Em (a), (b) e (c), o w600i, o w300i e o emulador, respectivamente. Os valores do desvio padrão em (a), (b) e (c) são 0,0882; 0,0583 e 0,0024s, respectivamente.



(a)

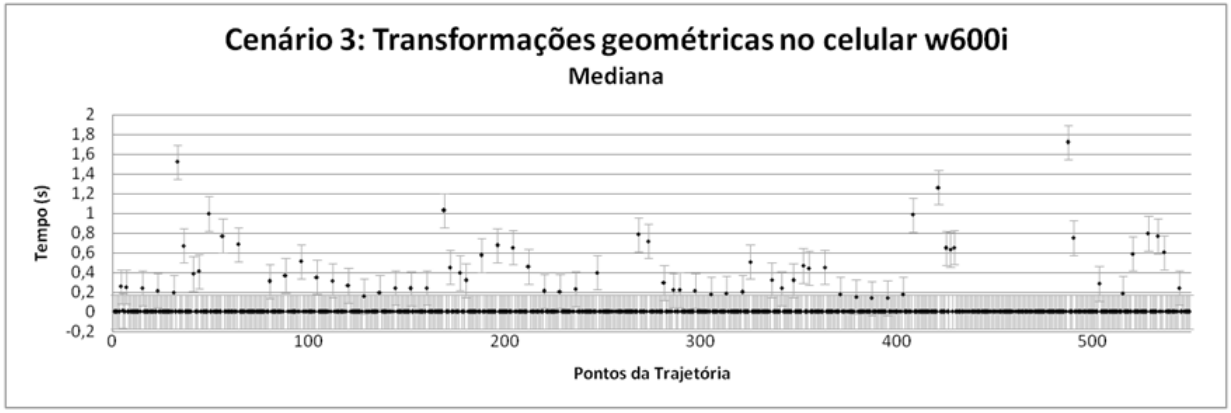


(b)

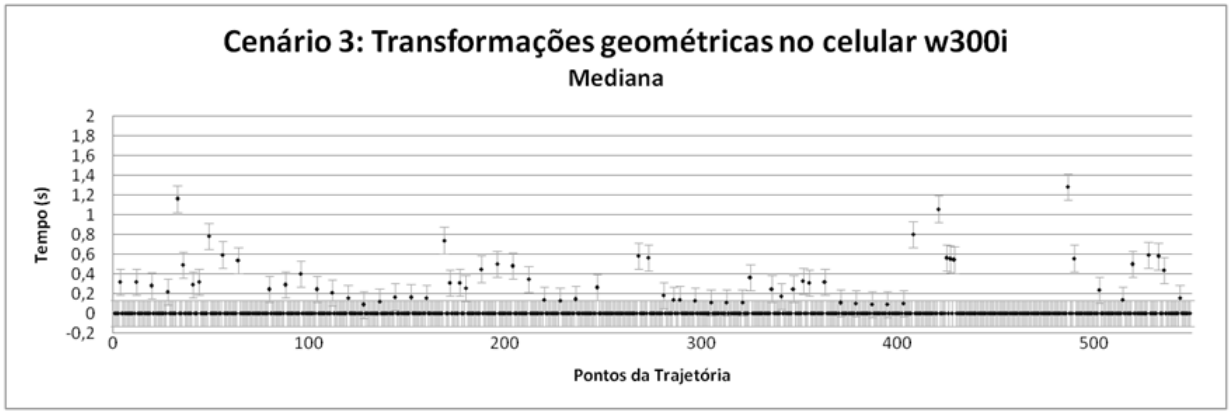


(c)

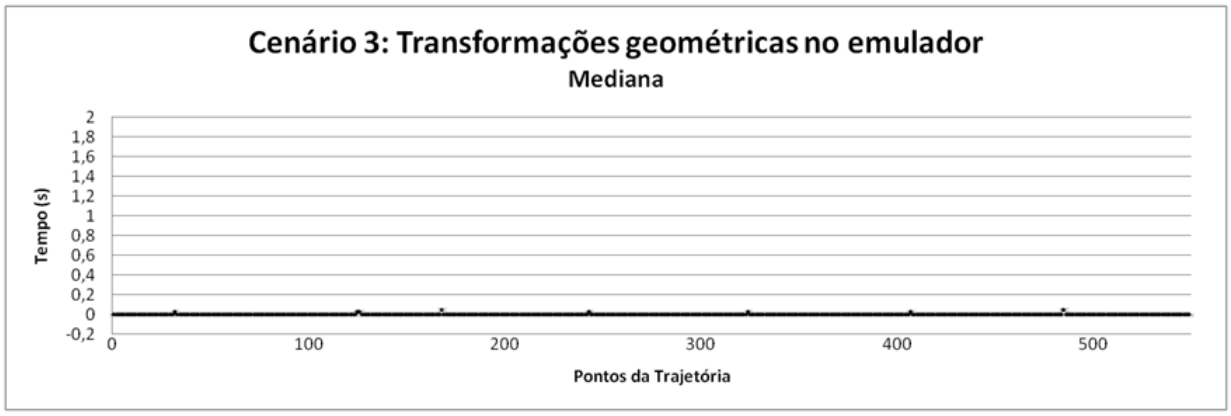
Figura 7.14: Valores de mediana do tempo para aplicação de transformações geométricas para o Cenário 2. Em (a), (b) e (c), o w600i, o w300i e o emulador, respectivamente. Os valores do desvio padrão em (a), (b) e (c) são 0,1860; 0,1438 e 0,0039s, respectivamente.



(a)



(b)



(c)

Figura 7.15: Valores de mediana do tempo para aplicação de transformações geométricas para o Cenário 3. Em (a), (b) e (c), o w600i, o w300i e o emulador, respectivamente. Os valores do desvio padrão em (a), (b) e (c) são 0,1904; 0,1469 e 0,0042s, respectivamente.

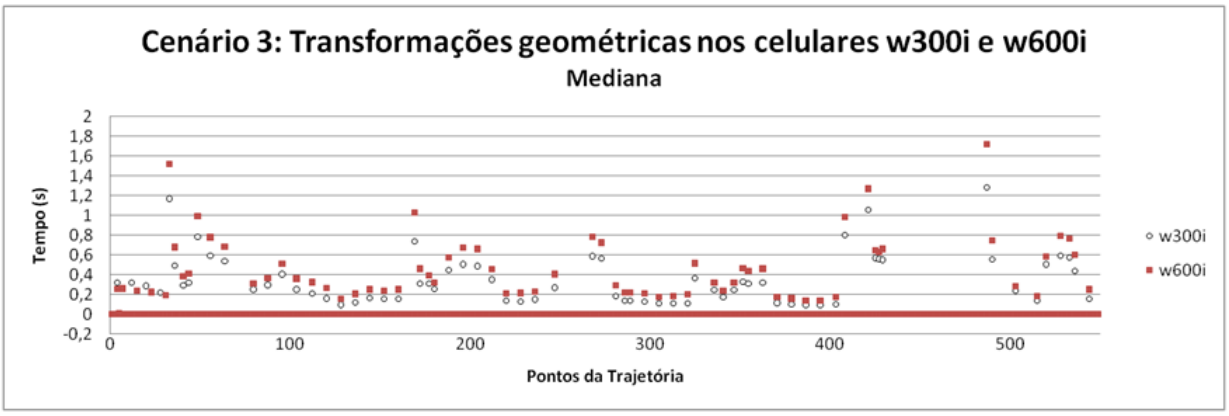


Figura 7.16: Comparação entre os tempos de renderização gastos pelos celulares no Cenário 3.

Capítulo 8

Conclusões

Como resultado desta pesquisa, foi desenvolvido um ambiente para visualização e navegação interativa em ambientes virtuais, utilizando dispositivos móveis. O sistema foi desenvolvido com base na API M3G, estendendo-a. O próprio sistema pode também ser estendido, através da implementação de novas funcionalidades ou da criação de métodos alternativos para realizar funções similares àquelas atualmente existentes, como outras envoltórias para a detecção de colisão e outros algoritmos de planejamento de trajetórias. Como forma de tornar o sistema robusto e flexível, o suficiente para adaptar-se a novos ambientes e dispositivos, este foi implementado tendo como base uma arquitetura modular genérica baseada no padrão MVC, que utiliza a API M3G. As extensões no sistema são facilitadas pelo uso desta arquitetura, projetada de forma a separar os principais componentes, em módulos. Assim, a inclusão ou modificação de uma funcionalidade é efetuada de forma simplificada, afetando apenas aquele módulo específico para onde a funcionalidade foi projetada.

Usuários do sistema podem utilizar os controles (teclas) dos dispositivos para navegar manualmente pelo cenário 3D, através da aplicação de transformações geométricas de translação e rotação. Para tornar a navegação manual mais realista, foram criadas novas classes que possibilitam detectar colisões 3D com os elementos gráficos dos cenários. Em particular, a detecção de colisão é efetuada através do uso de caixas envoltórias associadas aos objetos do ambiente virtual, uma nova funcionalidade que foi acrescentada à API M3G.

Outro recurso que o sistema implementado disponibiliza é a navegação automática (planejamento de trajetórias). Através da seleção de um elemento de interesse dentro do mundo virtual,

é gerado, automaticamente, um caminho livre de colisões que conecta a posição atual do usuário com a posição do elemento de interesse buscado pelo mesmo. Isso é possibilitado através do uso de novas classes que foram implementadas e que são responsáveis por gerar estruturas que auxiliam neste processo, como grafos de conectividade.

Três aplicações distintas foram desenvolvidas para ilustrar as funcionalidades do sistema: um museu virtual, um esquema para resgate de pessoas em perigo e um *shopping* virtual. Essas aplicações podem ser utilizadas para diversas tarefas: guiar usuários enquanto caminham fisicamente pelos ambientes, provê-los com informações detalhadas sobre os lugares e itens visitados, bem como criar caminhos automaticamente até um dado ponto de interesse. Baseando-se nos resultados obtidos, estudos preliminares mostraram que o sistema pode ser reutilizado para o desenvolvimento de aplicações gráficas 3D em celulares. Entretanto, é recomendável, para que o grau de reutilização do sistema seja melhor avaliado, que novas aplicações sejam desenvolvidas; métodos alternativos sejam implementados para as funcionalidades ora existentes (detecção de colisão e planejamento de trajetórias); e novos recursos gráficos possam ser adicionados ao sistema, possibilitando que o mesmo evolua futuramente para um *framework*.

Como forma de analisar o sistema implementado, o *shopping* virtual foi tomado como base para realização de testes em três plataformas diferentes: o emulador WTK da Sun Microsystems e dois celulares Sony Ericsson, w300i e w600i. Através da aplicação de seqüência de transformações geométricas, uma trajetória automática foi criada. Usando esta trajetória, pode-se analisar com sucesso o desempenho das plataformas e dos principais processos do sistema, como a renderização, as transformações geométricas, o carregamento de cenas e a detecção de colisões. Em particular, os testes mostraram que o emulador não é confiável como parâmetro único indicativo para se avaliar o desempenho da aplicação em celulares, pois claramente utiliza as configurações da máquina onde executa. Entre os celulares, os resultados mostraram que o w300i apresentou, em todos os cenários, um desempenho superior ao w600i, uma vez que precisou de um menor tempo para efetuar o cálculo dos processos do sistema. Entre os processos do sistema, a renderização em geral foi a que mais consumiu tempo de processamento. Entretanto, para os celulares, as transformações geométricas também consumiram muito tempo de processamento, indicando que tais métodos precisam ser otimizados para reduzir estes valores. Quanto à memória, apesar do w300i apresentar uma maior quantidade de *dynamic heap*, para estes testes específicos, ambos os celulares se comportaram de forma similar, ressaltando os pontos onde o coletor de lixo do Java foi executado (explicitamente, através de uma chamada de método na aplicação, ou implicitamente, através da própria máquina virtual). Por sua vez, o tempo gasto com o cálculo da detecção de colisão foi muito pequeno, indicando a eficiência do algoritmo implementado. Finalmente, o algoritmo de planejamento de trajetórias também foi implementado com sucesso.

Adicionalmente, a API M3G provou ser capaz de representar a visualização realista (cenas 3D com alta qualidade gráfica e um nível de detalhe razoável, incluindo vários objetos do tipo malha, compostos por mais de mil vértices e com texturas de alta resolução).

8.1 Contribuições

As contribuições decorrentes deste trabalho concentram-se no projeto, implementação e análise de desempenho de um sistema para navegação e visualização interativa em celulares. Além disso, este trabalho também resultou nas seguintes publicações em veículos científicos nacionais e internacionais:

- Barbosa, Rafael Garcia, Rodrigues, Maria Andréia Formico. Um *Framework* para Navegação em Ambientes Virtuais utilizando Dispositivos Móveis. Em Anais do *IX Symposium on Virtual and Augmented Reality* (SVR), 2007, p. 162-169.

- Rodrigues, Maria Andréia Formico; Barbosa, Rafael Garcia; Silva, Wendel Bezerra. Desenvolvimento de Aplicações 3D para Dispositivos Móveis utilizando as APIs M3G e OpenGL ES. *Revista de Informática Teórica e Aplicada* (RITA), 2006, v. XIII, p. 69-102.

- Barbosa, Rafael Garcia; Rodrigues, Maria Andréia Formico. Supporting Guided Navigation in Mobile Virtual Environments. Em Anais do 13^{th} *ACM Symposium Virtual Reality Software and Technology* (ACM VRST 2006), ACM Press, 2006. v. 1. p. 220-226.

- Rodrigues, Maria Andréia Formico; Barbosa, Rafael Garcia; Mendonça, Nabor das Chagas. Interactive Mobile 3D Graphics for On-the-go Visualization and Walkthroughs. Em Anais do 21^{st} *Annual ACM Symposium on Applied Computing* (ACM SAC 2006), *Special Track on Handheld Computing*, ACM Press, 2006. v. 2. p. 1002-1007.

- Barbosa, Rafael Garcia; Rodrigues, Maria Andréia Formico; Mendonça, Nabor das Chagas. Ambientes Virtuais Colaborativos para Dispositivos Móveis. Em Anais do 4^{th} Workshop de Teses e Dissertações em Computação Gráfica e Processamento de Imagens (WTDCGPI) do *XVIII Simpósio Brasileiro de Computação Gráfica e Processamento de Imagens* (SIBGRAPI), 2005 (meio digital, material nacional).

8.2 Trabalhos Futuros

Como trabalhos futuros, é antecipada a possibilidade do uso de colaboração entre os usuários de celulares que estejam presentes em um mesmo ambiente. Usuários possivelmente poderiam cooperar para a realização de uma dada tarefa, modificando a geometria e os atributos de uma célula. Entretanto, deve-se levar em consideração a heterogeneidade dos ambientes, que podem possuir diferentes capacidades de alcance para a comunicação entre os dispositivos e variações no poder de processamento e memória disponíveis.

Soluções alternativas para os métodos atualmente implementados podem também ser desenvolvidas. Por exemplo, para o módulo de detecção de colisão, esferas envoltórias podem ser utilizadas no lugar das caixas. Para o módulo de planejamento de trajetórias, campos potenciais podem ser utilizados para encontrar um caminho de navegação mínimo livre de obstáculos [21]. Essas novas implementações poderiam ainda ser comparadas com as atualmente existentes no sistema. Alternativamente, o algoritmo de planejamento de trajetórias poderia ser embutido localmente nos celulares, oferecendo, possivelmente, uma maior autonomia aos usuários dos dispositivos móveis.

A arquitetura proposta e implementada não foi sistematicamente utilizada para explorar todos os aspectos de visualização e exploração de mundos virtuais em um cenário móvel altamente dinâmico. Assim, é necessário incorporar mecanismos para particionamento automático dos mundos virtuais em células 3D, bem como utilizar técnicas de renderização que permitam a visualização de cenas 3D mais realistas (com efeitos especiais, tais como, fumaça e fogo, por exemplo).

Finalmente, com o uso de dispositivos móveis, onde a imersão é muito limitada, a eficiência das técnicas de interação e dos algoritmos de visibilidade é também de absoluta importância para o sucesso de aplicações interativas. Assim, é interessante utilizar métricas de usabilidade para quantificar o grau de interatividade alcançado pelas aplicações Java (com a API M3G), usando diferentes modelos de dispositivos móveis. Finalmente, métodos mais eficientes para renderização das cenas também podem ser implementados, reduzindo a quantidade de informação a ser desenhada e melhorando o desempenho da aplicação.

Referências Bibliográficas

[1] Ian Smith. Social Mobile Applications. *Computer*, 38(4):84–87, 2005.

[2] Sangmi Lee, Sunghoon Ko e Geoffrey Fox. Adapting Content for Mobile Devices in Heterogeneous Collaboration Environments. Em *Anais da International Conference on Wireless Networks*, p. 211–217. CSREA Press, 2003.

[3] Sun microsystems. JSR-184: Mobile 3D Graphics API for J2ME. Disponível em http://jcp.org/aboutJava/communityprocess/final/jsr184/index.html. Último acesso em 27/11/2007.

[4] Gregory D. Abowd, Christopher G. Atkeson, Jason Hong, Sue Long, Rob Kooper e Mike Pinkerton. Cyberguide: A Mobile Context-Aware Tour Guide. *Wireless Networks*, 3(5):421–433, 1997.

[5] Li-Der Chou, Chia-Hsieh Wu, Shih-Pang Ho e Chen-Chow Lee. Position-Aware Multimedia Mobile Learning Systems in Museums. Em *Anais da International Conference on Web-Based Education WBE*, 2004.

[6] Luca Chittaro, Lucio Ieronutti e Roberto Ranon. Navigating 3D Virtual Environments by Following Embodied Agents: A Proposal and Its Informal Evaluation on a Virtual Museum Application. *Psychnology*, 2(1):24–42, 2004.

[7] David M. Krum, William Ribarsky e Larry Hodges. Collaboration Infrastructure for a Mobile Situational Visualization System. Disponível em http://citeseer.ist.psu.edu/532479.html. Último acesso em 27/11/2007.

[8] Alberto Barbosa Raposo, Ivan Luiz Marques Ricarte, Luc Neumann e Léo Pini Magalhães. Visualization in a Mobile WWW Environment. Em *WebNet*, 1997.

[9] Antti Nurminen. m-LOMA - A Mobile 3D City Map. Em *Web3D'06: Anais do 11th International Conference on 3D Web Technology*, p. 7–18, New York, USA, 2006. ACM Press.

[10] André B. Trombetta, Felipe Bacim de A. e Silva e Márcio S. Pinho. Avaliação de Técnicas de Auxílio a Wayfinding em Ambientes Virtuais. Em *Anais do IX Symposium on Virtual and Augmented Reality SVR'07*, 2007.

[11] N. Di Blas, S. Hazan e P. Paolini. The SEE Experience. Edutainment in 3D Virtual Worlds. Em *Archives and Museum Informatics*, 2003.

[12] Rafal Wojciechowski, Krzysztof Walczak, Martin White e Wojciech Cellary. Building Virtual and Augmented Reality Museum Exhibitions. Em *Web3D '04: Anais do IX International Conference on 3D Web Technology*, p. 135–144, New York, USA, 2004. ACM Press.

[13] Katri Laakso, Ove Gjesdal e Jan Rasmus Sulebak. Tourist Information and Navigation Support by Using 3D Maps Displayed on Mobile Devices. Em *Anais do HCI in Mobile Guides, Workshop at Mobile HCI*, 2003.

[14] Heiko Blechschmied, Markus Etz e Jörg Haist. Providing of Dynamic Three-Dimensional City Models in Location-Based Services. Em *MOBILE MAPS 2005 - Interactivity and Usability of Map-based Mobile Services. A workshop., Mobile HCI*, 2005.

[15] Sébastien Paris, Stéphane Donikian e Nicolas Bonvalet. Environmental Abstraction and Path Planning Techniques for Realistic Crowd Simulation: Research Articles. *Computer Animation and Virtual Worlds*, 17(3-4):325–335, 2006.

[16] Brian Salomon, Maxim Garber, Ming C. Lin e Dinesh Manocha. Interactive Navigation in Complex Environments Using Path Planning. Em *I3D '03: Anais do 2003 Symposium on Interactive 3D Graphics*, p. 41–50, New York, USA, 2003. ACM Press.

[17] Fabrice Lamarche e Stéphane Donikian. Crowd of Virtual Humans: a New Approach for Real Time Navigation in Complex and Structured Environments. *Computer Graphics Forum*, 23(3):509–518, 2004.

[18] Tsai-Yen Li, Jyh-Ming Lien, Shih-Yen Chiu e Tzong-Hann Yu. Automatically Generating Virtual Guided Tours. Em *CA'99: Anais do Computer Animation*, p. 99–106, Washington, USA, 1999. IEEE Computer Society.

[19] Tsai-Yen Li e Hung-Kai Ting. An Intelligent User Interface with Motion Planning for 3D Navigation. Em *VR'00: Anais do IEEE Virtual Reality 2000 Conference*, p. 177–184, Washington, USA, 2000. IEEE Computer Society.

[20] Carlos Andújar Gran, Pere Pau Vázquez Alcocer e Marta Fairén González. Way-Finder: Guided Tours Through Complex Walkthrough Models. *Computer Graphics Forum*, 23(3):499–508, 2004.

[21] Qun Li, Michael De Rosa e Daniela Rus. Distributed Algorithms for Guiding Navigation Across a Sensor Network. Em *MobiCom'03: Anais do 9th Annual International Conference on Mobile Computing and Networking*, p. 313–325, New York, NY, USA, 2003. ACM Press.

[22] Lichan Hong, Shigeru Muraki, Arie Kaufman, Dirk Bartz e Taosong He. Virtual voyage: interactive navigation in the human colon. Em *SIGGRAPH'97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, p. 27–34, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[23] Brian W. Stout. Smart moves: Intelligent path-finding. *Game Developer*, p. 28–35, 2006.

[24] J. Mukerjee, P.P. Das e B.N. Chatterji. Thinning of 3d images using the safe point thinning algorithm (spta). *Pattern Recognition Letters*, 10(3):167–173, 1989.

[25] C. Wayne Niblack, Phillip B. Gibbons e David W. Capson. Generating skeletons and centerlines from the distance transform. *CVGIP: Graph. Models Image Process.*, 54(5):420–437, 1992.

[26] R. L. Ogniewicz e journal = Pattern Recognition volume = 28 number = 3 pages = 343-359 year = 1995 O. Kübler|, title = Hierarchic Voronoi Skeletons.

[27] Christian Kray e Jörg Baus. A Survey of Mobile Guides. Em *Workshop HCI in mobile guides*, 2003.

[28] Jean-Claude Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[29] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein. *Introduction to Algorithms*. MIT Press Cambridge, 1990.

[30] Srikanth Bandi e Daniel Thalmann. Space Discretization for Efficient Human Navigation. *Computer Graphics Forum*, 17(3):195–206, 1998.

[31] James J. Kuffner Jr. Goal-Directed Navigation for Animated Characters Using Real-Time Path Planning and Control. Em *CAPTECH'98: Anais do International Workshop on Modelling and Motion Capture Techniques for Virtual Environments*, p. 171–186, London, UK, 1998. Springer-Verlag.

[32] Marcelo Kallmann, Hanspeter Bieri e Daniel Thalmann. Fully dynamic constrained delaunay triangulations, 2003. Disponível em http://citeseer.ist.psu.edu/kallmann03fully.html. Último acesso em 27/11/2007.

[33] Christer Ericson. *Real-Time Collision Detection*. Elsevier, 2005.

[34] Sun microsystems. Sun Java Wireless Toolkit 2.5.2 for CLDC Download. Disponível em http://java.sun.com/products/sjwtoolkit/download.html. Último acesso em 27/11/2007.

[35] Sun microsystems. JSR-139: Connected Limited Device. Disponível em http://jcp.org/aboutJava/communityprocess/final/jsr139/index.html. Último acesso em 27/11/2007.

[36] Sun microsystems. JSR-37: Mobile Information Device Profile (MIDP). Disponível em http://jcp.org/aboutJava/communityprocess/final/jsr037/index.html. Último acesso em 27/11/2007.

[37] Sun microsystems. JSR-118: Mobile Information Device Profile 2.0. Disponível em http://jcp.org/aboutJava/communityprocess/final/jsr118/index.html. Último acesso em 27/11/2007.

[38] Nokia. Ri Binary For JSR-184 3D Graphics API For J2ME 1.0. Disponível em http://www.forum.nokia.com/main/resources/technologies/java/documentation/java_jsr.html. Último acesso em 27/11/2007.

[39] Sun microsystems. JSR-297: Mobile 3D Graphics API 2.0. Disponível em http://jcp.org/en/jsr/detail?id=297. Último acesso em 27/11/2007.

[40] Kari Pulli, Jani Vaarala, Ville Miettinen, Tomi Aarnio e Mark Callow. Developing mobile 3d applications with opengl es and m3g. Em *SIGGRAPH'05: ACM SIGGRAPH 2005 Courses*, p. 1, New York, NY, USA, 2005. ACM Press.

[41] Khronos group. OpenGL ES. Disponível em http://www.khronos.org/opengles/. Último acesso em 27/11/2007.

[42] Maria Andréia Formico Rodrigues, Rafael Garcia Barbosa e Wendel Bezerra Silva. Desenvolvimento de Alicações 3D para Dispositivos Móveis Utilizando as APIs M3G e OpenGL ES. *Revista de Informática Teórica e Aplicada - RITA*, 13(2):69–102, 2006.

[43] Khronos group. OpenGL. Disponível em http://www.opengl.org/. Último acesso em 27/11/2007.

[44] Sun microsystems. The java3d api. Disponível em http://java.sun.com/products/java-media/3D/. Último acesso em 27/11/2007.

[45] Autodesk. 3ds max 7. Disponível em http://usa.autodesk.com/adsk/servlet/ index?id=5659302&siteID=123112. Último acesso em 27/11/2007.

[46] Maria Andréia Formico Rodrigues, Rafael Garcia Barbosa e Nabor das Chagas Mendonça. Interactive Mobile 3D Graphics for On-the-go Visualization and Walkthroughs. Em *SAC'06: Anais do 2006 ACM Symposium on Applied Computing*, p. 1002–1007, New York, NY, USA, 2006. ACM Press.

[47] Rafael Garcia Barbosa, Maria Andréia Formico Rodrigues e Nabor das Chagas Mendonça. Ambientes virtuais colaborativos para dispositivos móveis. Em *Anais do IV Workshop de Teses e Dissertações em Computação Gráfica e Processamento de Imagens (WTDCGPI 2005), em XVIII SIBGRAPI*, p. 1–7, 2005.

[48] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad e Michael Stal. *Pattern-Oriented Software Architecture*. Wiley, 1996.

[49] Rafael Garcia Barbosa e Maria Andréia Formico Rodrigues. Supporting Guided Navigation in Mobile Virtual Environments. Em *VRST'06: Anais do ACM symposium on Virtual reality software and technology*, p. 220–226, New York, NY, USA, 2006. ACM Press.

[50] Rafael Garcia Barbosa e Maria Andréia Formico Rodrigues. Um Framework para Navegação em Ambientes Virtuais Utilizando Dispositivos Móveis. Em *Anais do IX Symposium on Virtual and Augmented Reality (SVR 2007)*, p. 162–169, 2007.

[51] Sun microsystems. JSR 82: Javatm APIs for Bluetooth. Disponível em http://www.jcp.org/en/jsr/detail?id=82. Último acesso em 27/11/2007.

[52] Bluetooth SIG. Bluetooth basics - range. Disponível em http://bluetooth.com/ Bluetooth/Learn/. Último acesso em 27/11/2007.

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