UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

MJ3D-PORTAIS – BIBLIOTECA DE ALGORITMOS DE PORTAIS 2D UTILIZANDO CENÁRIOS EM 3D PARA A PLATAFORMA IOS

MATEUS JUNIOR CASSANIGA

BLUMENAU
2013

2013/1-24

MATEUS JUNIOR CASSANIGA

MJ3D-PORTAIS – BIBLIOTECA DE ALGORITMOS DE PORTAIS 2D UTILIZANDO CENÁRIOS EM 3D PARA A PLATAFORMA IOS

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. Dalton Solano dos Reis , M.Sc. - Orientador

BLUMENAU
2013

2013/1-24

MJ3D-PORTAIS – BIBLIOTECA DE ALGORITMOS DE PORTAIS 2D UTILIZANDO CENÁRIOS EM 3D PARA A PLATAFORMA IOS

Por

MATEUS JUNIOR CASSANIGA

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

	_____
Presidente:	Prof. Dalton Solano dos Reis, M. Sc. – Orientador, FURB
	_____
Membro:	Prof. Aurélio Faustino Hoppe, M. Sc. – FURB
	_____
Membro:	Prof. Francisco Adell Péricas, M. Sc. – FURB

Blumenau, 08 de julho de 2013

Dedico este trabalho a todos que me apoiaram até hoje. Especialmente aos meus pais e melhores amigos.

AGRADECIMENTOS

Agradeço primeiramente aos meus pais, Délcio José Cassaniga e Edite Cassaniga que sempre estiveram do meu lado. A realização deste trabalho se deve primeiramente ao apoio incondicional deles.

Aos meus amigos, que considero minha segunda família. Sempre estiverem presentes em todos os momentos.

Ao meu orientador, Dalton Solano dos Reis, por ter me dado a ideia e por seu profissionalismo e atenção dados a minha pessoa durante todo o desenvolvimento deste trabalho.

A empresa aonde trabalho, Fácil, por ter flexibilizado meu horário para que eu pudesse concluir este trabalho.

Brincar é condição fundamental para ser sério

Arquimedes

RESUMO

Este trabalho apresenta a especificação, conversão e aperfeiçoamento de uma biblioteca que aplica os conceitos de algoritmo de portais. A biblioteca foi convertida da plataforma Android para a plataforma iOS e passou a permitir a utilização de cenários em 3D desenvolvidos através da ferramenta de modelagem Blender. A maior parte dos cálculos matemáticos da biblioteca original não funcionou com os novos mapas, sendo necessário desenvolvê-los de outras maneiras. Também apresenta uma aplicação de testes utilizando a linguagem de programação Objective-C, a biblioteca gráfica OpenGL ES 2.0 e a biblioteca desenvolvida, podendo ser utilizada nos dispositivos iPad e iPhone.

Palavras-chave: Algoritmo de portais. Plataforma iOS. Computação gráfica.

ABSTRACT

This work presents the specification, conversion and improvement of a library that applies the concepts of portal culling. The library was converted from Android platform to iOS platform, and allows the use of scenarios developed by 3D modeling tool Blender. The majority of the mathematical calculations of the original library did not work with the new maps, being necessary to develop them in other ways. It also presents a test application developed with the Objective-C programming language, the OpenGL ES 2.0 graphical library and the developed library, which may be used on iPhone and iPad.

Key-words: Portal culling. iOS platform. Graphics computing.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

2D – Duas dimensões

3D – Três dimensões

API – *Application Programming Interface*

CPU – *Central Processing Unit*

FPS – *Frames Per Second*

GHz – *Giga Hertz*

GPU – *Graphics Processing Unit*

IDC – *International Data Corporation*

iOS – *iPhone Operating System*

MB – *Mega Bytes*

OpenGL ES – *Open Graphics Library for Embedded Systems*

RF – Requisito Funcional

RNF – Requisito Não Funcional

SDK – *Software Development Kit*

UML – *Unified Modeling Language*

SUMÁRIO

1 INTRODUÇÃO

Atualmente vive-se a era da mobilidade. A utilização de dispositivos móveis, como *smartphones* e *tablets* estão tornando-se cada vez mais popular. De acordo com uma pesquisa realizada pelo *International Data Corporation* (IDC), estima-se que só no Brasil, devem ser vendidos cerca de 15,4 milhões de *smartphones* em 2012 (PETRY, 2012). Já o mercado de *tablets* terá um crescimento de 98% em relação às vendas globais, atingindo cerca de 118,9 milhões de dispositivos vendidos em 2012 (AGUIARI, 2012).

Com o aumento no uso de dispositivos móveis, surge no mercado uma vasta gama de aplicativos, de diversas categorias, desenvolvidos para estes dispositivos. Dentre estas categorias está a de entretenimento (jogos). Com a evolução dos *smartphones* e *tablets*, tornou-se viável a utilização destes dispositivos móveis para jogos. No mercado há dispositivos com capacidade de processamento necessária para permitir a execução de jogos bem mais elaborados, o que permite um aumento de realismo nas cenas e uma qualidade melhor nos gráficos. O problema encontrado no desenvolvimento de jogos que utilizam estas características está no custo de processamento necessário ao produzir estas cenas.

No desenvolvimento de jogos existem técnicas utilizadas para se diminuir o custo e otimizar o processamento de cenas. Algumas destas técnicas utilizam algoritmos que determinam quais objetos na cena devem ser renderizados. Entre estes algoritmos está o algoritmo de portais, que é uma técnica de *culling* utilizada para diminuir o número de objetos que vão ser processados pela placa gráfica. A ideia geral do algoritmo é dividir a cena em células ligadas através de portais. Se a câmera está dentro de uma célula, os objetos pertencentes às outras células só podem ser vistos através de portais. Com isso pode-se obter uma redução considerável de objetos processados. Esta técnica também pode ser utilizada para determinar quais objetos da cena estão dentro do campo de visão de um personagem virtual, muito utilizado em animações comportamentais.

Diante do exposto, converteu-se a biblioteca de algoritmos de portais desenvolvida por Pandini (2012) no sistema operacional Android para a plataforma *iPhone Operating System* (iOS). Também são utilizados cenários em três dimensões (3D), bem como obteve-se um melhor desempenho no algoritmo de teste de visibilidade.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é converter e estender para a plataforma iOS a biblioteca de algoritmos de portais desenvolvida por Pandini (2012).

Os objetivos específicos são:

a) converter as funcionalidades já existentes para a plataforma iOS;
b) estender o trabalho de Pandini (2012) para permitir utilizar cenários em 3D, mantendo os testes dos portais em duas dimensões (2D);
c) testar outras formas de verificar se um ponto está dentro de um triângulo para melhorar o desempenho do teste de visibilidade.

1.2 ESTRUTURA DO TRABALHO

Tendo sido apresentados a introdução e os objetivos, no capítulo dois é descrita a fundamentação teórica. Nele são apresentados explicações sobre algoritmos de visibilidade (com ênfase no algoritmo de portais), a plataforma iOS de desenvolvimento, a biblioteca *Open Graphics Library for Embedded Systems* (OpenGL ES) e a biblioteca de algoritmos de portais desenvolvida por Pandini (2012). Ainda no capítulo dois, discorre-se sobre trabalhos correlatos à biblioteca proposta.

No capítulo três são apresentados os requisitos, a especificação, a implementação e o funcionamento da biblioteca proposta. Também são apresentados os resultados obtidos ao termino do trabalho, assim como uma discussão sobre os mesmos. Este capítulo trata de cada etapa desenvolvida, dos algoritmos e técnicas utilizadas na implementação e dos resultados obtidos através da biblioteca.

O quarto e último capítulo trata das conclusões do presente trabalho, juntamente com sugestões de possíveis extensões do mesmo.

2 FUNDAMENTAÇÃO TEÓRICA

Na seção 2.1 é apresentada uma visão geral sobre algoritmos de visibilidade. Na seção 2.2 é apresentada a plataforma iOS e suas características. Na seção 2.3 é descrita a biblioteca gráfica OpenGL ES. A seção 2.4 aborda a biblioteca de algoritmos de portais desenvolvida para Android. Por fim, na seção 2.5, são citados três trabalhos correlatos.

2.1 ALGORITMOS DE VISIBILIDADE

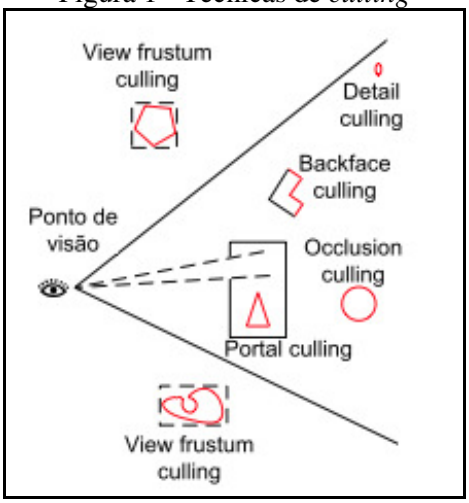
Segundo Tavares (2009, p. 15), "o conceito de visibilidade é uma abstração matemática da noção de visibilidade da vida real". Um dos mais importantes problemas da computação gráfica é a classificação dos objetos em uma cena, determinando se eles estão pelo menos parcialmente visíveis ou totalmente invisíveis (HADWIGER; VARGA, 1998, p. 1). O ser humano consegue identificar facilmente partes visíveis de um objeto em um ambiente. Entretanto, a determinação de visibilidade para muitos objetos em uma cena é uma tarefa difícil para o computador (TAVARES, 2009, p. 15).

Uma das técnicas para determinação de visibilidade é a técnica de *culling*. A ideia dos algoritmos de *culling* é determinar a visibilidade dos objetos antes que eles sejam enviados para a placa gráfica (CARLOS, 2009, p. 18). Alguns tipos de algoritmos de *culling* são:

a) *view-frustum culling*: descarta objetos que estão fora do campo de visualização;
b) *backface culling*: descarta as faces dos objetos que não estão voltadas para o observador, ou seja, não estão visíveis. Um exemplo pode ser dado ao remover todas as faces que estão localizadas atrás dos objetos;
c) *occlusion culling*: descarta objetos ocultos por outros objetos;
d) *detail culling*: descarta objetos que ocupam uma pequena parcela da área da tela. Ocorre normalmente quando um objeto está muito longe da tela e não irá contribuir muito com a imagem final;
e) *portal culling*: conhecida também como algoritmo de portais. Divide um cenário 3D em células e portais, renderizando os objetos localizados em outras células apenas se eles estão visíveis através de um portal.

Na Figura 1 é dado um exemplo das técnicas de *culling*. Partes ou objetos com contorno vermelho não serão renderizados.

Figura 1 - Técnicas de *culling*



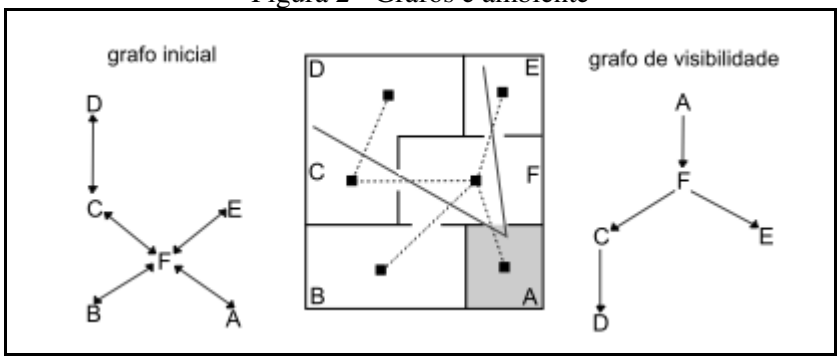
Fonte: Carlos (2009, p. 19).

Na Figura 1 são representadas as técnicas de *culling*. O ponto de visão é onde o observador está. As linhas saindo do observador compõem o *frustum* de visão do mesmo. No exemplo, os objetos (ou parte deles) que estão com contorno vermelho não serão renderizados. Observa-se que a técnica de *portal culling* faz com que o observador gere um novo *frustum* (representado na imagem com a linha tracejada) com o tamanho do portal. Na imagem, o triângulo seria renderizado apenas se estivesse dentro desse novo frustum.

2.1.1 Algoritmo de Portais

Uma técnica de *culling* utilizada principalmente em ambientes internos é conhecida como algoritmo de portais (do inglês *portal culling*) (SILVA, 2008, p. 31). Esta técnica tem como ideia principal dividir uma cena 3D em células ligadas através de portais. Estas células e portais são representados através de grafos. Células correspondem aos nós (vértices) e portais as conexões (arestas). Com isso cria-se um grafo de visibilidade (SILVA, 2008, p. 31-32). A Figura 2 ilustra o algoritmo tendo como cena um ambiente subdividido em seis células (A, B, C, D, E, F), contendo ao todo quatro portais.

Figura 2 - Grafos e ambiente



Fonte: Carlos (2009, p. 19).

Na Figura 2 tem-se o ambiente (centro da imagem) subdividido em seis células (A, B, C, D, E, F). Como grafo inicial é criado um grafo com todas as ligações possíveis das células (A – F, B – F, E – F, C – F, D – C). A partir deste grafo é gerado o grafo de visibilidade (A – F, C – F, E – F, C – D). O grafo de visibilidade elimina a célula B por não estar visível pelo *frustum* (campo de visão da câmera).

O processo de renderização das cenas, quando se utiliza o algoritmo de portais, é dividido em duas etapas (SILVA, 2008, p. 32). A primeira etapa consiste em iniciar a renderização na célula onde a câmera encontra-se (na Figura 2 representada pela letra A). Esta célula é renderizada normalmente. Porém, por cada portal visível na célula é criado um *frustum* do mesmo tamanho e então a célula visível dentro deste *frustum* é renderizada. Isso ocorre de maneira recursiva até que todos os portais visíveis dos subambientes sejam processados (GAME RENDERING, 2008). A segunda etapa determina quais objetos pertencentes às células visíveis estão dentro deste campo de visão. A Figura 3 ilustra esta etapa do algoritmo.

Figura 3 – Na imagem, apenas os objetos com preenchimento são renderizados

Fonte: Silva et al. (2003).

2.2 PLATAFORMA IOS

iOS é o sistema operacional que roda no iPhone, iPod touch e iPad. Este sistema operacional gerencia o hardware do dispositivo e fornece as tecnologias necessárias para programar aplicações nativas (APPLE INC, 2012).

O *Software Development Kit* (SDK) do iOS contém as ferramentas necessárias para o desenvolvimento, teste e depuração de aplicativos nativos da plataforma (APPLE INC, 2012). Aplicativos nativos são desenvolvidos utilizando os *frameworks* do iOS, a linguagem de programação Objective-C e rodam diretamente sobre a plataforma iOS (APPLE INC, 2012). Junto com o SDK está incluso o Xcode, ambiente de desenvolvimento utilizado para gerenciar projetos, permitindo editar, compilar, rodar e depurar aplicações desenvolvidas para a plataforma (APPLE INC, 2012).

Para testar as aplicações desenvolvidas, o SDK inclui o iOS Simulator, aplicativo que simula um dispositivo com iOS em um computador Macintosh com processador Intel (APPLE INC, 2012).

2.3 OPENGL ES

OpenGL ES é uma versão compacta e adaptada para sistemas embarcados da biblioteca gráfica OpenGL. A primeira versão, OpenGL ES 1.0, teve como objetivo prover uma biblioteca gráfica extremamente compacta sem sacrificar seus recursos. Teve que ser implementada totalmente por software, não podendo ultrapassar os 50kB de código e ao mesmo tempo ser bem adequada a aceleração de hardware. Esta versão compacta foi desenvolvida para que os dispositivos móveis pudessem usufruir dos efeitos gráficos até então disponíveis apenas em computadores pessoais (PULLI et al., 2008, p. 18).

Dispositivos que rodam iOS permitem a utilização de duas versões do OpenGL ES. A versão 1.1 e a 2.0. A versão 1.1 é suportada por todos os dispositivos que rodam iOS, enquanto a versão 2.0 é suportada apenas por dispositivos mais novos, como o iPhone 3GS ou superior, iPad de todas as gerações e iPod touch a partir da 3ª geração (SUGRUE, 2011, p. 6). Com o OpenGL ES, no iPhone é possível ter acesso (através de uma boa interface) a recursos da *Graphics Processing Unit* (GPU) permitindo criar objetos em espaços 2D e 3D, aplicar cores, texturas e transformações nestes objetos (SADUN, 2009, p. 27-28). Por usar a GPU ao invés da *Central Processing Unit* (CPU), a renderização gráfica é feita via hardware, o que torna o processo mais rápido. A GPU é capaz de realizar operações de ponto flutuante 100 vezes mais rápido que a CPU (SADUN, 2009, p. 28).

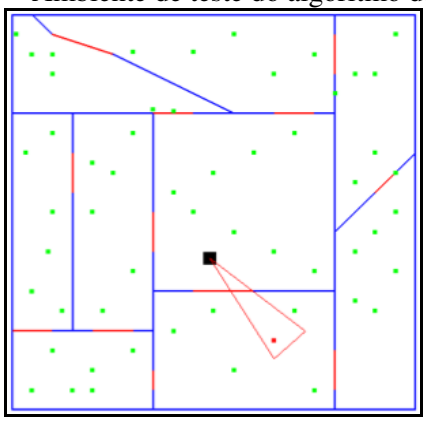
2.4 BIBLIOTECA DE ALGORITMOS DE PORTAIS PARA A PLATAFORMA ANDROID

Utilizando-se da linguagem de programação Java e da *Application Programming Interface* (API) de desenvolvimento para Android 2.3.4, Pandini (2012) desenvolveu uma

biblioteca de algoritmo de portais para a plataforma Android. Segundo Pandini (2012, p. 47), o algoritmo de *portal culling* desenvolvido para a biblioteca apresenta um bom desempenho, conseguindo lidar com uma grande quantidade de pontos de interesse sem errar ao identificar os pontos de interesse dentro do campo de visão. Pandini (2012, p. 48) afirma ainda que durante os testes foi encontrado um problema ao renderizar uma grande quantidade de pontos.

Referente a extensões, foram dadas algumas sugestões como, por exemplo, permitir que o usuário especifique o tamanho do campo de visão, permitir a configuração do ângulo de visão, acoplar a biblioteca ao OpenGL ES para permitir o uso em otimização de renderização, estudar a aplicação de outras técnicas de visibilidade e um editor de ambientes. A Figura 4 demonstra a execução do algoritmo de portais em um ambiente de testes desenvolvido por Pandini (2012).

Figura 4 – Ambiente de teste do algoritmo de portais



Fonte: Pandini (2012, p. 41).

Na Figura 4 o observador é representado pelo ponto preto, seu campo de visão (*frustum*) é representado pelo triângulo vermelho que sai do observador. Paredes não podem ser ultrapassadas e estão na cor azul. Portais fazem a ligação entre salas e estão na cor vermelha. Os pontos verdes são todos os objetos disponíveis nas salas que podem ser renderizados. Quando um ponto muda de verde para vermelho entende-se que o mesmo está sendo visto pelo observador.

2.5 TRABALHOS CORRELATOS

Abaixo são descritos três trabalhos acadêmicos relacionados ao tema. Entre eles está o "Frustum Culling Híbrido em CPU e GPU para Visualização de Modelos Massivos em Tempo Real" (CARLOS; RAPOSO; SOARES, 2010), "Renderização Interativa em Dispositivos Móveis utilizando Algoritmos de Visibilidade e Estruturas de Particionamento Espacial" desenvolvido por Silva (2009) e por fim, "Experiência de Portais em Ambientes Arquitetônicos Virtuais", desenvolvido por Silva et al. (2003).

2.5.1 Frustum culling híbrido em CPU e GPU para visualização de modelos massivos em tempo real

Este trabalho descreve uma nova técnica que utiliza o poder computacional da GPU para criar um algoritmo híbrido que aproveita o melhor da CPU e GPU. Para demonstrar a eficiência do algoritmo proposto os autores realizaram diversas comparações com implementações de algoritmos clássicos de *frustum culling*. "Como principal resultado desta pesquisa está o desenvolvimento de um *frustum culling* híbrido utilizando a GPU em momentos oportunos para determinar quais os objetos estão visíveis pela câmera em conjunto com a CPU" (CARLOS; RAPOSO; SOARES, 2010, p. 175). O objetivo foi comparar as duas implementações e encontrar seus pontos positivos e negativos.

As implementações foram testadas utilizando modelos 3D e caminhos construídos sobre estes modelos, tentando explorar o maior número de situações possíveis.

Nos testes propostos o algoritmo de *frustum culling* híbrido (com os devidos parâmetros especificados) teve um melhor desempenho. A partir dos resultados obtidos, os autores concluíram que a técnica apresentada trás melhoria significativa em relação as técnicas convencionais. O Quadro 1 mostra uma comparação de desempenho em *Frames Per Second* (FPS) dos métodos. P-38, P-40, P-43, P-50 e Boeing são nomes dados aos modelos tridimensionais.

Quadro 1 - Comparação de desempenho entre os métodos desenvolvidos

Método	FPS Mínimo	FPS Máximo	Média do Caminho
CPU na P-38	2368	7241	5494
Híbrido na P-38	**2756**	**7301**	**5615**
CPU na P-40	320	7263	3143
Híbrido na P-40	**596**	**7266**	**3490**
CPU na P-43	358	**7334**	3265
Híbrido na P-43	**527**	7270	**3627**
CPU na P-50	311	**7256**	3022
Híbrido na P-50	**578**	7241	**3505**
CPU no Boeing	96	7259	2605
Híbrido no Boeing	**280**	**7300**	**4629**

Fonte: Carlos, Raposo e Soares (2010, p. 183).

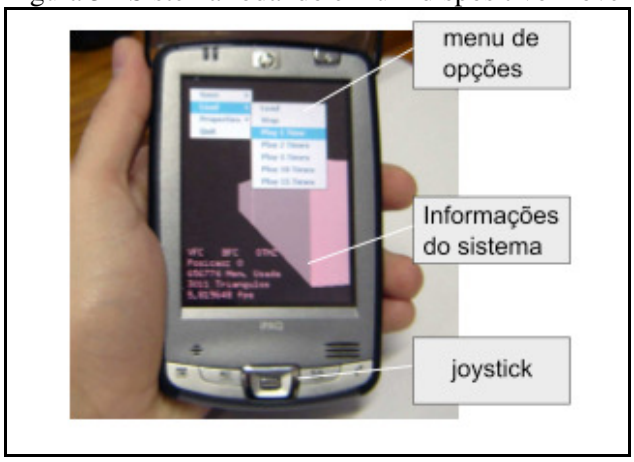
No Quadro 1 tem-se a comparação entre os métodos desenvolvidos. Quanto maior o valor em FPS, maior o desempenho. Observa-se que o método que utiliza processamento híbrido em CPU e GPU na maioria das vezes mostrou-se superior em questões de desempenho. Em todos os exemplos citados no Quadro 1, o método híbrido teve uma média de FPS superior a implementação do algoritmo que é executado apenas na CPU.

2.5.2 Renderização interativa em dispositivos móveis utilizando algoritmos de visibilidade e estruturas de particionamento espacial

O objetivo do trabalho de Silva (2009) é desenvolver um sistema para renderização em tempo real de cenas 3D em dispositivos móveis usando a biblioteca gráfica OpenGL ES. Silva (2009) utiliza várias combinações de algoritmos de visibilidade, associados a diferentes estruturas de particionamento espacial para comparar seus resultados.

Segundo Silva (2009), "Aplicações gráficas em Medicina, Engenharia e Entretenimento vêm utilizando ambientes 3D amplos, com uma quantidade massiva de dados. Assim, implementar otimizações no processo de renderização é importante [...]". O autor também apresenta um novo algoritmo de *culling*, denominado *backface culling* conservativo, o qual alega ser simples, porém eficiente. Também realiza a implementação e combinação de vários algoritmos (*view-frustum culling*, *backface culling* e *occlusion culling*), além de resolver problemas de navegação em tempo real utilizando estruturas de dados espaciais. A Figura 5 demonstra a execução do sistema.

Figura 5 - Sistema rodando em um dispositivo móvel



Fonte: Silva (2009, p. 59).

Para a análise de desempenho, das estruturas de dados e dos algoritmos de visibilidade, foram realizados 720 testes em dois dispositivos móveis diferentes (Pocket PC iPaq hx2490b e celular Nokia n82), variando-se os ambientes gráficos 3D modelados (locais, complexidade e disposição de objetos) e as trajetórias da câmera pelos ambientes. Com os resultados obtidos o autor concluiu que a renderização em tempo real de ambientes 3D executados em dispositivos móveis pode ser obtida com sucesso, se as combinações corretas de algoritmos de visibilidade e estruturas de particionamento espacial forem escolhidas.

2.5.3 Experiência de portais em ambientes arquitetônicos virtuais

Silva et al. (2003) utilizam o algoritmo de portais para melhorar o desempenho da navegação em tempo real em ambientes arquitetônicos virtuais com alto realismo gráfico. Os autores realizam a implementação do algoritmo de portais utilizando uma biblioteca de grafo de cena. Para a navegação no ambiente virtual é utilizada uma biblioteca de realidade virtual.

O modelo arquitetônico criado para os testes possui definições de células e portais codificados nos nomes dos objetos. Para identificar a qual célula pertence um objeto, os autores utilizam relações hierárquicas (objeto é filho de uma célula pai) (SILVA et al., 2003). A Figura 6 ilustra o modelo arquitetônico utilizado nos testes.

Figura 6 – Modelo arquitetônico utilizado nos testes

Fonte: Silva et al. (2003).

Nos testes realizados é utilizado um modelo de apartamento com milhares de faces, algumas células e portais. Através do teste proposto os autores concluem que ouve ganho de desempenho com o algoritmo de portais para o modelo utilizado. Silva et al. (2003) fazem uma observação ao citar que em alguns casos o desempenho foi equivalente, utilizando ou não o algoritmo de portais e afirma que o fato ocorre por dois motivos: complexidade dos objetos de uma única célula ou células que foram mal divididas.

3 DESENVOLVIMENTO

Neste capítulo é apresentado o desenvolvimento da biblioteca, a aplicação de testes e demonstração. Na parte de desenvolvimento serão abordados o *add-on*[1] desenvolvido para exportar os mapas utilizados na biblioteca, o desenvolvimento da biblioteca de portais e a biblioteca gráfica utilizada para a renderização do cenário 3D. São abordados os principais requisitos, a especificação e implementação. Por fim será apresentado os resultados e discussões.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

A biblioteca deverá:

a) ter uma implementação de algoritmos de portais (Requisito Funcional – RF);
b) permitir que o ambiente (mapa) seja particionado e separado em salas (RF);
c) permitir a utilização de cenários 3D (RF);
d) executar na plataforma iOS (Requisito Não Funcional – RNF);
e) utilizar o SDK fornecido para a plataforma iOS (RNF);
f) utilizar a biblioteca gráfica OpenGL ES (RNF);
g) ser testado nos dispositivos físicos iPhone e iPad (RNF);
h) testar os cenários com diferentes quantidades de salas e portais para aferir o tempo de execução dos principais métodos (RNF).

3.2 ESPECIFICAÇÃO

Nesta seção é apresentada uma relação dos nomes dos objetos que representam o mapa utilizado na aplicação. São apresentados também os diagramas de casos de uso, de classes e de sequência da *Unified Modeling Language* (UML), os quais foram especificados utilizando a ferramenta Enterprise Architect.

3.2.1 Criação de cenários utilizados na aplicação

Um cenário (mapa) pode conter várias salas, portais e pontos de interesse. Para permitir a criação de cenários customizados optou-se pela possibilidade de desenvolve-los com a ferramenta de modelagem Blender. Tendo assim a possibilidade de criação dos mapas 2D e de sua representação 3D na mesma ferramenta.

[1] É um script utilizado para estender as funcionalidades do Blender (BLENDER, 2013).

Considerando esta possibilidade, torna-se necessário identificar e listar quais nomes de objetos a ferramenta é capaz de detectar e interpretar para a geração do cenário. Esta relação de nomes de objetos é apresentado no Quadro 2.

Quadro 2 - Nome dos objetos reconhecidos pela aplicação

Nome	Descrição
OBSERVADOR_XX_YY	Representa o observador no cenário. Um observador é o personagem que se movimenta entre as salas e enxerga os pontos de interesse. O observador deve ser representado no Blender por um objeto do tipo MESH e possuir as dimensões aproximadas de 0.2,0.2,0.2 (x,y,z), posicionado logo acima do mapa da sala. No seu nome deve conter o código seqüencial (XX) e a sala em que ele se encontra (YY).
WAYPOINT_XX_YY	Representa um ponto de interesse. Pontos de interesse são vistos e desenhados pelo observador. O ponto de interesse deve ser representado no Blender por um objeto do tipo MESH e possuir as dimensões aproximadas de 0.2,0.2,0.2 (x,y,z), posicionado logo acima do mapa da sala. No seu nome deve conter o código seqüencial (XX) e a sala em que ele se encontra (YY).
MAPA_SALA_XX	Representa uma sala em duas dimensões no cenário. Não deve possuir faces e deve ser representada somente por arestas. Os únicos vértices que deve conter são vértices localizados no término de cada extremidade da aresta (comumente em locais onde se tem um lado). No seu nome deve conter o código seqüencial (XX) que representa o identificador da sala.
PORTAL_XX_YY_ZZ	Representa um portal em duas dimensões entre duas salas. Deve ser uma aresta contendo apenas dois vértices sendo um em cada extremidade. No seu nome deve conter o código seqüencial (XX), a sala de origem (YY) e a sala de destino (ZZ).
SALA_XX	Representa uma sala em três dimensões no cenário. Deve-se obrigatoriamente sincronizar o MAPA_SALA_XX com a SALA_XX a fim de sobrepor ambos. No seu nome deve conter o código seqüencial (XX).
CHAO	Representa o chão do cenário. Deve obrigatoriamente ser posicionado abaixo dos mapas da sala.

No APÊNDICE A encontra-se uma versão detalhada de como criar os cenários utilizados na aplicação.

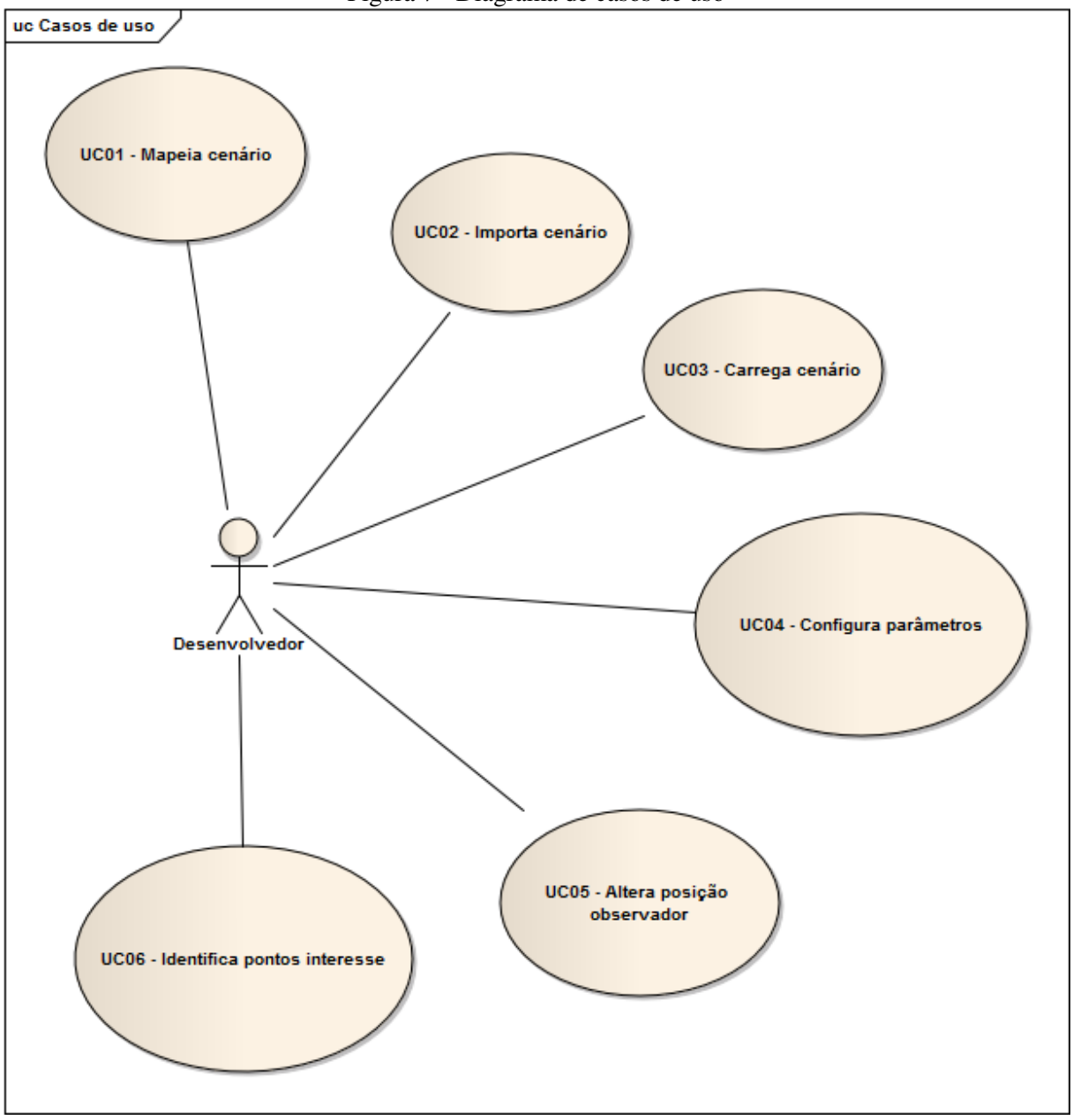
Com todos os nomes de objetos que são interpretados pela aplicação descritos, o desenvolvedor pode criar seus próprios cenários, obedecendo as regras expostas no Quadro 2.

3.2.2 Diagramas de casos de uso

Conforme pode ser observado na Figura 7, a aplicação desenvolvida possui seis casos de uso: Mapeia cenário, Importa cenário, Carrega cenário, Configura aplicação, Altera posição observador e Identifica pontos de interesse. Como ator foi

identificador o `Desenvolvedor`. Este ator utilizará a biblioteca para aplicar os conceitos da técnica de portais em sua aplicação.

Figura 7 - Diagrama de casos de uso



O primeiro caso de uso descreve como deve-se mapear o cenário na ferramenta de modelagem 3D. O segundo caso de uso mostra como o `Desenvolvedor` pode importar o cenário para dentro da aplicação que estiver desenvolvendo. O terceiro caso de uso demonstra como o `Desenvolvedor` deve carregar os cenários na aplicação. O quarto caso de uso descreve como realizar as configurações da biblioteca. O quinto descreve como é o processo de movimentação do observador no cenário. O último caso de uso detalha o funcionamento do algoritmo de portal desenvolvido.

3.2.2.1 Mapeia cenário

O caso de uso `Mapeia cenário`, detalhado no Quadro 3, mostra a forma com que o `Desenvolvedor` deve criar os cenários que serão interpretados pela biblioteca. Pode-se criar cenários contendo diversas salas, portais e pontos de interesse, além de um observador.

Quadro 3 - Detalhamento do caso de uso `Mapear cenário`

UC 01 – Mapear cenário.	
Pré-condições	01) Possuir o *add-on* `MJ3D-Portais` instalado no Blender.
Cenário principal	01) O `Desenvolvedor` cria um novo arquivo com um cubo no Blender. 02) O `Desenvolvedor` altera a dimensão desse novo cubo (`x,y,z`) e separa sua face superior a fim de criar o chão. 03) O `Desenvolvedor` altera o nome do objeto para `CHAO`. 04) O `Desenvolvedor` seleciona o objeto `CHAO` e em modo de edição o subdivide em pequenas partes. 05) O `Desenvolvedor` seleciona as subdivisões que vão fazer parte da sala e separa-as em um novo objeto. 06) O `Desenvolvedor` altera o nome do objeto para `SALA_XX`. 07) Se ainda houveram salas a serem criadas, voltar ao passo 05 do cenário principal. 08) O `Desenvolvedor` cria uma cópia de cada objeto `SALA`. 09) O `Desenvolvedor` adiciona ao nome o prefixo `MAPA_`. 10) O `Desenvolvedor` faz a junção das faces de cada um dos objetos com prefixo `MAPA_SALA_`. 11) O `Desenvolvedor` remove a face desses objetos. 12) Para cada objeto sem face, o `Desenvolvedor` remove os vértices desnecessários deixando apenas os que estão localizados nas extremidades. 13) O `Desenvolvedor` remove a aresta que representa o portal da sala de origem. 14) O `Desenvolvedor` separa a aresta que representa o portal da sala de destino em um novo objeto. 15) O `Desenvolvedor` altera o nome desse novo objeto para `PORTAL_XX_YY_ZZ`. 16) O `Desenvolvedor` adiciona um cubo. 17) O `Desenvolvedor` altera o nome desse cubo para `OBSERVADOR_XX_YY`. 18) O `Desenvolvedor` adiciona um cubo. 19) O `Desenvolvedor` altera o nome desse cubo para `WAYPOINT_XX_YY` 20) Se ainda houver pontos de interesse a serem adicionados, voltar ao passo 18 do cenário principal. 21) O `Desenvolvedor` seleciona o objeto `CHAO`. 22) O `Desenvolvedor` aplica o comando `Extrude` para criar as paredes. 23) No menu de exportação, o `Desenvolvedor` exporta o cenário para a aplicação através do comando `MJ3D-Portais (.map)`.
Pós-condições	O `Desenvolvedor` deve possuir os arquivos com extensão `OBJ, MTL e MAP` gerados pelo Blender.

3.2.2.2 Importa cenário

Após criar o cenário, o `Desenvolvedor` deve importar os arquivos contendo o mapa e informações do modelo 3D na ferramenta de desenvolvimento. O Quadro 4 descreve como funciona a importação de cenários para a aplicação.

Quadro 4 - Detalhamento do caso de uso Importa cenário

UC 02 – Importa cenário.	
Pré-condições	01) Possuir os arquivos do cenário.
Cenário principal	01) O Desenvolvedor adiciona os três arquivos (OBJ, MTL e MAP) que foram gerados pela ferramenta de modelagem na pasta da aplicação que está sendo desenvolvida. 02) Se houverem mais cenários a serem importados, voltar ao passo 01 do cenário principal.
Pós-condições	Os arquivos devem ser transferidos para a pasta da aplicação de desenvolvimento.

3.2.2.3 Carrega cenário

O Desenvolvedor deve carregar os cenários desenvolvidos e previamente importados. Existem três arquivos que compõe o cenário: um arquivo contendo as coordenadas das salas, um arquivo contendo as informações do modelo 3D e por último o arquivo contendo as configurações de iluminação do modelo 3D. O Quadro 5 descreve o carregamento dos cenários.

Quadro 5 - Detalhamento do caso de uso Carrega cenário

UC 03 – Carrega cenário.	
Pré-condições	01) Ter importado os arquivos dos cenários para a ferramenta de desenvolvimento.
Cenário principal	01) O Desenvolvedor chama a biblioteca que trata do modelo 3D passando o nome do arquivo contendo tais informações. 02) O Desenvolvedor chama a biblioteca que trata do mapa (MAP) passando o nome do arquivo contendo as coordenadas das salas.
Pós-condições	O Desenvolvedor deverá configurar os parâmetros da aplicação.

3.2.2.4 Configura aplicação

O Desenvolvedor deve informar as configurações que interferem na execução da biblioteca. Pode-se alterar as configurações que envolvem a câmera do ambiente e os parâmetros da biblioteca de portais, como por exemplo a abertura do *frustum*, a distância do *frustum*, a velocidade de movimentação do observador e o tipo de cenário (2D ou 3D). O detalhamento do caso de uso está no Quadro 6.

Quadro 6 - Detalhamento do caso de uso Configura aplicação

UC 04 – Configura aplicação.	
Cenário principal	01) O Desenvolvedor altera as configurações da câmera. 02) O Desenvolvedor altera a Abertura do campo de visão do observador. 03) O Desenvolvedor altera a Distancia do campo de visão do observador. 04) O Desenvolvedor altera a Velocidade do observador. 05) O Desenvolvedor altera o modo de visualização entre duas e três dimensões.
Pós-condições	O Desenvolvedor deve chamar o método responsável por mover o observador.

3.2.2.5 Altera posição observador

Para que o observador possa percorrer as salas é necessário que ele se movimente entre as diversas salas contidas no cenário. O caso de uso Altera posição observador descreve o

processo de movimentação do observador entre as salas do cenário e o que a biblioteca deve fazer quando isso acontece. O detalhamento do caso de uso está no Quadro 7.

Quadro 7- Detalhamento do caso de uso `Altera posição observador`

UC 05 – Altera posição observador.	
Pré-condições	É necessário que o `Desenvolvedor` já tenha o cenário, com as salas, pontos de interesse e observador previamente carregados. Também é necessário que as configurações tenham sido feitas.
Cenário principal	01) O `Desenvolvedor` chama a biblioteca passando qual deverá ser a nova posição do observador e informando também os dados do campo de visão previamente mapeados. 02) A biblioteca verifica se indo para a nova posição o observador não ira colidir com alguma divisão do ambiente. 03) A biblioteca grava no observador a nova posição solicitada pelo `Desenvolvedor`.
Fluxo alternativo 01	01) No passo 02 do cenário principal pode ocorrer de o observador estar de frente para uma parede, neste caso a biblioteca não permite que a nova posição seja gravada.
Fluxo alternativo 02	01) No passo 02 do cenário principal pode ocorrer de que o observador passe por um portal (finalizando em outra sala). Neste caso a biblioteca deve gravar no observador o identificador da nova sala dele, além da nova posição.
Pós-condições	O `Desenvolvedor` deve chamar a verificação de pontos de interesse para a nova posição do observador.

Fonte: adaptado de Pandini (2012, p. 25).

3.2.2.6 Identifica pontos de interesse

Quando o observador altera sua posição no mapa é necessário que seja verificado se existem pontos de interesse presentes no seu campo de visão. O Quadro 8 descreve como é o funcionamento do algoritmo que será aplicado para verificar se os pontos de interesse estão visíveis para o observador.

Quadro 8 - Detalhamento do caso de uso `Identifica ponto interesse`

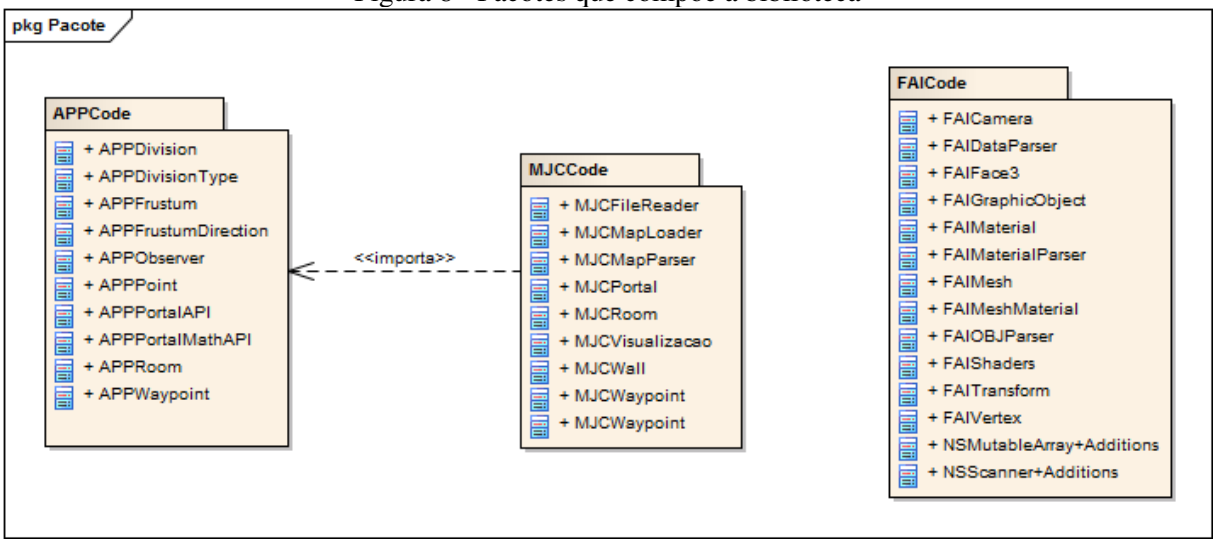
UC 06 – Identifica ponto interesse.	
Pré-condições	É necessário que o `Desenvolvedor` já tenha o cenário, com as salas, pontos de interesse e observador previamente carregados e configurados.
Cenário principal	01) O `Desenvolvedor` chama a biblioteca informando as coordenadas do observador e do campo de visão, a lista de salas do ambiente e também os pontos de interesse. 02) A biblioteca deve verificar quais pontos de interesse que estão na mesma sala do observador e estão em seu campo de visão. 03) A biblioteca deve verificar se há portais no campo de visão do observador.
Fluxo alternativo 01	01) Caso houver algum portal no campo de visão, a biblioteca deve verificar quais pontos de interesse da sala que é vista por este portal estão no campo de visão do observador também.
Pós-condições	A biblioteca deve retornar para o `Desenvolvedor`, em uma estrutura de lista, todos os pontos de interesse que foram identificados como estando no campo de visão do observador.

Fonte: adaptado de Pandini (2012, p. 25).

3.2.3 Diagrama de classes

Os diagramas de classes mostram as principais classes que fazem parte da biblioteca, a forma como elas estão estruturadas e interligadas. São mostrados diagrama de classes dos pacotes `APPCode` e `MJCCode`, além de uma breve explicação sobre a biblioteca utilizada para a utilização de cenários 3D `FAICode`. A Figura 8 mostra os três pacotes, bem como suas dependências e classes que os compõe.

Figura 8 - Pacotes que compõe a biblioteca

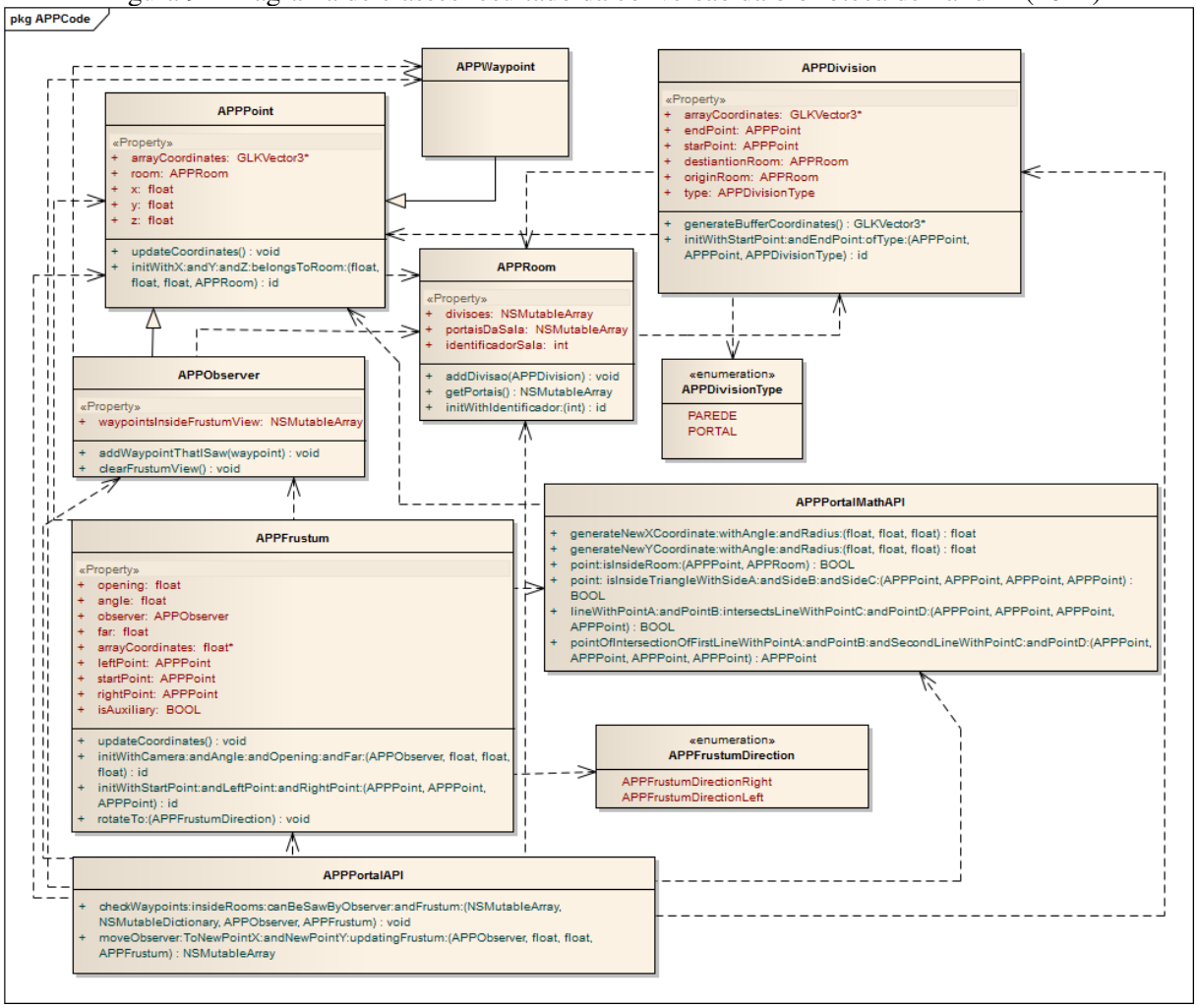


3.2.3.1 Pacote `APPCode`

O pacote `APPCode` abriga todas as classes responsáveis pelo algoritmo de portais. O pacote foi desenvolvido com base na biblioteca criada por Pandini (2012). O novo pacote contém todas as classes já desenvolvidas por ela, assim como adaptações que foram

necessárias ao decorrer do desenvolvimento. O diagrama resultado da conversão e adaptação das classes pode ser visto na Figura 9.

Figura 9 - Diagrama de classes resultado da conversão da biblioteca de Pandini (2012)



No diagrama exposto na Figura 9 tem-se as classes já convertidas e adaptadas, além das dependências entre elas. No diagrama são mostrados os principais métodos e propriedades.

O pacote tem como classe principal a classe `APPPortalAPI`. Esta classe possui o método responsável por movimentar o observador pelo cenário. A classe também possui o método que verifica todos os pontos de interesse que estão sendo vistos pelo observador.

Para realizar os cálculos matemáticos, o pacote utiliza a classe `APPPortalMathAPI`. Esta classe possui os métodos necessários para realizar todos os cálculos envolvidos na biblioteca. Possui os métodos responsáveis por calcular a nova posição do observador, verificar se um ponto se encontra dentro de uma sala, verificar se o ponto está dentro do *frustum* e métodos utilizados em calculos auxiliares.

Cada objeto do tipo `APPRoom` representa uma sala. Uma sala é composta por várias divisões (objetos do tipo `APPDivision`), que representam portais e paredes (enumeração

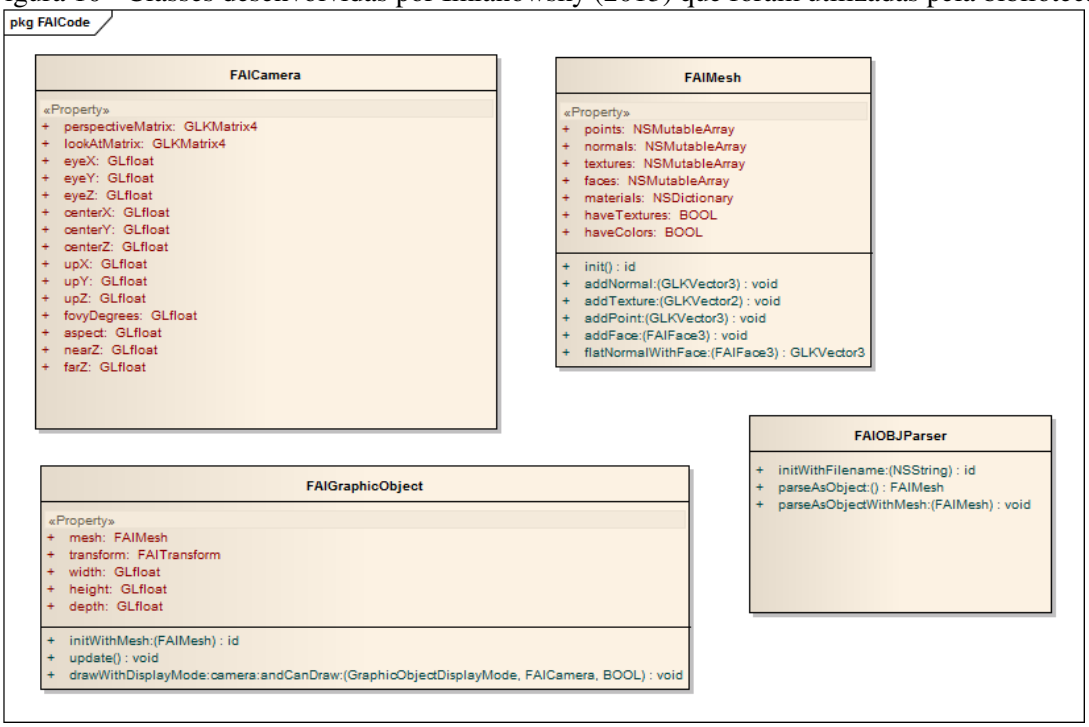
`APPDivisionType`). Uma sala pode armazenar vários pontos de interesse (`APPWaypoints`) que podem ser vistos pelo observador.

A classe `APPObserver` representa o observador no mapa. O observador percorre o mapa em busca de pontos de interesse que estão no seu campo de visão. O *frustum* (campo de visão) do observador é representado pela classe `APPFrustum`. Tem a opção de ser movimentado para o lado direito ou esquerdo (enumeração `APPFrustumDirection`) para por exemplo, similuar uma curva ou rotação do observador.

3.2.3.2 Pacote `FAICode`

Para a utilização de cenários em três dimensões optou-se pela utilização da biblioteca desenvolida por Imianowsky (2013). A biblioteca em questão já possui as classes necessárias para realizar a importação de arquivos no formato `OBJ` que contém os modelos 3D exportados por ferramentas de modelegam, bem como a manipulação dos objetos em tela. A Figura 10 mostra as classes do pacote que foram utilizadas explicitamente pela biblioteca desenvolvida.

Figura 10 - Classes desenvolvidas por Imianowsky (2013) que foram utilizadas pela biblioteca



A principal classe mostrada na Figura 10 é a `FAIGraphicObject`. Esta classe recebe um modelo 3D já convertido e faz a renderização na tela através do método `drawWithDisplayMode:camera:andCanDraw:`. Esse método tem como parâmetros o modo que vai ser renderizado (podendo ser renderizado objetos contendo texturas, sem textura, malha de pontos e apenas os pontos), a câmera que vai ser utilizada no ambiente onde será mostrado o objeto e um parâmetro informando se deve ou não renderizar o objeto na tela. Esse

último parâmetro (`canDraw`) foi criado para este pacote funcionar em conjunto com os demais pacotes desenvolvidas neste trabalho.

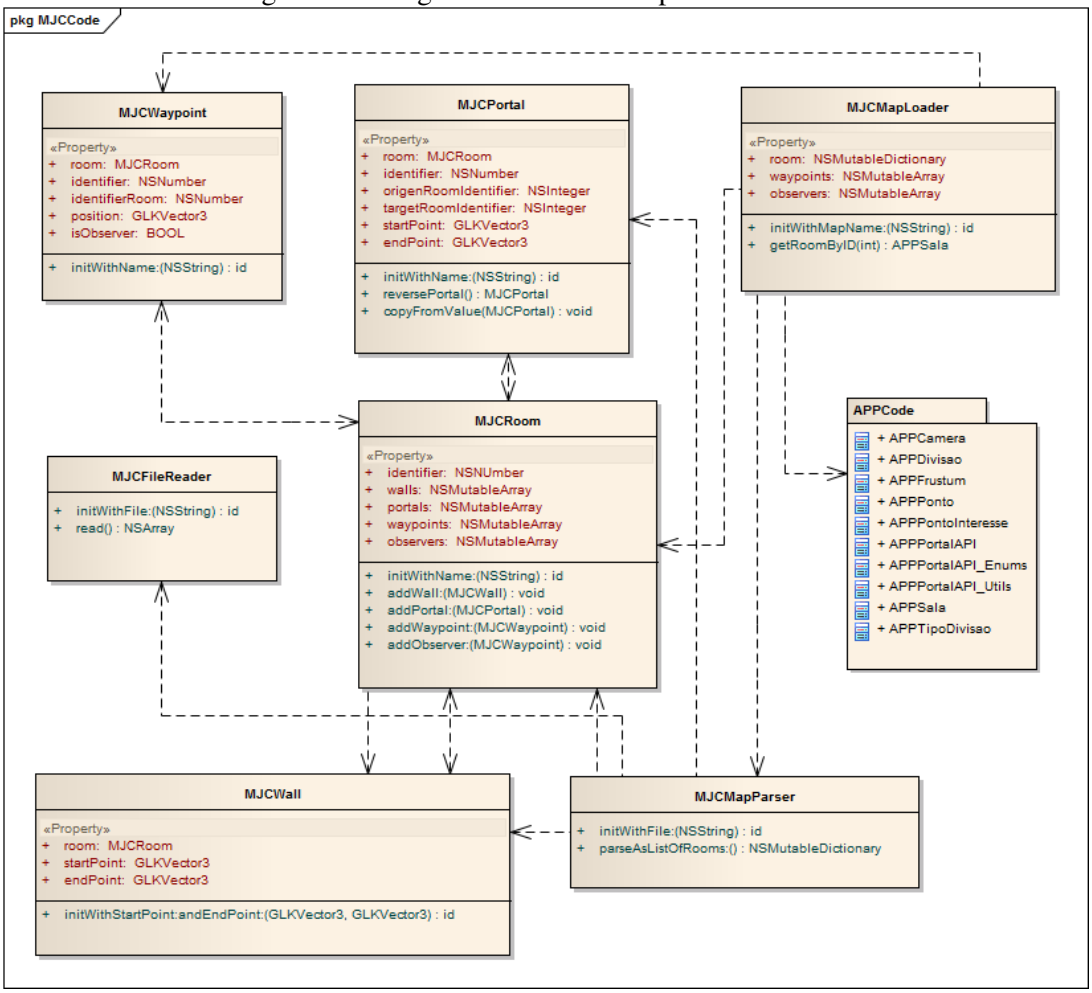
Para que seja possível renderizar o objeto na tela, é necessário que ele seja convertido e armazenado na classe `FAIMesh`. Esta classe dispõe de métodos e propriedades que armazenam listas das informações que foram importadas dos arquivos do modelo 3D.

Por fim, a classe `FAIOBJParser` interpreta o arquivo contendo o modelo 3D e armazena as informações em um objeto do tipo `FAIMesh`. A classe recebe o nome do arquivo contendo o modelo 3D por parâmetro através do método `initWithFileName:` e retorna o objeto `FAIMesh` através do método `parseAsObject`. Essa classe sofreu várias alterações para funcionar em conjunto com os outros pacotes.

3.2.3.3 Pacote `MJCCode`

O pacote `MJCCode` é responsável por interpretar e importar os cenários previamente desenvolvidos na ferramenta de modelagem. Esse pacote interpreta o mapa dos cenários (arquivos com extensão `MAP`) e os disponibiliza em uma estrutura de listas para serem utilizados pelos demais pacotes. Na Figura 11 é mostrado o diagrama de classes do pacote.

Figura 11 - Diagrama de classes do pacote `MJCCode`



A classe principal neste contexto é a classe `MJCMapParser`. Esta classe tem como função converter o arquivo `MAP` para um formato composto por listas de objetos. A classe recebe como parâmetro em seu construtor o nome do arquivo que contém as informações do mapa a ser carregado.

Ao solicitar as informações através do método `parseAsListOfRooms`, executa a varredura no arquivo buscando as informações pertinentes. A classe então cria estruturas de listas contendo objetos `MJCRoom`, `MJCWall`, `MJCPortal` e `MJCWaypoint` que representam salas, paredes das salas, portais e pontos de interesse/observador respectivamente. O Quadro 9 mostra o exemplo de um arquivo que a classe `MJCMapParser` interpreta.

Quadro 9 - Modelo de arquivo interpretado pela classe `MJCMapParser`

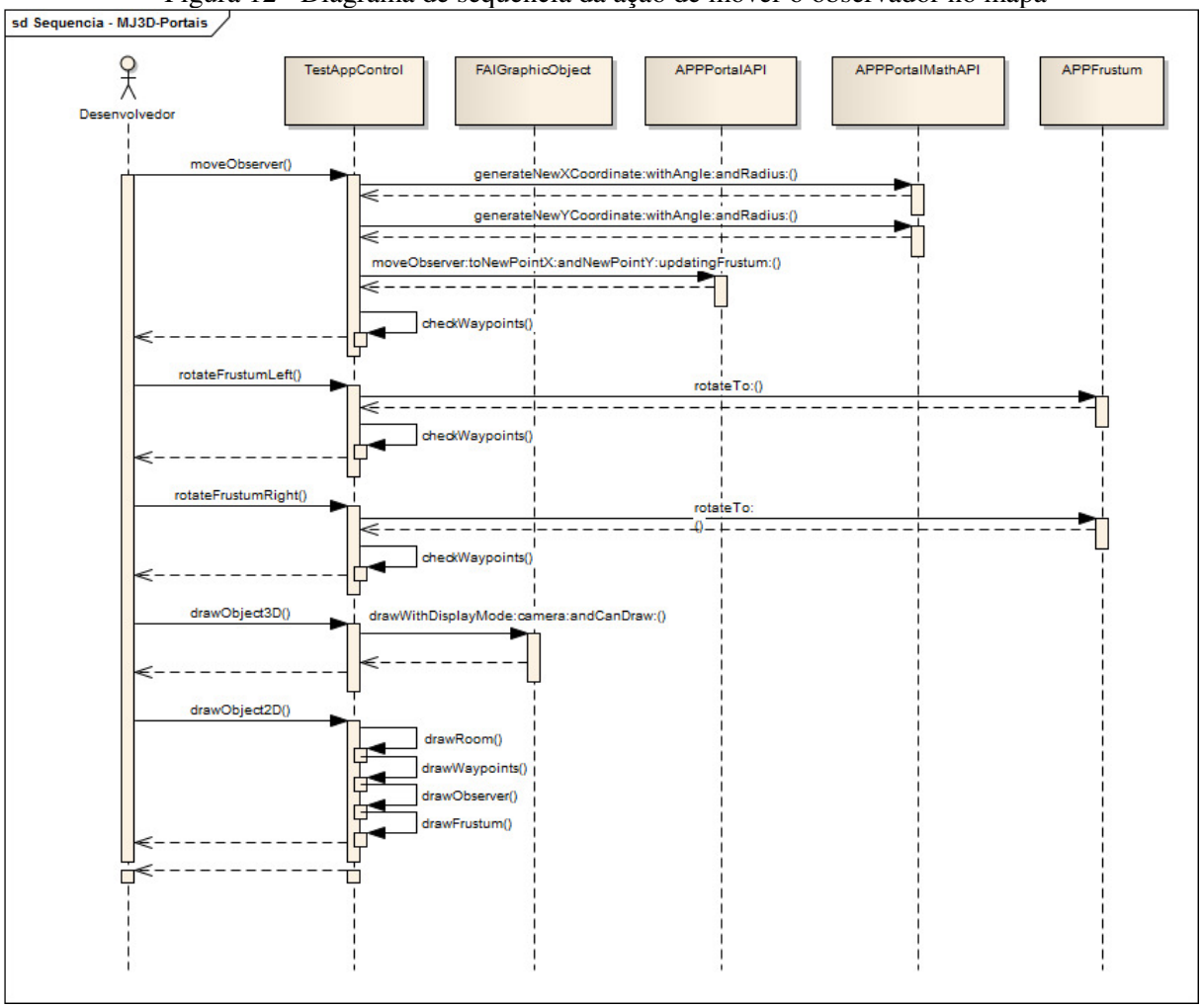
```
o MAPA_SALA_01
e -0.000000 1.874998 0.050000 -0.000001 0.749999 0.050000
e -0.000001 0.749999 0.050000 -1.500000 0.749999 0.050000
e -5.624997 0.750001 0.050000 -3.749999 0.750000 0.050000
e -5.624997 5.624997 0.050000 -5.624997 0.750001 0.050000
e 0.000001 5.624996 0.050000 0.000001 4.499997 0.050000
e -5.624997 5.624997 0.050000 0.000001 5.624996 0.050000

o MAPA_SALA_02
e 0.375001 4.499997 0.050000 0.000001 4.499997 0.050000
e -0.000000 1.874998 0.050000 0.375000 1.874998 0.050000

o PORTAL_01_01_02
e -3.749999 0.750000 0.050000 -1.500000 0.749999 0.050000

o WAYPOINT_01_01
v -1.498443 3.765211 0.200000

o OBSERVADOR_01_01
v -3.015483 3.765211 0.200000
```

Para que o desenvolvedor utilize as informações juntamente com o pacote `APPCode`, criou-se a classe `MJCMapLoader`. Ela recebe e interpreta as informações fornecidas pela classe `MJCMapParser`, transformando-as em informações que são reconhecidas pelo pacote `APPCode`. A classe recebe como parâmetro em seu construtor o nome do arquivo a ser carregado, converte o arquivo através da classe `MJCMapParser` e então cria uma estrutura de listas contendo os objetos que são reconhecidos pelo pacote da biblioteca de portais.

3.2.4 Diagrama de sequência

Esta seção apresenta um diagrama de sequência com os passos executados pelo método responsável por movimentar o observador no mapa. A Figura 12 apresenta este diagrama.

Figura 12 - Diagrama de sequência da ação de mover o observador no mapa



Fonte: adaptado de Pandini (2012, p. 31).

No diagrama é apresentada a principal execução da biblioteca. O `Desenvolvedor` executa o método responsável por mover o observador. A biblioteca então calcula a nova posição, verifica se o observador pode ir para esse novo local e faz a movimentação. Após isso é verificado os pontos de interesse que estão no seu campo de visão. É possível então rotacionar o observador para a direita ou esquerda. Após rotacionar, é verificado novamente se existem pontos de interesse em seu campo de visão. Após realizar todos os cálculos e movimentações, o `Desenvolvedor` chama os métodos responsáveis por desenhar os modelos na tela.

3.3 IMPLEMENTAÇÃO

Nesta seção são apresentadas informações sobre as técnicas e ferramentas utilizadas para a implementação da biblioteca. Também é apresentado o *add-on* desenvolvido para a exportação de mapas, a forma como é feita a conversão dos arquivos `MAP`, as alterações feitas na biblioteca de portais originalmente desenvolvida por Pandini (2012) e as adaptações

necessárias para a utilização da biblioteca desenvolvida por Imianowsky (2013). Por fim é descrita a operacionalidade da biblioteca.

3.3.1 Técnicas e ferramentas utilizadas

Para o desenvolvimento da biblioteca foram utilizadas duas linguagens de programação. A linguagem de programação Python foi utilizada na criação do *add-on* responsável pela geração do arquivo `MAP`. A implementação foi feita no aplicativo *Code Runner 1.3* desenvolvido por Nikolai Krill. Para a criação e alteração das bibliotecas envolvidas foi utilizada a linguagem de programação Objective-C. O ambiente de desenvolvimento utilizado nesta etapa foi o Xcode 4.6.1, disponibilizado pela Apple. Para a geração das imagens gráficas no espaço 3D foi utilizado a biblioteca gráfica OPENGL ES 2.0 juntamente com a biblioteca desenvolvida por Imianowsky (2013).

3.3.2 Exportação de mapas

A exportação de mapas é a parte da biblioteca responsável por gerar os arquivos contendo as informações referentes ao mapa onde o algoritmo de portais será reproduzido. O mapa que será exportado é projetado pelo usuário na ferramenta de modelagem Blender. O *add-on* desenvolvido para realizar esta parte da tarefa segue dois passos principais para exportar o mapa.

O primeiro passo é o reconhecimento dos objetos que vão compor o mapa da aplicação. Para isso é utilizado o método `exportar_arestas(self)`. Este método gera um arquivo `MAP` contendo a relação de todos os objetos e coordenadas que compõe o mapa exportado. O algoritmo utilizado para gerar este arquivo pode ser visualizado no Quadro 10.

Quadro 10 - Algoritmo que cria o arquivo MAP contendo as informações do mapa

```
01 def exportar_arestas(self):
02   # altera o modo no blender
03   bpy.ops.object.mode_set(mode='OBJECT')
04   # abre um arquivo para gravar as informacoes
05   file = open(self.filepath, 'w')
06   # imprime o cabecalho do arquivo
07   file.write("# Blender v%s OBJ File: '%s'\n" % (bpy.app.version_string,
08               os.path.basename(self.filepath).replace(".map", ".obj")))
09   file.write("# Desenvolvido por Mateus Junior Cassaniga\n")
10   file.write("# Esse script contem todos os objetos que representam a
11               planta baixa do modelo 3D\n")
12   # percorre todos os objetos
13   for obj in bpy.data.objects:
14     # exporta apenas objetos do tipo MESH
15     if obj.type == 'MESH':
16       # se for um objeto que representa um MAPA ou um PORTAL
17       if obj.name.upper().startswith('MAPA_SALA_')
18          or obj.name.upper().startswith('PORTAL_'):
19          # pega todos os edges (arestas)
20          edges = obj.data.edges
21          # imprime todos os edges desse objeto
22          file.write("\no %s\n" % (obj.name.upper()))
23          for edge in edges:
24            v0 = obj.data.vertices[edge.vertices[0]].co * obj.matrix_world
25            v1 = obj.data.vertices[edge.vertices[1]].co * obj.matrix_world
26            file.write("e %f %f %f %f %f %f\n" %
27            (round(v0.x, 6), round(v0.y, 6), round(v0.z, 6),
28             round(v1.x, 6), round(v1.y, 6), round(v1.z, 6)))
29       # se for um objeto que representa um WAYPOINT ou um OBSERVADOR
30       elif obj.name.upper().startswith('WAYPOINT_')
31            or obj.name.upper().startswith('OBSERVADOR_'):
32         # imprime o nome e a localizacao
33         v0 = obj.location
34         file.write("\no %s\n" % (obj.name.upper()))
35         file.write("v %f %f %f\n" %
36         (round(v0.x, 6), round(v0.y, 6), round(v0.z, 6)))
37 file.flush()
38 file.close()
```

São percorridos os objetos que foram criados previamente no Blender. Logo em seguida é verificado se o objeto em questão é do tipo MESH. No Blender é possível criar objetos de vários tipos, como por exemplo textos, curvas e superfícies. Para os arquivos MAP, os únicos objetos que realmente importam são os objetos do tipo MESH, por serem uma coleção de vértices, arestas e faces. Para identificar se são objetos que representam uma sala ou portal é necessário verificar seu nome fazendo comparações para verificar se o objeto em questão possui um nome com prefixo MAPA_SALA_ ou PORTAL_.

A linha 22 grava no arquivo MAP o nome do objeto que está sendo interpretado, para que seja identificado que as coordenadas abaixo deste nome fazem parte deste objeto. Objetos que representam uma sala ou portal possuem uma ou mais arestas, por terem um ponto de origem e outro de destino.

Para isso é necessário percorrer cada aresta do objeto e pegar seu ponto de origem e seu ponto de destino. Logo em seguida é gravado uma linha no arquivo contendo tais informações.

Para objetos que representam um ponto de interesse ou observador, a única coordenada que importa é a sua localização no mapa. Nas linhas 30 e 31 são verificados se o objeto atual corresponde a um desses dois, verificando se seu prefixo é igual a WAYPOINT_ ou OBSERVADOR_. Para cada um desses objetos, são gravados seu nome e sua coordenada no arquivo.

O segundo passo da exportação dos mapas é exportar os arquivos que representam o modelo 3D e suas configurações de cores e iluminação. O Blender por padrão possui um método para gerar esses arquivos, bastando apenas chamá-lo passando os parâmetros de configuração. O Quadro 11 mostra o algoritmo utilizado para exportar tais arquivos.

Quadro 11 - Método utilizado para criar os arquivos com informação do modelo 3D

```
01  bpy.ops.export_scene.obj(filepath=self.filepath.replace(".map", ".obj"),
02                           axis_forward='Y',
03                            axis_up='Z',
04                            use_mesh_modifiers=True,
05                            use_normals=True,
06                            use_uvs=True,
07                            use_materials=True,
08                            use_triangles=True,
09                            use_blen_objects=True)
```

Para realizar essa função é chamado o método padrão do Blender para exportar arquivos OBJ contendo as informações do modelo 3D. O arquivo contendo as informações de cores e iluminação (arquivo MTL) é criado pelo mesmo método, internamente. É passado como parâmetro o nome do arquivo que vai ser gerado. Por padrão, esse nome é o mesmo nome do arquivo MAP, porém com a extensão diferente. Os demais parâmetros são configurações de exportação relacionados ao formato que será gerado.

3.3.3 Algoritmo de portais

O pacote APPCode é responsável por conter as classes do algoritmo de portais. Por se tratar de uma conversão da biblioteca previamente desenvolvida por Pandini (2012), manteve-se a mesma estrutura de classes, realizando as alterações necessárias em cada uma delas.

Duas das principais classes do pacote sofreram várias modificações. A primeira classe, APPPortalAPIMath contém todos os métodos que realizam cálculos. Nesta classe, manteve-se apenas três métodos desenvolvidos por Pandini (2012). São eles: método que calcula a próxima coordenada X, método que calcula a próxima coordenada Y e um método que calcula o ponto de intersecção de duas linhas. O Quadro 12 mostra os três algoritmos.

Quadro 12 - Métodos responsáveis por gerar coordenadas e calcular ponto de interseção de duas linhas

```
01 // Calcula uma nova coordenada X baseando-se no ângulo e raio
02 +(float)generateNewXCoordinate:(float)currentX
03                       withAngle:(float)angle
04                       andRadius:(float)radius {
05   return (float) (currentX + (radius * cos(M_PI * angle / 180.0)));
06 }
07
08 // Calcula uma nova coordenada Y baseando-se no ângulo e raio
09 +(float)generateNewYCoordinate:(float)currentY
10                       withAngle:(float)angle
11                       andRadius:(float)radius {
12   return (float) (currentY + (radius * sin(M_PI * angle / 180.0)));
13 }
14
15 // Retorna o ponto de interseção de duas linhas
16 +(APPPoint*)pointOfIntersectionOfFirstLineWithPointA:(APPPoint*)k
17                                             andPointB:(APPPoint*)l
18                                 andSecondLineWithPointC:(APPPoint*)m
19                                             andPointD:(APPPoint*)n {
20   float det = (n.x - m.x) * (l.y - k.y) - (n.y - m.y) * (l.x - k.x);
21   float s = ((n.x - m.x) * (m.y - k.y) - (n.y - m.y) * (m.x - k.x)) / det;
22   APPPoint* ponto = [[APPPoint alloc] initWithX:(k.x + (l.x - k.x) * s)
23                                             andY:(k.y + (l.y - k.y) * s)
24                                             andZ:k.z belongsToRoom:nil];
25   return ponto;
26 }
```

Acima estão os métodos desenvolvidos por Pandini (2012). Nas linhas `02-04` e `09-11` estão listados os métodos que geram novas coordenadas e por último o método utilizado para descobrir o ponto de interseção de duas linhas que se cruzam.

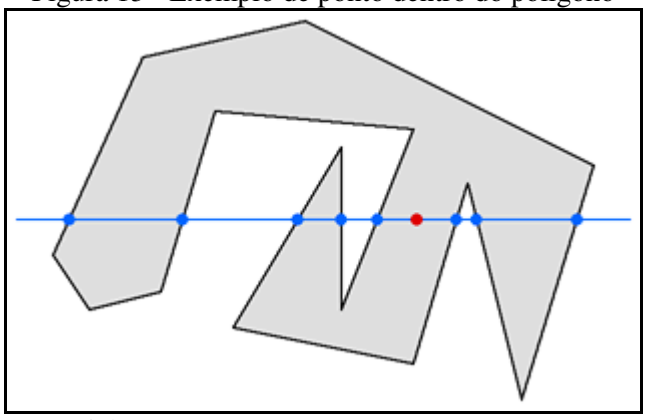
Além desses métodos convertidos da biblioteca desenvolvida por Pandini (2012), foram criados outros métodos para cálculos específicos, que apesar de já terem sido desenvolvidos por Pandini (2012), foram construídos de formas totalmente diferentes. Foram desenvolvidos métodos para verificar se o observador está dentro de uma sala, verificar se um ponto está dentro do campo de visão e um método para verificar se dois segmentos de reta se cruzam. O primeiro método descrito pode ser visto no Quadro 13.

Quadro 13 - Método responsável por verificar se um ponto está dentro de uma sala

```
01 // Verifica se um ponto está dentro de uma sala
02 +(BOOL)point:(APPPoint *)point isInsideRoom:(APPRoom *)room {
03   int sides = room.coordinatesLenght;
04   int i, j = sides - 1;
05   BOOL inside = NO;
06
07   for (i = 0; i < sides; i++) {
08     if (((room.yCoordinates[i] < point.y && room.yCoordinates[j] >= point.y)
09       || (room.yCoordinates[j] < point.y && room.yCoordinates[i] >= point.y))
10       && (room.xCoordinates[i] <= point.x || room.xCoordinates[j] <= point.x)) {
11       if ((room.xCoordinates[i] + (point.y - room.yCoordinates[i]) /
12           (room.yCoordinates[j] - room.yCoordinates[i]) *
13           (room.xCoordinates[j] - room.xCoordinates[i]) < point.x)) {
14         inside = !inside;
15       }
16     }
17     j = i;
18   }
19   return inside;
20 }
```

O método acima foi desenvolvido baseando-se no algoritmo desenvolvido por Finley (2007). O algoritmo compara cada lado do polígono com a coordenada `Y` do ponto que está sendo testado. Se a quantidade de vezes que a linha imaginária cruzar os lados do polígono for ímpar, significa que o ponto está dentro do polígono. Se a quantidade de vezes for par, significa que o ponto está localizado no lado de fora (FINLEY, 2007). A Figura 13 mostra um exemplo onde o ponto está dentro do polígono.

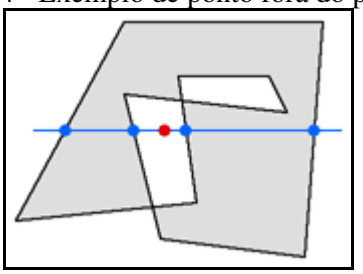
Figura 13 - Exemplo de ponto dentro do polígono



Fonte: Finley (2007).

Na imagem acima, a área interna do polígono está preenchida com a cor cinza. O ponto que está sendo testado é representado pela cor vermelha e a linha imaginária pela cor azul. No exemplo, a linha imaginária cruza cinco vezes o lado esquerdo do polígono e três vezes o lado direito. Logo, para ambas as direções, a quantidade de vezes que a linha cruza um lado do polígono é impar, desta forma é considerado que o ponto está localizado dentro do polígono. A Figura 14 mostra um exemplo onde o ponto está fora do polígono.

Figura 14 - Exemplo de ponto fora do polígono



Fonte: Finley (2007).

No exemplo acima, o ponto se encontra fora do polígono. A linha imaginária (cor azul) cruza dois lados do polígono em ambas as direções.

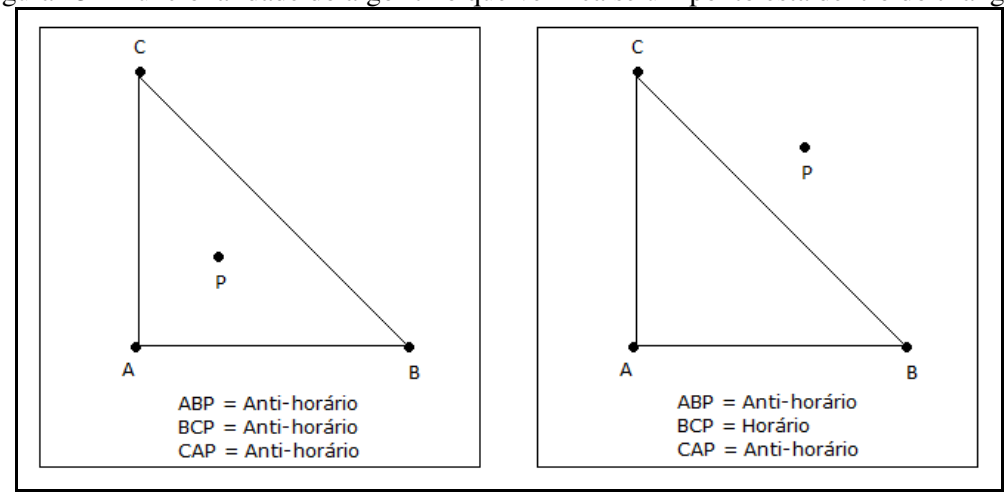
O segundo algoritmo desenvolvido é responsável por determinar se um ponto está dentro de um triângulo. O algoritmo foi desenvolvido como base nas explicações de Blackpawn (2011) e Boe (2006). O Quadro 14 mostra o algoritmo.

Quadro 14 - Métodos responsáveis por verificar se um ponto está dentro do triângulo

```
01 // Verifica se um ponto está dentro do triângulo
02 +(BOOL)point:(APPPoint *)point isInsideTriangleWithSideA:(APPPoint *)p1
03                                                 andSideB:(APPPoint *)p2
04                                                 andSideC:(APPPoint *)p3 {
05   double a = [self side:p1 andPontoB:p2 andPontoC:point];
06   double b = [self side:p2 andPontoB:p3 andPontoC:point];
07   double c = [self side:p3 andPontoB:p1 andPontoC:point];
08   return a == b && b == c;
09 }
10
11 // Verifica a direção da ligação dos pontos
12 +(double)side:(APPPoint*)a andPontoB:(APPPoint*)b andPontoC:(APPPoint*)c {
13   double sideCalc = (b.x - a.x) * (c.y - a.y) - (c.x - a.x) * (b.y - a.y);
14   if (sideCalc > 0) {
15     return 1;
16   }
17   else if (sideCalc < 0) {
18     return -1;
19   }
20   else {
21   return 0;
22   }
23 }
```

Para o algoritmo determinar se o ponto está dentro ou fora do triângulo é calculada a orientação da ligação dos pontos (Figura 15). Por exemplo, um triângulo possui três vertices `A`, `B` e `C`. O ponto a ser testado é representado pela letra `P`. Para um ponto estar dentro de um triângulo, é necessário que as ligações dos pontos `ABP`, `BCP` e `CAP` possuam a mesma orientação (horária ou anti-horária).

Figura 15 – Funcionalidade do algoritmo que verifica se um ponto está dentro do triângulo



Na imagem acima, observa-se que no quadro do lado esquerdo o ponto está dentro do triângulo. Para esse modelo, todas as três ligações (`ABP`, `BCP` e `CAP`) possuem uma ligação anti-horária. Para o modelo do lado direito, onde o ponto não está contido dentro do triângulo, observa-se que um dos resultados das ligações `ABP`, `BCP` e `CAP` está diferente das demais.

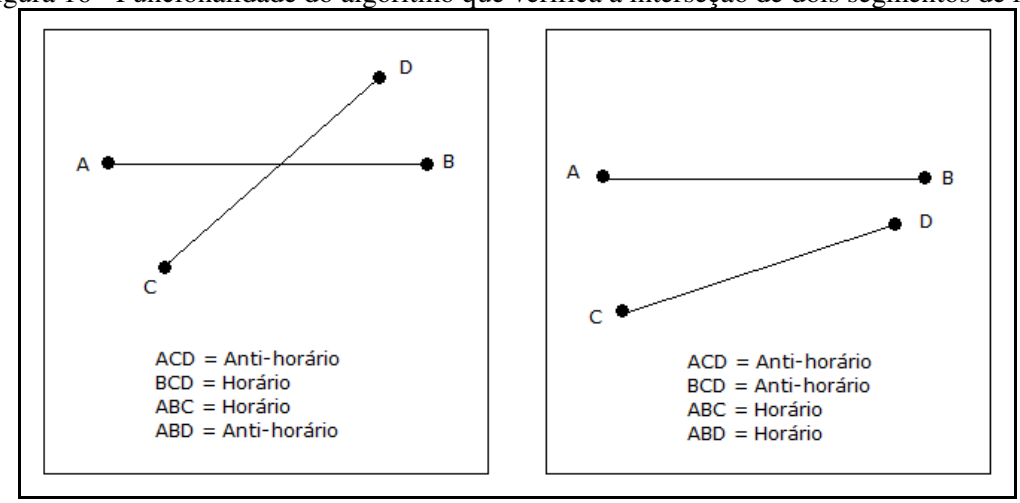
O último algoritmo desenvolvido para a classe `APPPortalAPIMath` é responsável por determinar se dois segmentos de retas se cruzam. O algoritmo foi desenvolvido baseando-se na implementação desenvolvida por Boe (2006). O Quadro 15 mostra o código.

Quadro 15 - Métodos responsáveis por verificar a interseção de dois segmentos de reta

```
01 // Verifica se os pontos estão no sentido anti-horário
02 +(BOOL)ccw:(APPPoint *)A b:(APPPoint*)B c:(APPPoint *)C {
03   return (C.y-A.y)*(B.x-A.x) > (B.y-A.y)*(C.x-A.x);
04 }
05
06 // Verifica se existe interseção entre duas linhas
07 +(BOOL)     lineWithPointA:(APPPoint *)a
08                  andPointB:(APPPoint *)b
09  intersectsLineWithPointC:(APPPoint *)c
10                  andPointD:(APPPoint *)d {
11   return [self ccw:a b:c c:d] != [self ccw:b b:c c:d]
12       && [self ccw:a b:b c:c] != [self ccw:a b:b c:d];
13 }
```

Para determinar se dois segmentos de retas se cruzam é calculado a orientação (horária ou anti-horária) da ligação de três dos quatro pontos que compoem as duas linhas. Por exemplo, tem-se duas linhas sendo que a primeira começa no ponto `A` e termina no ponto `B` e a segunda começa no ponto `C` e termina no ponto `D`. Para determinar se ambas se cruzam é verificado se a ligação das linhas `ACD` possui uma orientação inversa a ligação `BCD` e a ligação `ABC` possui uma orientação inversa a ligação `ABD`. A Figura 16 demonstra a funcionalidade.

Figura 16 - Funcionalidade do algoritmo que verifica a interseção de dois segmentos de reta



No exemplo acima, o quadro do lado esquero possui dois segmentos de retas que se cruzam. O resultado da ligação dos pontos `ACD` é inverso ao `BCD` e o resultado da ligação dos pontos `ABC` é inverso ao `ABD`. No quadro do lado direito, não há a interseção das linhas. O resultado da ligação dos pontos `ACD` é igual ao `BCD` e o resultado da ligação dos pontos `ABC` é igual ao `ABD`.

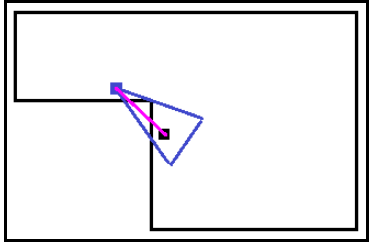
A segunda classe é a `APPPortalAPI`. Esta classe possui os métodos responsáveis por verificar os pontos de interesse que estão em seu campo de visão e movimentar o observador. Parte do método responsável por verificar os pontos de interesse que estão sendo vistos pelo observador pode ser visto no Quadro 16.

Quadro 16 - Trecho de código responsável por verificar os pontos de interesse estão dentro do campo de visão do observador

```
01 // Para cada ponto de interesse
02 for (APPWaypoint *wp in waypointsList) {
03   // Verifico se ele está no quarto atual
04   if (wp.room.roomIdentifier == currentRoom.roomIdentifier) {
05     // Verifico se ele está no campo de visão do observador
06     if ([APPPortalMathAPI point:wp
07       isInsideTriangleWithSideA:startPoint
08                       andSideB:rightPoint
09                       andSideC:leftPoint]) {
10       BOOL frontWall = NO;
11       // Para cada quarto visitado pelo frustum
12       for (APPRoom *testRoom in visitedRoomsList) {
13         BOOL canBreak = NO;
14         // Para cada parede do quarto
15         for (APPDivision *divisao in testRoom.divisions) {
16           if (divisao.type == APPDivisionTypeWall) {
17             // Verifico se ela não está interferindo na visualização
18             // do ponto de interesse
19             frontWall = [APPPortalMathAPI lineWithPointA:observer
20                                                 andPointB:wp
21                                 intersectsLineWithPointC:divisao.startPoint
22                                               andPointD:divisao.endPoint];
23             if (frontWall) {
24               canBreak = YES;
25               break;
26             }
27           }
28         }
29         if (canBreak) {
30           break;
31         }
32       }
33       if (!frontWall) {
34         [observer addWaypointThatISaw:wp];
35       }
36     }
37   }
38 }
```

Primeiramente é feita a verificação dos pontos de interesse presentes na sala atual que estão dentro do campo de visão do observador (linhas `06-09`). Para esses pontos, é criada uma linha imaginária entre o observador e cada um desses pontos de interesse (linhas `19-22`). Com isso é verificado se alguma parede está cruzando essa linha imaginária. Se alguma parede cruzar a linha imaginária, significa que o ponto de interesse está dentro do campo de visão mas não está sendo visto pelo observador (uma parede impede essa visão). Caso contrário, é adicionado na lista de pontos vistos (linha `34`). A Figura 17 mostra um exemplo.

Figura 17 - Exemplo onde o ponto de interesse está no campo de visão, mas não está sendo visto



Na imagem acima, paredes são representadas pelas linhas pretas, o observador e seu campo de visão são representados pela cor azul. O ponto de interesse é representado pela cor preta. A linha imaginária criada é representada pela cor rosa. Na imagem, o ponto de interesse está na mesma sala que o observador, está no campo de visão do observador mas não está sendo visto porque existem duas paredes na sua frente.

Após verificar todos os pontos de interesse, é necessário testar se o campo de visão está cruzando algum portal. O trecho de código do Quadro 17 mostra de forma simplificada esta parte do método.

Quadro 17 - Trecho de código responsável por criar os campos de visão auxiliares

```
01 // Para cada portal da sala
02 for (APPDivision *portal in [currentRoom listOfPortals]) {
03   // ...
04
05   // Caso apenas a linha esquerda do campo de visão esteja
06   // passando pelo portal
07   if (leftLineOfFrustumIntersectsPortal
08   && !rightLineOfFrustumIntersectsPortal) {
09     // Calculo a nova linha do lado direito do frustum
10     // Crio um frustum auxiliar
11   }
12
13   // Caso apenas a linha direita do campo de visão esteja passando pelo portal
14   if (rightLineOfFrustumIntersectsPortal
15   && !leftLineOfFrustumIntersectsPortal) {
16     // Calculo a nova linha do lado esquerdo do frustum
17     // Crio um frustum auxiliar
18   }
19
20   // Caso as duas linhas do campo de visão passem pelo portal
21   if (leftLineOfFrustumIntersectsPortal
22    && rightLineOfFrustumIntersectsPortal) {
23     // Crio um frustum auxiliar com o mesmo tamanho do original
24   }
25
26   // Se apenas a linha final do campo de visão cruze o portal
27   if (endOfFrustumIntersectsPortal) {
28     // Calculo qual lado do frustum deve ser alterado
29     // Crio um novo frustum auxiliar
30   }
31
32   // Caso o portal esteja dentro do campo de visão
33   if (firstVertexOfPortalIsInsideFrustum
34    && secondVertexOfPortalIsInsideFrustum) {
35     // Crio um novo frustum com o tamanho do portal
36   }
37
38   // caso tenha sido criado um novo frustum auxiliar
39   if (newLeftPoint && newRightPoint) {
40     // adiciono na lista de frustum auxiliares chamo esse mesmo método
41     // para verificar recursivamente (até não enxergar mais portais)
42   }
43 }
```

Após verificar todos os pontos de interesse, é necessário testar se o campo de visão está cruzando algum portal. O trecho de código do quadro acima faz as seguintes verificações:

a) se a linha esquerda do campo de visão está ultrapassando um portal (linhas `07` e

08);

b) se a linha direita do campo de visão está ultrapassando um portal (linhas 14 e 15);

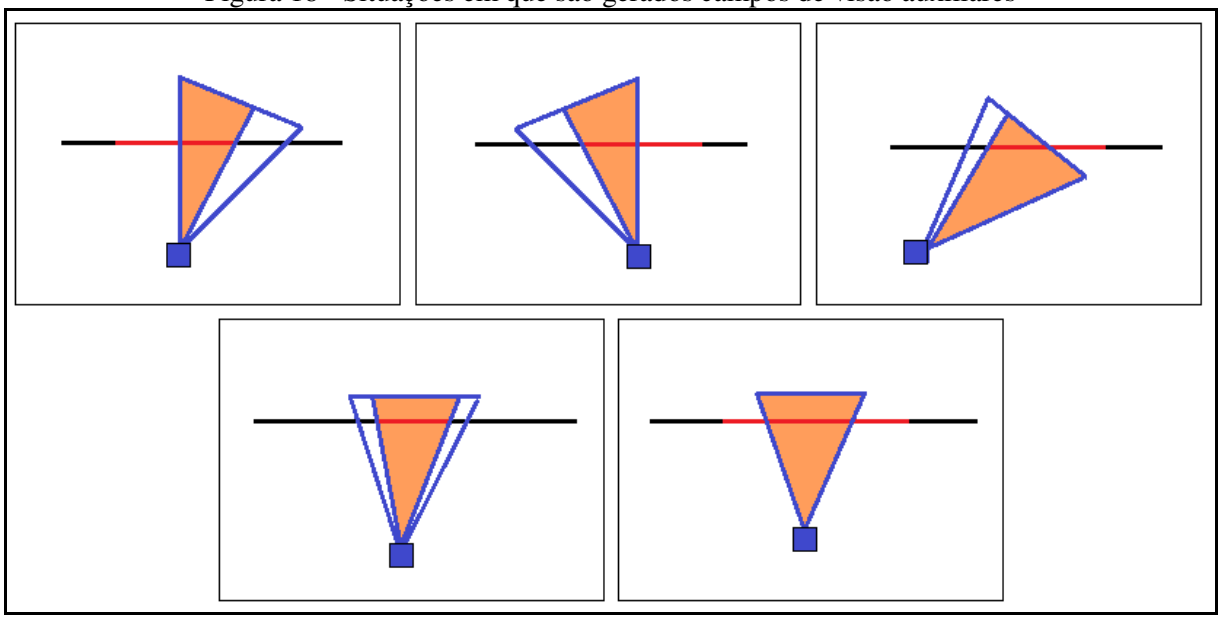
c) se as duas linhas do campo de visão (esquerda e direita) estão ultrapassando o portal (linhas 21 e 22);

d) se apenas a linha final do campo de visão está ultrapassando o portal (linha 27);

e) se o portal está dentro do campo de visão (linhas 33 e 34).

A Figura 18 ilustra as situações que estão sendo tratadas pelo algoritmo.

Figura 18 - Situações em que são gerados campos de visão auxiliares



Na figura acima, as paredes são representadas pelas linhas pretas. O portal é representado pela linha vermelha. O observador é representado pelo ponto azul. O campo de visão principal é representados pelo triângulo de linhas azuis e os novos campos de visão (auxiliares) são representados pelos triângulos preenchidos com a cor laranja. Um campo de visão auxiliar nunca será maior que o campo de visão principal.

Para cada campo de visão auxiliar criado, o mesmo método é executado passando-se a sala que está sendo vista pelo campo de visão, até que mais nenhum portal esteja sendo visto pelo observador, num processo recursivo.

Além do método responsável por verificar os pontos de interesse e criar campos de visão auxiliar, a classe `APPPortalAPI` possui um método responsável por movimentar o observador. O Quadro 18 mostra o código fonte do método.

Quadro 18 - Método responsável por fazer a movimentação do observador

```
01 // Movimenta o observador
02 -(void)moveObserver:(APPObserver*)observer
03         ToNewPointX:(float)newPointX
04        andNewPointY:(float)newPointY
05     updatingFrustum:(APPFrustum*)frustum {
06   BOOL canMove = true;
07
08   APPPoint *newPoint = [[APPPoint alloc] initWithX:newPointX
09                                               andY:newPointY
10                                               andZ:0
11                                     belongsToRoom:nil];
12   BOOL found = NO;
13   // Se o ponto está dentro da sala
14   if (![APPPortalMathAPI point:newPoint isInsideRoom:observer.room]) {
15     // Para cada portal da sala em que o observador está
16     for (APPDivision *portal in [observer.room listOfPortals]) {
17       // Verifica se a nova posição do observador cruza um desses portais
18       if ([APPPortalMathAPI lineWithPointA:observer
19                                  andPointB:newPoint
20                    intersectsLineWithPointC:portal.startPoint
21                                  andPointD:portal.endPoint]) {
22         // Altera a sala em que o observador está
23         if (portal.originRoom.roomIdentifier == observer.room.roomIdentifier)
24           observer.room = portal.destinationRoom;
25         else
26           observer.room = portal.originRoom;
27
28         found = YES;
29         break;
30       }
31     }
32     canMove = found;
33   }
34
35   if (canMove) {
36     // Define a nova coordenada do observador
37     observer.x = newPointX;
38     observer.y = newPointY;
39     // Atualiza o campo de visão
40     [frustum updateCoordinates];
41   }
42 }
```

O método acima faz a movimentação do observador. Primeiramente cria-se um ponto imaginário que representa para onde o observador irá se mover (linhas `08-11`). Após isso, é verificado se essa nova posição não está mais dentro da sala atual (linha `14`). Se a nova posição não estiver mais dentro da sala, é necessário verificar se a movimentação da posição atual do observador para a nova posição cruza algum portal. Essa verificação é feita nas linhas `18-21`. Caso ele esteja cruzando um portal, são atualizadas as informações da sala atual do observador (linhas `23-26`) e a movimentação é feita (linhas `37-40`). Se ao invés de um portal, existir uma parede, a movimentação é cancelada. Não encontrando paredes ou portais no caminho, o observador pode se mover livremente pela sala.

3.3.4 Carregar o modelo 3D

A biblioteca `FAICode` permite a utilização do cenário 3D. Optou-se pela utilização dessa biblioteca desenvolvida por Imianowsky (2013) por ser de código aberto, possibilitando realizar as alterações necessárias para que fosse utilizada em conjunto com a biblioteca de portais. Foi necessário alterar duas classes dessa biblioteca.

A principal alteração foi na classe `FAIOBJParser` que interpreta o arquivo contendo o modelo 3D e armazena as informações em um objeto do tipo `FAIMesh`. Teve-se a necessidade de realizar várias alterações nesta classe a fim de adaptar a mesma para ser utilizada em conjunto com os pacotes `MJCCode` e `APPCode`. Todas as alterações feitas foram com relação a quais objetos devem ser importados dos arquivos contendo os modelos 3D. Essas alterações podem ser vistas no Quadro 19 que mostra o trecho de código fonte do método `parseLine:toMesh:materials:currentMaterial:` onde é feita essa limitação.

Quadro 19 - Alterações na classe `FAIOBJParser` para limitar os objetos importados

```
01 - (void)parseLine:(NSString *)line toMesh:(FAIMesh *)mesh
02                                 materials:(NSMutableDictionary **)materials
03                           currentMaterial:(FAIMaterial **)currentMaterial {
04   NSScanner *scanner = [NSScanner scannerWithString:line];
05   NSString *word = nil;
06   if ([scanner scanWord:&word]) {
07     // verifico se é um objeto valido para ser importado
08     if ([word isEqualToString:@"o"]) {
09       self.objetoValido = [line hasPrefix:@"o SALA_"]
10                         || [line hasPrefix:@"o CHAO"];
11     }
12     else if ([word isEqualToString:@"v"]) {
13       GLKVector3 point = [self parsePointWithScanner:scanner];
14       // apenas armazeno a coordenada no objeto mesh se for um objeto válido
15       if (self.objetoValido) {
16         [mesh addPoint:point];
17       }
18       NSValue *value = [NSValue value:&point withObjCType:@encode(GLKVector3)];
19       // guardo os valores lidos em uma lista para utilizar depois
20       [_points addObject:value];
21     }
22     else if ([word isEqualToString:@"vn"]) {
23       GLKVector3 normal = [self parsePointWithScanner:scanner];
24       // apenas armazeno a coordenada no objeto mesh se for um objeto válido
25       if (self.objetoValido) {
26         [mesh addNormal:normal];
27       }
28       NSValue *value = [NSValue value:&normal withObjCType:@encode(GLKVector3)];
29       // guardo os valores lidos em uma lista para utilizar depois
30       [_normals addObject:value];
31     }
32     else if ([word isEqualToString:@"f"] && self.objetoValido) {
33       NSArray *faces = [self parseFacesWithScanner:scanner
34                                             toMesh:mesh
35                                       withMaterial:*currentMaterial];
36       for (FAIFace3 *face in faces) {
37         [mesh addFace:face];
38       }
39     }
40     else if ([word isEqualToString:@"vt"] && self.objetoValido) {
41       GLKVector2 texture = [self parseTextureCoordinateWithScanner:scanner];
42       [mesh addTexture:texture];
43       NSValue *value = [NSValue value:&texture
44                          withObjCType:@encode(GLKVector2)];
45       // guardo os valores lidos em uma lista para utilizar depois
46       [_textures addObject:value];
47     }
48     ...
49   }
50 }
```

As linhas `09` e `10` verificam se o objeto é válido para ser importado. Os únicos objetos do modelo 3D que serão importados e consequentemente renderizados na tela serão objetos que reprentam a sala (suas paredes) e o chão. O algoritmo antes de importar o objeto verifica se ele possui em seu nome o prefixo `SALA_` ou `CHAO`, armazenando o resultado dessa comparação na variável `objetoValido`. Essa variável é utilizada antes de adicionar as informações no objeto `mesh`. Nas linhas `15` e `25` é verificado se o objeto é valido e então, nas linhas `16` e `26`, as informações são guardadas em estruturas de listas do objeto `mesh`. Teve-se a necessidade de guardar as coordenadas lidas (independente de serem de um objeto válido ou

não) em uma lista, representada no código acima pelas variáveis `_points` (linha `20`), `_normals` (linha `30`) e `_textures` (linha `46`). Isso foi nessário porque os arquivos trabalham com índices e todas as coordenadas do arquivo são necessárias para que a lista tenha o índice completo.

A segunda classe alterada foi a `FAIGraphicObject`. Essa classe é utilizada para desenhar o modelo 3D na tela. Optou-se por adicionar um novo parâmetro no método `drawWithDisplayMode:camera:`, para informar se o modelo 3D deve ou não ser desenhado na tela. O Quadro 20 mostra o trecho do código fonte do método que foi alterado.

Quadro 20 - Alterações na classe `FAIGraphicObject` para informar se o modelo deve ser desenhado

```
01 - (void)drawWithDisplayMode:(GraphicObjectDisplayMode)displayMode
02                      camera:(FAICamera *)camera
03                   andCanDraw:(BOOL)canDraw {
04   // Código omitido por não ter alteração
05   ...
06
07   if (canDraw) {
08     glDrawArrays(mode, 0, meshMaterial.trianglePointsLength);
09   }
10
11   // Código omitido por não ter alteração
12   ...
13 }
```

No código acima, foi criado um novo parâmetro (linha `03`) onde deve-se informar se o modelo 3D vai ser desenhado na tela. A linha `07` verifica se o mesmo pode ser denhado. Caso sim, a linha `08` executa o comando padrão do OpenGL ES `gkDrawArrays` responsável por desenhar as informações passadas.

3.3.5 Interpretação dos mapas

A leitura dos arquivos `MAP` contendo informações dos mapas é feita através das classes disponíveis no pacote `MJCCode`. Essa parte do processo é composta por dois grandes passos realizados pelas duas principais classes do pacote.

O primeiro passo é realizado pela classe `MJCMapParser`. Essa classe contém os métodos que realizam a interpretação do arquivo `MAP`. Primeiramente a classe utiliza o método privado `parseLine` para reconhecer e interpretar cada linha do arquivo. O Quadro 21 mostra o código fonte desse método.

Quadro 21 - Método `parseLine:` responsável por ler as linhas do arquivo

```
01 -(void)parseLine:(NSString*)line {
02   if ([line length] == 0) {
03     return;
04   }
05   unichar identifier = [line characterAtIndex:0];
06   // Caso seja um nome de objeto
07   if (identifier == 'o') {
08     NSString *name = [line substringFromIndex:2];
09     // Caso seja um mapa de sala
10     if ([name.uppercaseString hasPrefix:@"MAPA_SALA_"]) {
11       [self createRoom:name];
12     }
13     // Caso seja um portal
14     else if ([name.uppercaseString hasPrefix:@"PORTAL_"]) {
15       [self createPortal:name];
16     }
17     // Caso seja um ponto de interesse
18     else if ([name.uppercaseString hasPrefix:@"WAYPOINT_"]) {
19       [self createWaypoint:name];
20     }
21     // Caso seja um observador
22     else if ([name.uppercaseString hasPrefix:@"OBSERVADOR_"]) {
23       [self createObserver:name];
24     }
25     else {
26       [NSException raise:@"Objeto desconhecido"
27                   format:@"'%@' representa um objeto desconhecido.",
28                          name.uppercaseString];
29     }
30   }
31   // Caso a linha comece com 'e' significa que contém informações das
32   // arestas (parede de uma sala ou portal)
33   else if (identifier == 'e') {
34     [self readEdge:line];
35   }
36   // Caso a linha comece com 'v' significa que contém informações de um
37   // vértice (ponto de interesse ou observador)
38   else if (identifier == 'v') {
39     [self readVertex:line];
40   }
41 }
```

O método recebe como parâmetro o texto contido na linha que está sendo lida pelo algoritmo. Na linha `05` o método armazena na variável `identifier` o primeiro caractere do texto, que representa a informação que está contida naquele texto. Caso a variável `identifier` contenha o caractere `o` significa que a linha contém informações sobre o nome do objeto. Logo, é verificado o prefixo do objeto a fim de reconhecer seu tipo. Para cada tipo reconhecido, é chamado um método específico para interpretar e criar os objetos necessários. Se o primeiro caractere do texto for um `e` ou `v`, significa que o texto contém informações sobre coordenadas. Logo o método chama outros métodos específicos para interpretarem essas coordendas.

Após ler todas as linhas do arquivo é necessário fazer a ligação entre os objetos. Salas são compostas por paredes e portais. Pontos de interesse e observadores estão dentro de uma

determinada sala. Essa ligação é feita no método público `parserAsListOfRooms`. Os detalhes do método podem ser vistos no Quadro 22.

Quadro 22 - Método `parseAsListOfRooms` responsável por estruturar a lista de objetos

```
01 -(NSMutableDictionary *)parserAsListOfRooms {
02   // Abro o arquivo MAP
03   MJCFileReader *fileReader =
04                   [[MJCFileReader alloc] initWithFile:self.fileName];
05   NSArray *lines = [fileReader read];
06
07   // Para cada linha
08   for (NSString *line in lines) {
09     NSString *lineWithoutComments =
10               [[line componentsSeparatedByString:@"#"] objectAtIndex:0];
11     // Interpreto o texto da linha
12     [self parseLine:lineWithoutComments];
13   }
14
15   // Para cada portal encontrado
16   for (MJCPortal *portal in self.portals) {
17     // Pego o quarto de origem dele
18     MJCRoom *room = (MJCRoom*)[self.rooms objectForKey:[[NSNumber alloc]
19                       initWithInteger:portal.origenRoomIdentifier]];
20     // Adiciono o portal nesse quarto
21     [room addPortal:portal];
22
23     // Pego o quarto de destino dele
24     room = (MJCRoom*)[self.rooms objectForKey:[[NSNumber alloc]
25                                initWithInteger:portal.targetRoomIdentifier]];
26     // Adiciono ele nesse quarto
27     [room addPortal:portal];
28   }
29   // Para cada ponto de interesse
30   for (MJCWaypoint *waypoint in self.waypoints) {
31     // Pego o quarto onde ele se encontra
32     MJCRoom *room = (MJCRoom*)[self.rooms objectForKey:[[NSNumber alloc]
33                       initWithInteger:waypoint.identifierRoom.integerValue]];
34     // Se for um observador
35     if (waypoint.isObserver) {
36       // Adiciono ele na lista de observadores desse quarto
37       [room addObserver:waypoint];
38     }
39     // Se for um ponto de interesse
40     else {
41       // Adiciono ele na lista de pontos de interesse desse quarto
42       [room addWaypoint:waypoint];
43     }
44   }
45   return self.rooms;
46 }
```

Este trecho de código mostra o principal método da classe. Primeiro é feito o carregamento do arquivo em memória (linhas `03` e `04`) e a interpretação de cada linha dele (método `parseLine:` demonstrado no Quadro 21). Com os objetos que representam salas, portais, pontos de interesse e observadores já criados, é necessário interligar cada um deles. Por exemplo, um portal pertence a duas salas e um ponto de interesse ou observador está dentro de uma sala. Na estrutura criada, uma sala possui uma lista dos portais que ela possui. É buscado a sala de origem do portal e nessa sala é adicionado o portal. O mesmo ocorre com a sala de destino. Após estruturar as salas é necessário adicionar todos os pontos de interesse e o

observador. A linha 30 percorre todos os pontos de interesse encontrados no arquivo. O ponto de interesse que for definido como observador é adicionado a lista de observadores da sala. Todos os outros pontos de interesse são adicionados na própria lista de pontos de interesse.

O segundo passo consiste em disponibilizar a estrutura das salas, portais, pontos de interesse e observador em um formato aceito pela biblioteca de portais. Para isso foi criado a classe MJCMapLoader. Esta classe possui um método privado que converte os objetos criados pela classe MJCMapParser em objetos da biblioteca de portais. O Quadro 23 mostra a visão geral do método.

Quadro 23 - Método loadRooms: que converte os objetos para a biblioteca de portais

```
01 -(void)loadRooms:(NSString *)file {
02    MJCMapParser *mapParser = [[MJCMapParser alloc] initWithFile:file];
03    NSMutableDictionary *roomsList = [mapParser parserAsListOfRooms];
04    // Converto as salas
05    [self convertRooms:roomsList];
06    for (MJCRoom *room in [roomsList allValues]) {
07      // Converto os portais
08      [self convertPortals:room];
09      // Converto os pontos de interesse
10      for (MJCWaypoint *wp in room.waypoints) {
11        APPWaypoint *pontoInteresse = [[APPWaypoint alloc]
12                      initWithX:wp.position.x
13                           andY:wp.position.y
14                           andZ:wp.position.z
15                  belongsToRoom:[self getRoomByID:[wp.identifierRoom intValue]]];
16        [self.waypoints addObject:pontoInteresse];
17      }
18      // Converto o obsdervador
19      for (MJCWaypoint *wp in room.observers) {
20        APPPoint *observer = [[APPPoint alloc]
21                  initWithX:wp.position.x
22                       andY:wp.position.y
23                       andZ:wp.position.z
24              belongsToRoom:[self getRoomByID:[wp.identifierRoom integerValue]]];
25        [self.observers addObject:observer];
26      }
27    }
28  }
```

O código acima primeiramente utiliza o método parserAsListOfRooms para criar a lista de objetos. Logo em seguida utiliza-se o método convertRooms: para converter todas as salas. Após converter as salas, é convertido cada portal (através do método convertPortals:), ponto de interesse e o observador. Os quadros Quadro 24 e Quadro 25 mostram com detalhes os dois métodos utilizados.

Quadro 24 - Método `convertRooms:` responsável por converter as salas

```
01 - (void)convertRooms:(NSMutableDictionary *)roomsList {
02   // converto as salas
03   for (MJCRoom *room in [roomsList allValues]) {
04     APPRoom *sala = [[APPRoom alloc]
05                      initWithRoomIdentifier:[room.identifier intValue]];
06     // converto as paredes
07     for (MJCWall *wall in room.walls) {
08       APPPoint *ponto1 = [[APPPoint alloc]
09                           initWithX:wall.startPoint.x
10                                andY:wall.startPoint.y
11                                andZ:wall.startPoint.z
12                        belongsToRoom:sala];
13       APPPoint *ponto2 = [[APPPoint alloc]
14                           initWithX:wall.endPoint.x
15                                andY:wall.endPoint.y
16                                andZ:wall.endPoint.z
17                        belongsToRoom:sala];
18       APPDivision *divisao = [[APPDivision alloc]
19                               initWithStartPoint:ponto1
20                                         andEndPoint:ponto2
21                                              ofType:APPDivisionTypeWall];
22       [sala addDivision:divisao];
23     }
24     [self.rooms setObject:sala forKey:
25                 [[NSNumber alloc] initWithInt:sala.roomIdentifier]];
26   }
27 }
```

O método `convertRooms:` pega todos as salas do tipo `MJCRoom` e as converte em salas do tipo `APPRoom`. Para cada parede do tipo `MJCWall` o método cria uma divisão (`APPDivision`) com as mesmas informações. Isso é feito para disponibilizar a sala numa estrututa interpretada pela biblioteca de portais. O mesmo ocorre para os portais.

Quadro 25 - Método `convertPortals:` responsável por converter os portais

```
01 - (void)convertPortals:(MJCRoom *)room {
02   // para cada portal da sala
03   for (MJCPortal *portal in room.portals) {
04     APPRoom *salaOrigem = [self.rooms objectForKey:[[NSNumber alloc]
05                            initWithInt:portal.origenRoomIdentifier]];
06     APPRoom *salaDestino = [self.rooms objectForKey:[[NSNumber alloc]
07                            initWithInt:portal.targetRoomIdentifier]];
08
09     APPPoint *ponto1 = [[APPPoint alloc] initWithX:portal.startPoint.x
10                                               andY:portal.startPoint.y
11                                               andZ:portal.startPoint.z
12                                      belongsToRoom:salaOrigem];
13     APPPoint *ponto2 = [[APPPoint alloc] initWithX:portal.endPoint.x
14                                               andY:portal.endPoint.y
15                                               andZ:portal.endPoint.z
16                                      belongsToRoom:salaOrigem];
17     // crio uma divisão do tipo portal
18     APPDivision *divisao = [[APPDivision alloc]
19                             initWithStartPoint:ponto1
20                                    andEndPoint:ponto2
21                                         ofType:APPDivisionTypePortal];
22     [salaOrigem addDivision:divisao];
23
24     [divisao setOriginRoom:salaOrigem];
25     [divisao setDestinationRoom:salaDestino];
26   }
27 }
```

O método `convertPortals:` pega todos os portais de uma sala do tipo `MJCRoom` e os converte em portais do tipo `APPDivision`. A diferença entre as paredes e portais está no tipo que é passado no construtor do objeto. Para paredes, o tipo é `APPDivisionTypeRoom` e para portais é `APPDivisionTypePortal`.

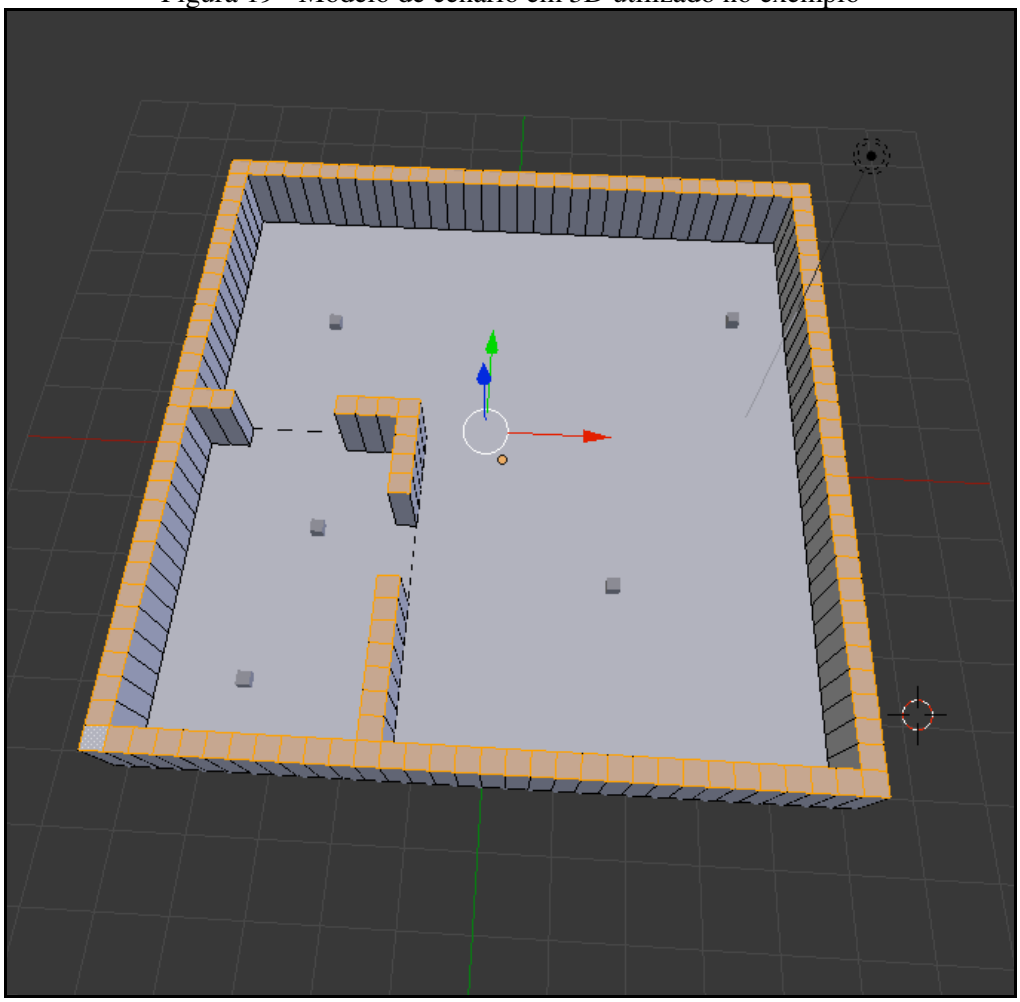
3.3.6 Operacionalidade da implementação

Esta seção tem por objetivo mostrar a operacionalidade da implementação em nível de desenvolvedor. Para isto será exemplificada a criação do mapa e a criação da classe de controle que contém os métodos necessários para executar a biblioteca de portais.

3.3.6.1 Criação do mapa

O desenvolvimento do mapa é feito através da ferramenta de modelagem Blender. A Figura 19 apresenta um cenário de exemplo desenvolvido no Blender (ver APÊNDICE A).

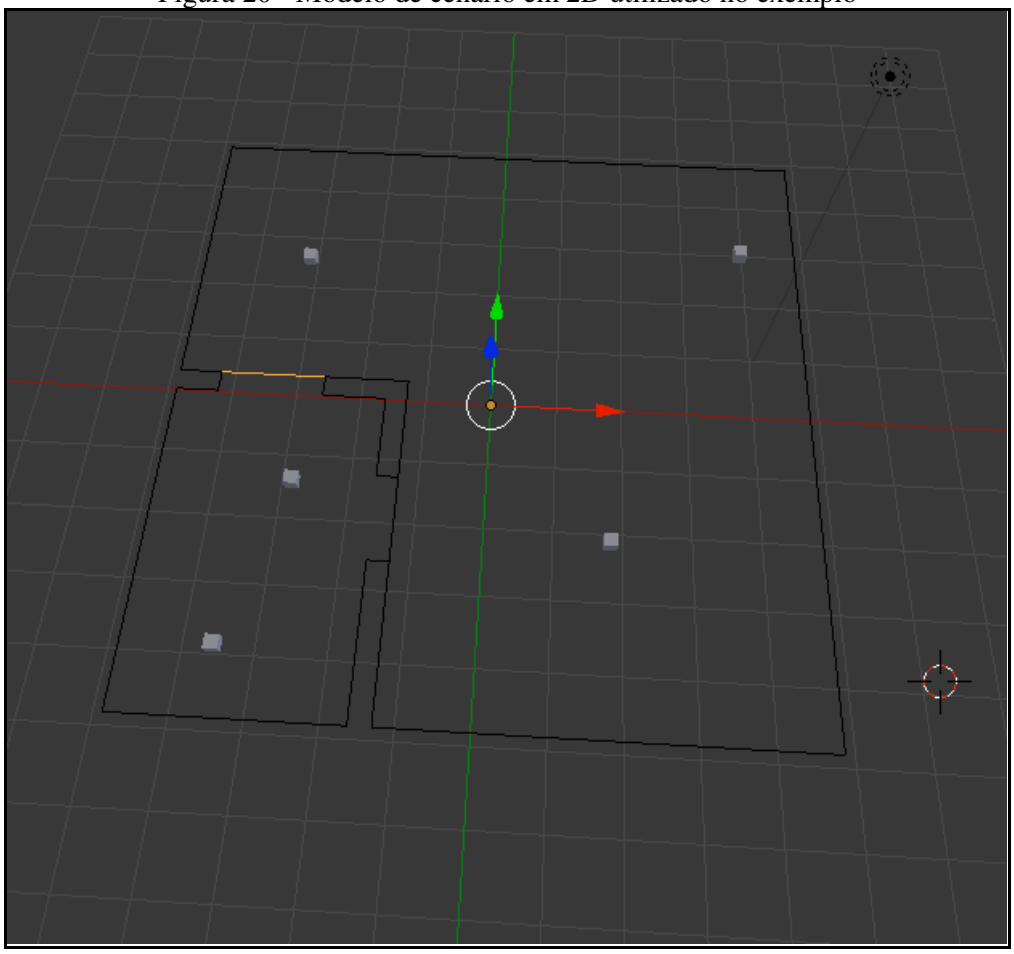
Figura 19 - Modelo de cenário em 3D utilizado no exemplo



O cenário apresentado na figura acima possui duas salas e dois portais. Além das salas e portais, ele possui um observador e quatro pontos de interesse. Visualmente não há diferença entre o observador e os pontos de intresse. Essa distinção é feita através do nome dados a eles.

O modelo apresentado na Figura 19 está com paredes e um chão, representado em sua forma 3D. Para o algoritmo de portais o que importa é o modelo em duas dimensões criado abaixo desse modelo. A Figura 20 apresenta o mapa da sala no ambiente de duas dimensões.

Figura 20 - Modelo de cenário em 2D utilizado no exemplo

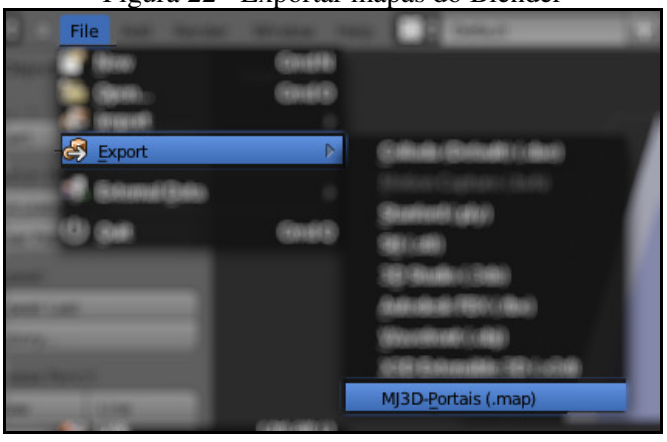


A biblioteca de portais utilizará o modelo 2D como referência para delimitar e reconhecer cada sala. A representação em 3D do mapa é apenas visual, não tendo importância para o algoritmo de portais. Na imagem estão sendo mostradas apenas as linhas que represesntam as extremidades de cada sala. Visualmente não há diferença entre paredes e portais. A distinção entre esses objetos é feita através do nome dados a eles. A Figura 21 apresenta a lista de objetos contido no cenário de exemplo.

Figura 21 - Lista de objetos do cenário de exemplo

Os objetos da lista são todos os objetos contidos no cenário. Após criar o cenário e nomear os objetos de acordo com o Quadro 2 é necessário utilizar o *add-on* de exportação. O *add-on* é disponibilizado no Blender através do menu de exportação. Como realizar a instalação do mesmo está descrito no APÊNDICE B. A Figura 22 mostra o comando no menu.

Figura 22 - Exportar mapas do Blender



3.3.6.2 Criando a classe de controle

A classe de controle é utilizada para agregar todas as funcionalidades da biblioteca. Nela serão armazenados os métodos que farão a leitura, configuração e execução dos passos

necessários para o funcionamento do algoritmo de portais. O Quadro 26 mostra os principais métodos da classe.

Quadro 26 - Principais métodos da classe de controle

```
01 // Métodos responsáveis pelo algoritmo de portais
02 -(void)moveObserver;
03 -(void)rotateFrustumLeft;
04 -(void)rotateFrustumRight;
05
06 // Métodos responsáveis pelo carregamento do modelo 3D
07 // e do cenários 2D
08 -(void)loadObject3D;
09 -(void)loadObject2D;
10 -(void)configureObject3D;
11 -(void)configureObject2D;
12 -(void)drawObject3D;
13 -(void)drawObject2D;
```

O desenvolvedor encapsula as funcionalidades da biblioteca nesses métodos e os utiliza na aplicação principal. Primeiramente o usuário deve executar o método que carrega o objeto (linhas `08` e `09`). Após isso deve ser executado o método de configuração (linhas `10` e `11`). Por fim, é necessário desenhar o objeto na tela através dos métodos que realizam o desenho (linhas `12` e `13`).

3.3.6.2.1 Carregar os mapas e configurar o algoritmo de portais

Primeiramente o desenvolvedor deverá carregar as informações referentes ao mapa das salas. O Quadro 27 mostra o de código responsável por esta funcionalidade.

Quadro 27 - Método responsável por carregar os mapas das salas

```
01 -(void)loadObject2D {
02   self.mapLoader = [[MJCMapLoader alloc] initWithMapName:self.filename];
03 }
```

Após carregar o arquivo contendo as informações do mapa (arquivo com extensão `MAP`) é necessário realizar as configurações do algoritmo de portais. O Quadro 28 mostra o código fonte do método de configurações.

Quadro 28 - Método que configura os parâmetros do algoritmo de portais

```
01 -(void)configureObject2D {
02   if (self.mapLoader.rooms.count == 0
03    || self.mapLoader.observers.count == 0) {
04     // Mensagens de erro...
05   }
06   else {
07     // Configuro o observador
08     APPPoint *firstObserver = [self.mapLoader.observers objectAtIndex:0];
09
10     self.frustumAngle = 180.0;
11     self.observerSpeed = 0.1;
12     self.frustumOpening = 15;
13     self.frustumFar = 3;
14
15     self.observer = [[APPObserver alloc] initWithX:firstObserver.x
16                                               andY:firstObserver.y
17                                               andZ:firstObserver.z
18                                     belongsToRoom:firstObserver.room];
19     // Configuro o Frustum do observador
20     self.frustum = [[APPFrustum alloc] initWithCamera:self.observer
21                                              andAngle:self.frustumAngle
22                                            andOpening:self.frustumOpening
23                                               andFar:self.frustumFar];
24     self.auxiliaryFrustumsList = [[NSMutableArray alloc] init];
25     self.portalAPI = [[APPPortalAPI alloc] init];
26
27     self.currentRoom = self.observer.room.roomIdentifier;
28
29     [self setupColorsObject2D];
30   }
31 }
```

Primeiramente é definido quem é o observador no mapa (linha `08`). Logo em seguida são configurados sua posição (linhas `15-18`) e o tamanho do campo de visão (linhas `20-23`). Após isso, é inicializada a classe necessária para que o observador navega entre as salas (linha `25`). Por fim é feita a configuração das cores que cada objeto na sala receberá (linha `29`).

Após configurar os parâmetros do algoritmo de portais é necessário desenhar os objetos gráficos na tela. O Quadro 29 mostra o código fonte responsável por esta parte da classe.

Quadro 29 - Método responsável por desenhar os objetos do algoritmo de portais na tela

```
01 -(void)drawObject2D {
02   glEnableVertexAttribArray(GLKVertexAttribColor);
03   glEnableVertexAttribArray(GLKVertexAttribPosition);
04   // desenha as salas
05   for (APPRoom *room in self.mapLoader.rooms.allValues) {
```

```
06      [self drawRoom:room];
07    }
08    // desenha os pontos de interesse vistos
09    glVertexAttribPointer(GLKVertexAttribColor, 4, GL_FLOAT,
10                          GL_FALSE, 0, self.insideFrustumWaypointColor);
11    for (APPWaypoint *waypoint in self.observer.waypointsInsideFrustumView) {
12      [self drawWaypoint:waypoint];
13    }
14    // desenha os pontos de interesse não vistos
15    glVertexAttribPointer(GLKVertexAttribColor, 4, GL_FLOAT,
16                          GL_FALSE, 0, self.normalWaypointColor);
17    for (APPWaypoints *waypoint in self.mapLoader.waypoints) {
18      [self drawWaypoint:waypoint];
19    }
20    // desenha o frustum
21    glVertexAttribPointer(GLKVertexAttribColor, 4, GL_FLOAT,
22                          GL_FALSE, 0, self.frustumColor);
23    [self drawFrustum:self.frustum];
24    // desenha o observador
25    glVertexAttribPointer(GLKVertexAttribColor, 4, GL_FLOAT,
26                          GL_FALSE, 0, self.observerColor);
27    [self drawObserver:self.observer];
28    // desenha os frustums auxiliares
29    glVertexAttribPointer(GLKVertexAttribColor, 4, GL_FLOAT,
30                          GL_FALSE, 0, self.frustumColor);
31    glEnable (GL_BLEND);
32    glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
31    for (APPFrustum *f in self.auxiliaryFrustumsList) {
33      [self drawFrustum:f];
34    }
35    glDisable (GL_BLEND);
36
37    glDisableVertexAttribArray(GLKVertexAttribColor);
38    glDisableVertexAttribArray(GLKVertexAttribPosition);
39  }
```

O código acima é responsável por desenhar os objetos (salas, portais, pontos de interesse e observador) na tela. Primeiramente são desenhadas todas as paredes de todas as salas (linha `06`). Logo em seguida são desenhados os pontos de interesse que estão sendo vistos pelo observador (linha `12`). Depois são desenhados os outros pontos de interesse (linha `18`), o campo de visão principal (linha `23`) e por fim todos os campos de visão auxiliares (linha `33`).

3.3.6.2.2 Carregar o modelo 3D

Após carregar e configurar os mapas e o algoritmo de portais, é necessário realizar os mesmos passos para o modelo 3D. O Quadro 30 mostra o código fonte responsável por carregar as informações do referido modelo.

Quadro 30 - Método responsável por carregar as informações do modelo 3D

```
01 -(void)loadObject3D {
02   FAIOBJParser *parser = [[FAIOBJParser alloc] initWithFilename:self.filename];
03   self.mesh = [parser parseAsObject];
04 }
```

Após o carregamento do arquivo que contém as informações do modelo, é necessário realizar a configuração do ambiente. O Quadro 31 mostra o trecho de código que realiza essa configuração.

Quadro 31 - Método responsável por configurar os parâmetros do modelo 3D

```
01 -(void)configureObject3D {
02   self.object3D = [[FAIGraphicObject alloc] initWithMesh:self.mesh];
03   [self.object3D.transform translateToOrigin];
04   [self configureCamera];
05 }
06
07 -(void)configureCamera {
08   self.camera = [[FAICamera alloc] init];
09   self.camera.aspect = fabsf(self.aspect);
10   self.camera.fovyDegrees = 60.0f;
11   self.camera.eyeZ = 100.0f;
12
13   self.camera.centerX = 0.0f;
14   self.camera.centerY = 0.0f;
15   self.camera.centerZ = 0.0f;
16
17   self.camera.upX = 0.0f;
18   self.camera.upY = 1.0f;
19   self.camera.upZ = 0.0f;
20
21   self.camera.eyeX = 0.0f;
22   self.camera.eyeY = 0.0f;
23
24   GLfloat maxValue = MAX(self.object3D.width, self.object3D.height);
25
26   if (maxValue > 0.0f) {
27     self.camera.eyeZ = maxValue * 2;
28   }
29 }
```

No código acima é feita a configuração do ambiente (linha 03) e o posicionamento da câmera no ambiente (linhas 08-27). A câmera será posicionada para que o objeto caiba por completo na tela (linha 27).

Por fim é necessário desenhar o modelo na tela. O Quadro 32 apresenta o método utilizada para realizar esta função.

Quadro 32 - Método responsável por desenhar o modelo 3D

```
01 -(void)drawObject3D {
02   // caso o objeto já esteja carregado em memória
03   if (self.object3D && self.camera) {
04     [self.object3D drawWithDisplayMode:GraphicObjectDisplayModeSolid
05                                 camera:self.camera
06                          andCanDraw:self.viewType == ViewType3D];
07   }
08   else {
09     NSAssert(!self.object3D, @"object3D can't be null");
10     NSAssert(!self.camera, @"camera can't be null");
11   }
12 }
```

O código acima desenha o modelo na tela de forma sólida (sem texturas). Como parâmetros são especificados a câmera previamente configurada e por último um parâmetro que informa se o modelo deve ou não ser desenhado.

3.3.6.2.3 Movimentação do observador

A movimentação do observador é feita atráves de três métodos. Os métodos `rotateFrustumLeft` e `rotateFrustumRight` são responsáveis por fazer a rotação do campo de visão do observador. O código fonte dos dois métodos pode ser visto no Quadro 33.

Quadro 33 - Métodos responsáveis por girar o campo de visão do observador

```
01 -(void)rotateFrustumRight {
02   [self.frustum rotateTo:APPFrustumDirectionRight];
03   self.frustumAngle = self.frustum.angulo;
04   [self checkWaypoints];
05 }
06 -(void)rotateFrustumLeft {
07   [self.frustum rotateTo:APPFrustumDirectionLeft];
08   self.frustumAngle = self.frustum.angulo;
09   [self checkWaypoints];
10 }
```

Os métodos acima chamam funcionalidades já presentes na biblioteca de portais. Essa funcionalidades fazem a rotação do campo de visão para o sentido desejado (horário ou anti-horário). Por fim, após executar a rotação é feita a verificação dos pontos de interesse que estão dentro do campo de visão do observador.

O terceiro método é responsável por realizar a movimentação do observador. O Quadro 34 mostra o código responsável pela movimentação.

Quadro 34 - Método responsável por movimentar o observador

```
01 -(void)moveObserver {
02   // busco a nova coordenada x
03   float x = [APPPortalMathAPI generateNewXCoordinate:self.observer.x
04                                             withAngle:self.frustumAngle
05                                             andRadius:self.observerSpeed];
06   // busco a nova coordenada y
07   float y = [APPPortalMathAPI generateNewYCoordinate:self.observer.y
08                                             withAngle:self.frustumAngle
09                                             andRadius:self.observerSpeed];
10   // chamo a biblioteca de portais
11   [self.portalAPI moveObserver:self.observer
12                     ToNewPointX:x
13                    andNewPointY:y
14                 updatingFrustum:self.frustum];
15   // verifico os pontos de interesse dentro do
16   // campo de visão do observador
17   [self checkWaypoints];
18   self.currentRoom = self.observer.room.roomIdentifier;
19 }
```

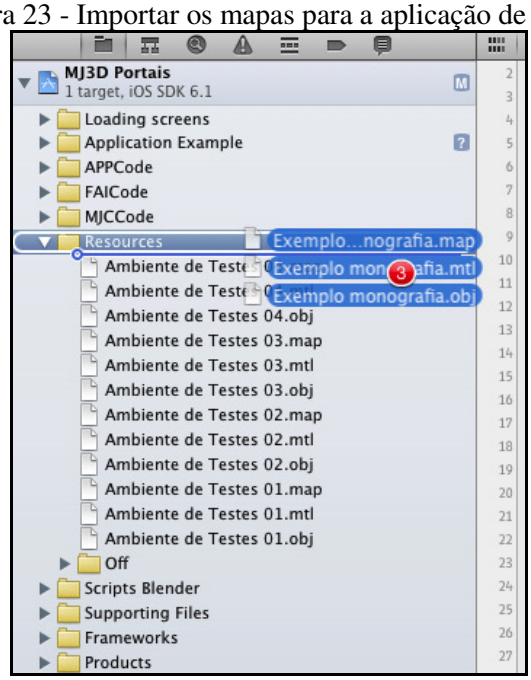
O método executa as funcionalidades presentes na biblioteca de portais para buscar as novas coordenadas (linhas `03-05` e `07-09`). Logo em seguida movimenta o observador para as novas coordenadas (linhas `11-14`). Por fim verifica se existe algum ponto de interesse dentro do campo de visão do observador (linha `17`).

3.3.7 Aplicação de exemplo

Para testar a biblioteca desenvolvida, foi criado um aplicativo de exemplo que pode ser executado tanto no iPhone quanto no iPad.

Primeiramente o usuário deverá transferir os mapas que vão ser carregados na aplicação. Esta etapa do processo é feita com a ferramenta de desenvolvimento Xcode, arrastando os mapas para a pasta `Resources` da aplicação. A Figura 23 demonstra o processo.

Figura 23 - Importar os mapas para a aplicação de testes

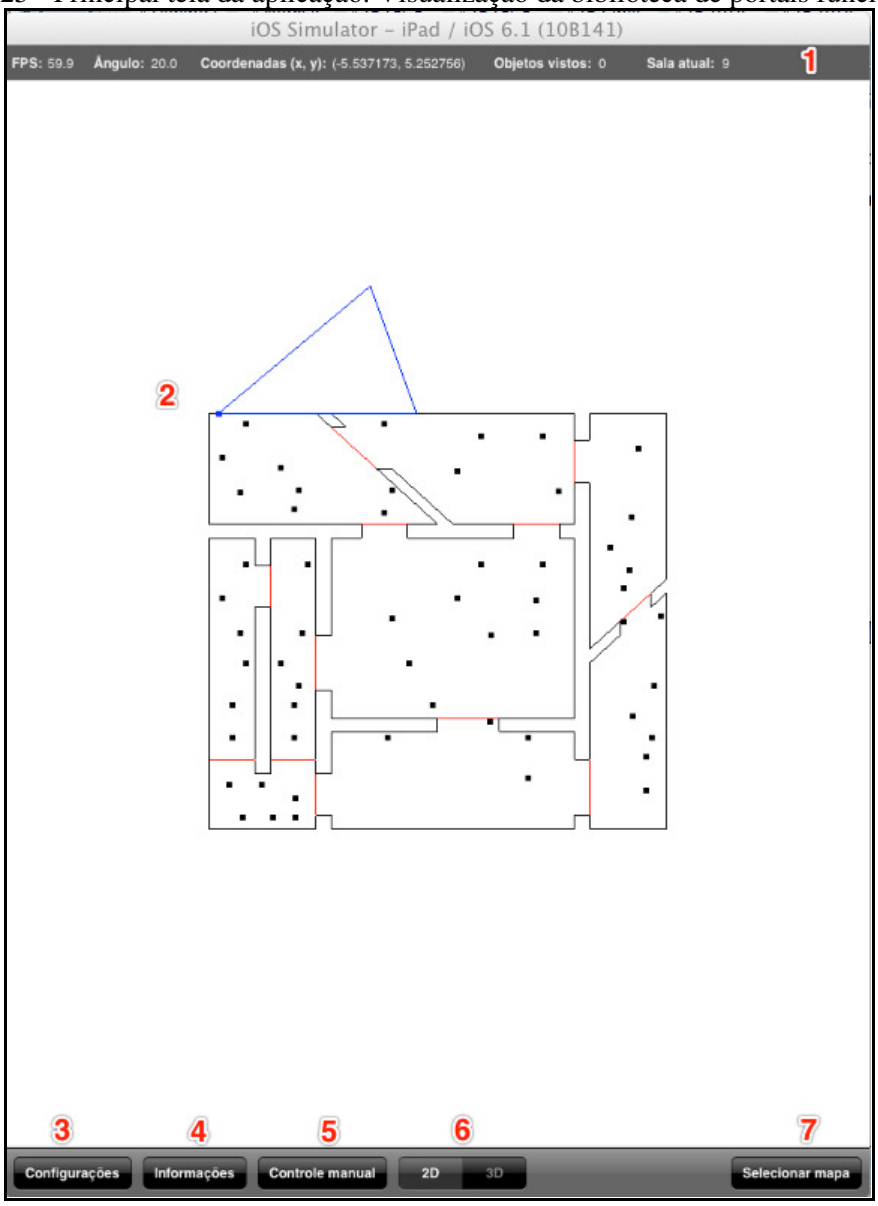


O aplicativo desenvolvido possui basicamente duas telas. A tela principal onde será exibido o mapa e as funcionalidades da biblioteca e uma tela onde o usuário poderá selecionar o mapa que deseja carregar. Ao inicializar o aplicativo, o usuário será redirecionado para a tela de seleção de mapas. Na Figura 24 pode ser vista a lista de mapas disponíveis na aplicação.

Figura 24 - Tela onde o usuário irá selecionar o mapa que deseja carregar

Na lista de mapas (Figura 24 (1)) a aplicação mostrará todos os mapas previamente carregados para o aplicativo. Para carregar um mapa, o usuário deve selecionar um mapa da lista e clicar no botão `Carregar` (Figura 24 (3)). Caso o usuário deseje voltar à tela anterior, pode-se cancelar esta ação e voltar à tela principal (Figura 24 (2)). Após carregar um mapa ou cancelar a operação o aplicativo volta para a tela principal. O mapa carregado pode ser visto na Figura 25.

Figura 25 - Principal tela da aplicação. Visualização da biblioteca de portais funcionando



Na tela principal é executado o algoritmo de portais sobre o mapa selecionado. Informações relacionadas a execução da biblioteca podem ser vistas no topo da tela (Figura 25 (1)). As informações apresentadas são:

a) `FPS`: taxa de atualização da tela em FPS;

b) `Ângulo`: ângulo em que o observador está apontando o campo de visão;

c) `Coordenadas (x, y)`: coordenadas do observador no mapa;

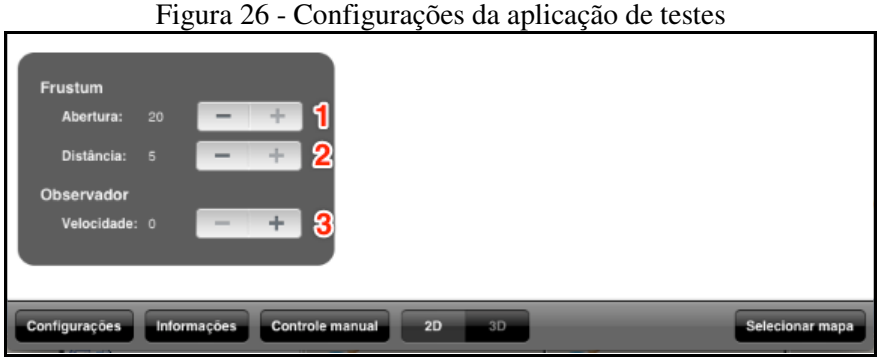
d) `Objetos vistos`: quantidade de objetos que estão sendo vistos pelo observador;

e) `Sala atual`: identificador da sala atual.

Caso o usuário não deseje ver tais informações, pode-se ocultar ou reexibir as mesmas através do botão `Informações` (Figura 25 (4)).

O usuário tem a opção de realizar configurações na biblioteca de portais em tempo real. As configurações são exibidas/ocultas através do botão `Configurações` (Figura 25 (3)). A Figura 26 mostra as configurações disponíveis.

Figura 26 - Configurações da aplicação de testes



São permitidas três configurações básicas:

a) `Abertura` (Figura 26 (1)): define o ângulo de abertura do campo de visão do observador;

b) `Distância` (Figura 26 (2)): distância entre o observador e o término do campo de visão;

c) `Velocidade` (Figura 26 (3)): velocidade do observador.

Caso o usuário deseje movimentar o personagem pelo mapa manualmente, pode-se alterar o modo de execução para controle manual (Figura 25 (5)). A Figura 27 mostra os controles disponíveis.

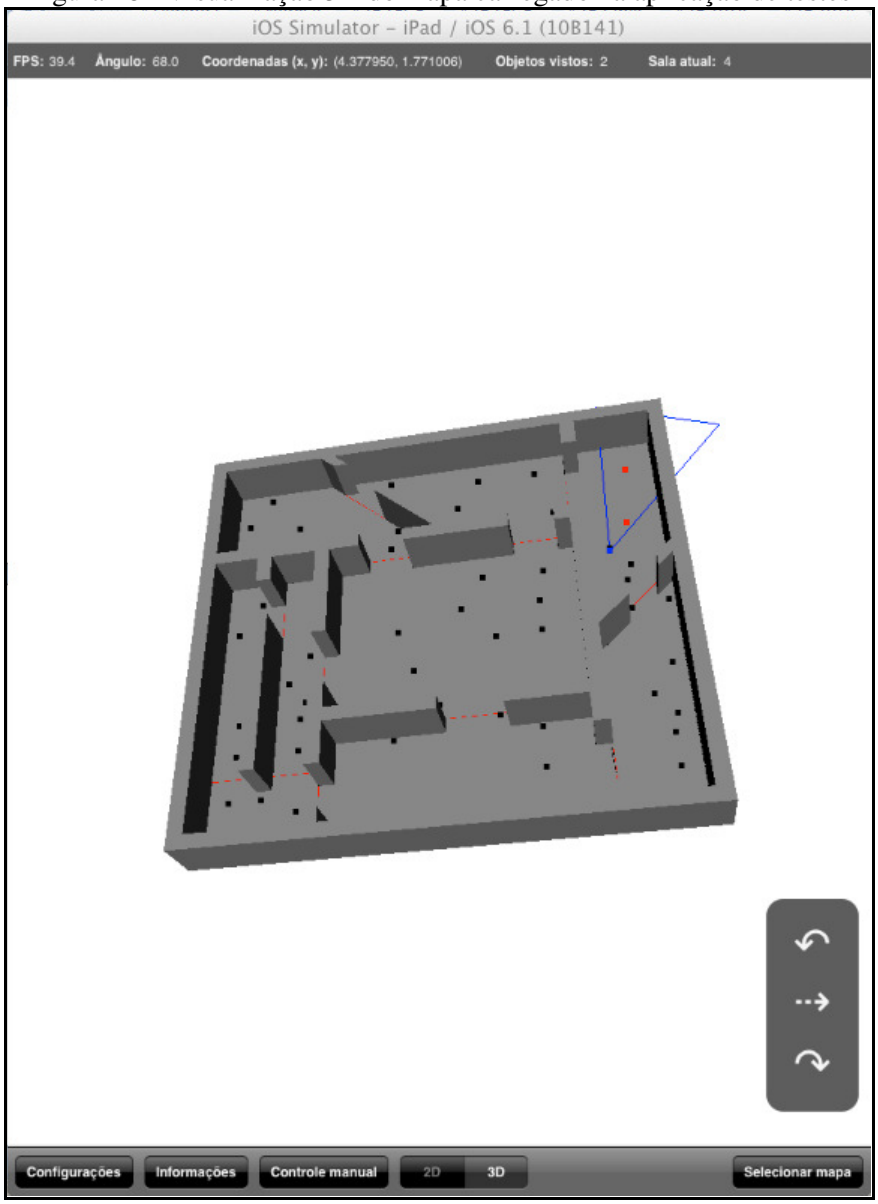
Figura 27 - Controle manual da movimentação do observador

O usuário tem a opção de rotacionar o campo de visão para o sentido anti-horário (Figura 27 (1)) ou sentido horário (Figura 27 (3)), além de poder movimentar o personagem para a direção em que o campo de visão está apontando (Figura 27 (2)). Todos os três botões

responsáveis por movimentar o personagem possuem uma função que os deixa executando se o botão for pressionado duas vezes rapidamente (execução automática da função do botão).

A aplicação de exemplo permite que o usuário alterne entre as visualizações em duas e três dimensões (Figura 25 (6)). A Figura 28 demonstra o modo 3D.

Figura 28 - Visualização 3D do mapa carregado na aplicação de testes



A aplicação permite que o usuário realize rotações ou escala em cima do mapa carregado (tanto em 2D quanto em 3D), tais comandos podem ser feitos através de gestos:

a) gesto de pinça: realiza uma escala no mapa carregado;

b) gesto de clicar e arrastar: realiza uma rotação no mapa carregado.

Por fim, caso o usuário deseje carregar outro mapa, tal ação pode ser executada clicando no botão `Carregar mapa` (Figura 25 (7)). A tela carregada ao clicar neste botão é a mesma da Figura 24.

3.4 RESULTADOS E DISCUSSÃO

Algumas dificuldades foram encontradas na utilização das tecnologias envolvidas. A primeira dificuldade foi a diferença entre a linguagem Objective-C, utilizada no desenvolvimento deste trabalho e a linguagem Java, utilizada na biblioteca desenvolvida por Pandini (2012). Por serem linguagens de programação distintas, algumas funcionalidades presentes no Java não estavam presentes no Objective-C. Isso fez com que outras formas de se resolver o mesmo problema fossem desenvolvidas.

Outra dificuldade foi encontrada ao converter a biblioteca original para a plataforma iOS. Os algoritmos matemáticos, testes de visibilidade e movimentação do observador desenvolvidos por Pandini (2012) não se mostraram aptos para serem utilizados nos cenários desenvolvidos neste trabalho. Ao executar os algoritmos convertidos da biblioteca desenvolvida por Pandini (2012) em alguns mapas criados no Blender, foram encontrados problemas na movimentação do observador pelo mapa, encontrando paredes onde não deveria e saindo das limitações do mapa.

Optou-se então por refazer os algoritmos matemáticos responsáveis por verificar se um ponto está dentro de um polígono, verificar se um ponto está dentro de um triângulo e incluir um algoritmo utilizado para verificar se duas linhas se cruzam (ver seção 3.3.3). Além disso, foi necessário refazer os métodos responsáveis pela movimentação do observador e pelos testes de visibilidade para que a biblioteca convertida tivesse a mesma funcionalidade já presente na biblioteca de Pandini (2012).

A biblioteca desenvolvida neste trabalho possui algumas limitações. Quando dois portais estão sendo vistos pelo observador e ambos os portais apontam para a mesma sala, nem todos os pontos de interesses presentes nessa sala são vistos pelo observador. Também existe uma restrição na criação dos ambientes, já que os mapas devem ser desenvolvidos com a ferramenta de modelagem Blender. Além da obrigatoriedade do uso dessa ferramenta, quem deseja criar os mapas deve seguir um padrão de nomes (descrito no Quadro 2) e utilizar o *add-on* desenvolvido para exportá-los.

Outra limitação está nos testes de visibilidade. Todos os testes executados pela biblioteca são feitos com informações de um ambiente 2D, sendo que a parte 3D da biblioteca é apenas visual, caracterizando uma aplicação de espaço em 2D e ½.

3.4.1 Testes de desempenho da aplicação da biblioteca

Para aferir o desempenho da biblioteca desenvolvida, foram realizadas medições na aplicação de exemplo. Foram realizados dois testes distindos. O primeiro testa o desempenho

de dois dos principais métodos da biblioteca. O segundo compara o desempenho do método que verifica se um ponto está dentro de um triângulo. A comparação envolve o método (já convertido para a plataforma iOS) desenvolvido por Pandini (2012) e o implementado neste trabalho.

Os testes foram realizados em dois dispositivos físicos. O primeiro é o iPhone 4S com processador *Dual-core* modelo Cortex-A9 de 1 *Giga Hertz* (GHz), GPU PowerVR SGX543MP2, 512 *Mega Bytes* (MB) de RAM e resolução de 640x960 rodando iOS 6.1.3 (GSMARENA, 2013a). O segundo é o iPad com processador Cortex-A8 de 1GHz, GPU PowerVR SGX535, 256MB de RAM e resolução de 768x1924 rodando iOS 5.1.1 (GSMARENA, 2013b). Não foram realizados testes utilizando os simuladores disponibilizados pelo Xcode.

3.4.1.1 Desempenho dos métodos da biblioteca

Para medir o desempenho da biblioteca desenvolvida, foram monitorados os tempos de execução dos métodos responsáveis por realizar a movimentação do observador (`moveObserver:ToNewPointX:andNewPointY:updatingFrustum:`) e o método utilizado para verificar os pontos de interesse que estão no campo de visão do observador (`checkWaypoints:insideRooms:canBeSawByObserver:andFrustum:`).

Os testes foram baseados nos executados por Pandini (2012). Utilizou-se uma quantidade fixa de pontos de interesse e um caminho predefinido. A variação foi na quantidade de salas e portais presentes em cada mapa e nas configurações do campo de visão do observador. Para cada mapa testado, é calculado o tempo médio de execução dos métodos citados acima. Os testes seguem o seguinte roteiro:

a) é carregado o mapa;
b) o observador percorre um caminho predefinido pelo mapa;
c) quando o observador terminar de percorrer o caminho predefinido, é calculado o tempo médio dos métodos citados.

O cenários onde foram realizados os testes foram baseados nos cenários desenvolvidos por Pandini (2012). São quatro cenários, sendo que todos possuem 56 pontos de interesse nas mesmas coordenadas. O observador caminha por uma rota predeterminada e passa sempre pelos mesmos locais. O mapa da Figura 29 contém 9 salas e 12 portais, o da Figura 30 contém 7 salas e 10 portais, o da Figura 31 possui 5 salas e 8 portais e o último mapa, mostrado na Figura 32 possui 3 salas e 6 portais.

Figura 29 - Mapa de testes 1

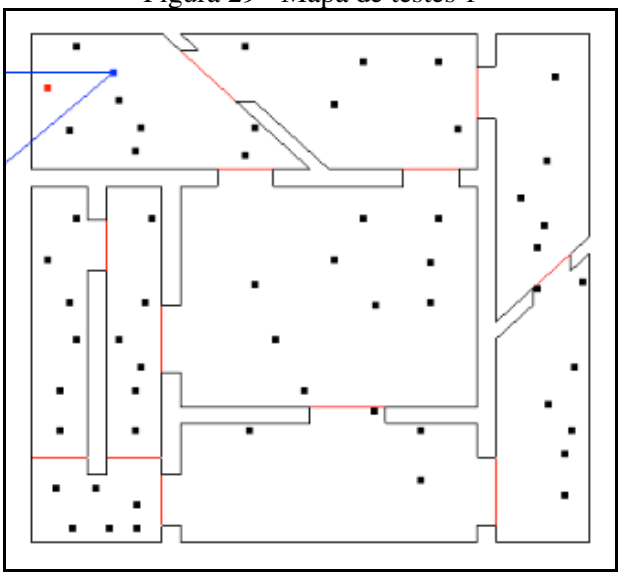


Figura 30 - Mapa de testes 2

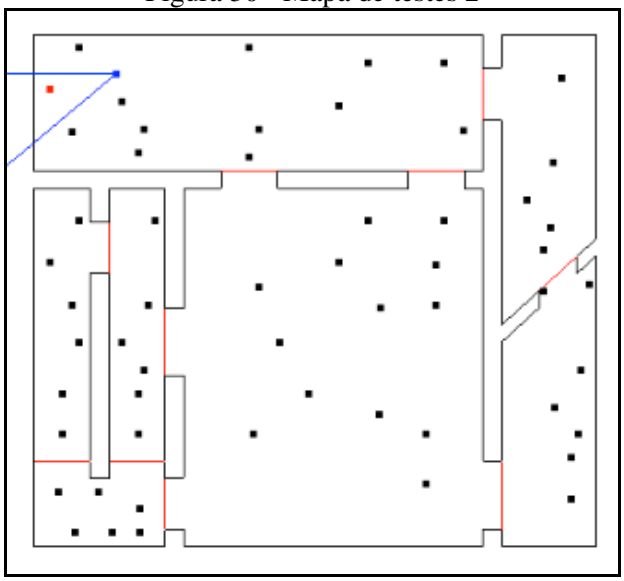


Figura 31 - Mapa de testes 3

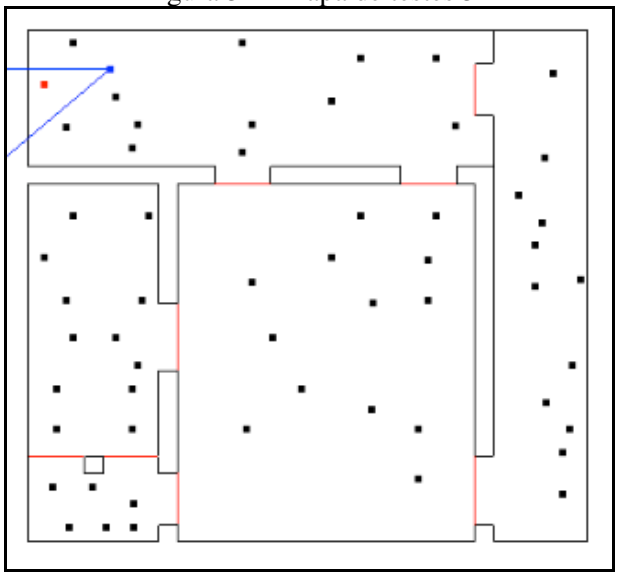
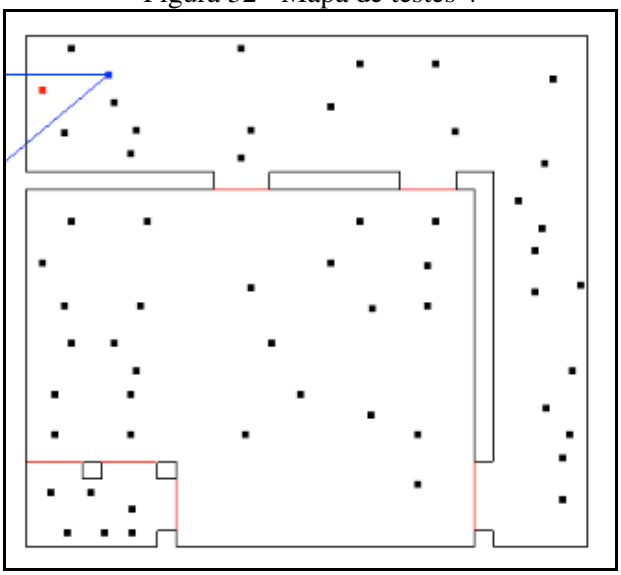


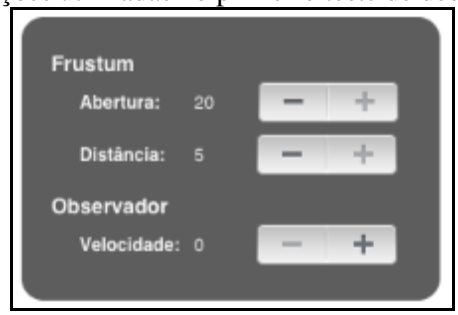
Figura 32 - Mapa de testes 4



Nos mapas acima, as paredes são representadas pelas linhas da cor preta. Os portais são as linhas da cor vermelha. Pontos de interesse são representados por pontos da cor preta. O observador e seu campo de visão são represetados pela cor azul.

O primeiro teste utilizou um campo de visão configurado com abertura 20 e distância 5. A Figura 33 mostra a configuração utilizada.

Figura 33 - Configurações utilizadas no primeiro teste de desempenho dos métodos



O Quadro 35 e Quadro 36 mostram o tempo médio de execução dos métodos em cada mapa, utilizando as configurações acima.

Quadro 35 - Tempo médio de execução dos métodos no iPad utilizando a primeira configuração

	Mapa 1	Mapa 2	Mapa 3	Mapa 4
Visão do observador	1,6943ms	1,8109ms	1,5841ms	1,4965ms
Mover observador	0,1310ms	0,1345ms	0,1326ms	0,1365ms

Quadro 36 - Tempo médio de execução dos métodos no iPhone utilizando a primeira configuração

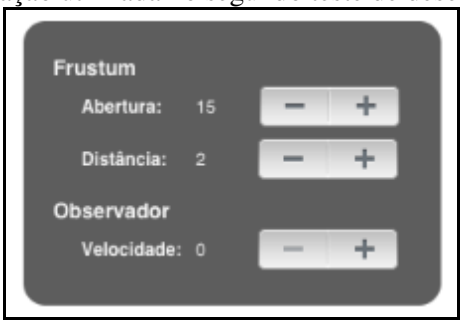
	Mapa 1	Mapa 2	Mapa 3	Mapa 4
Visão do observador	1,2447ms	1,4113ms	1,0848ms	1,5701ms
Mover observador	0,1437ms	0,0877ms	0,1850ms	0,1429ms

Conforme apresentado nos resultados acima, o método responsável por mover o observador teve um tempo de execução estável no dispositivo iPad e sofreu grande variação

no dispositivo iPhone. A visão do observador também sofreu uma variação de execução tanto no iPad quanto no iPhone. Com as configurações do observador iguais a Figura 33, nem sempre os mapas com mais salas tiveram um desempenho melhor. Isso ocorre porque o campo de visão é grande em relação ao mapa, fazendo com que vários portais fiquem no campo de visão do observador, obrigando a biblioteca a testar as várias salas que estão sendo vistas.

O segundo teste utilizou um campo de visão configurado com abertura 15 e distância 2. A Figura 34 mostra a configuração utilizada.

Figura 34 - Configuração utilizada no segundo teste de desempenho dos métodos



O Quadro 37 e Quadro 38 mostram o tempo médio de execução dos métodos em cada mapa, utilizando estas configurações.

Quadro 37 - Tempo médio de execução dos métodos no iPad utilizando a segunda configuração

	Mapa 1	Mapa 2	Mapa 3	Mapa 4
Visão do observador	0,8305ms	0,9291ms	0,8460ms	0,8923ms
Mover observador	0,1353ms	0,1482ms	0,1368ms	0,1368ms

Quadro 38 - Tempo médio de execução dos métodos no iPhone utilizando a segunda configuração

	Mapa 1	Mapa 2	Mapa 3	Mapa 4
Visão do observador	0,7155ms	0,8869ms	0,9440ms	1,0576ms
Mover observador	0,0767ms	0,1894ms	0,1803ms	0,1562ms

Nos resultados acima, o método responsável por mover o observador continuou tendo um tempo estável no dispositivo iPad e variando no dispositivo iPhone. Já o método responsável pela visão do observador mostrou-se mais eficiente em ambientes com mais salas. Como o campo de visão do observador nestes testes foi pequeno comparado ao tamanho do mapa, a quantidade de portais que o observador enxergava era menor, fazendo com que não fosse necessário testar vários portais e salas a cada execução.

3.4.1.2 Comparação de desempenho do método responsável pela visão do observador

Para que a biblioteca verifique se um ponto de interesse está dentro do campo de visão do observador, é necessário verificar se um ponto está dentro de um triângulo. Pandini (2012) utilizou um método baseado em coordenadas baricêntricas, diferente do que foi implementado

neste trabalho (ver seção 3.3.3). Para realzair estes testes, foi necessário converter o método desenvolvido por Pandini (2012) à plataforma iOS.

Foram realizados testes para comparar o desempenho desses dois métodos. O teste consiste em medir o tempo de execução do método responsável pela visão do observador. Como cenário principal, foi desenvolvido um mapa com quatro paredes e nenhum portal. Os métodos foram testados variando apenas a quantidade de pontos de interesse. Foram realizados testes com 1.000 e 10.000 pontos de interesse, sendo que o observador percorre o mesmo caminho em todos os testes. Como configuração do observador foi utilizada a mesma do primeiro teste (Figura 33). Os mapas utilizados podem ser vistos na Figura 35 e Figura 36.

Figura 35 - Mapa de testes 5

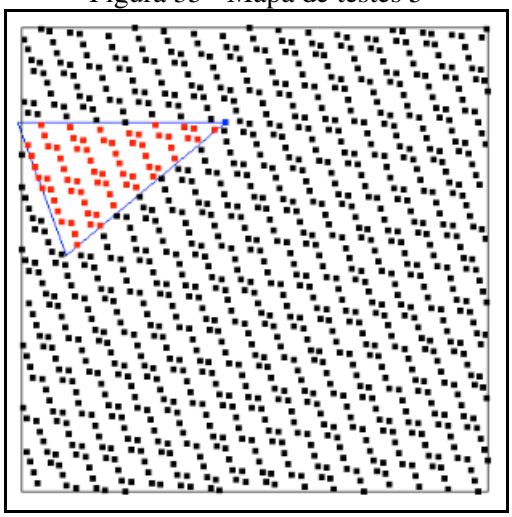
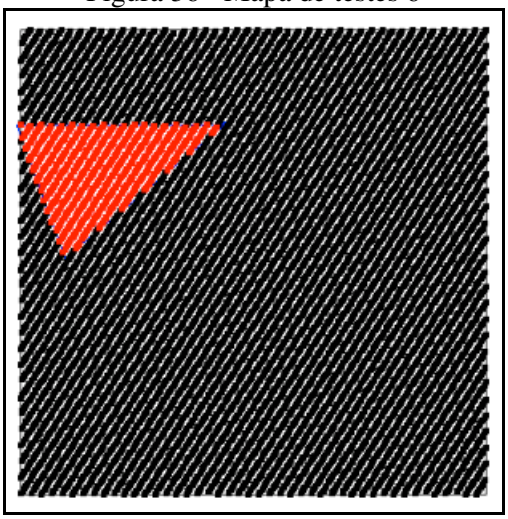


Figura 36 - Mapa de testes 6



A Figura 37 e Figura 38 mostram o gráfico de comparação do tempo médio de execução dos dois métodos nos dispositivos iPad e iPhone.

Figura 37 - Gráfico de comparação dos dois métodos no dispositivo iPad

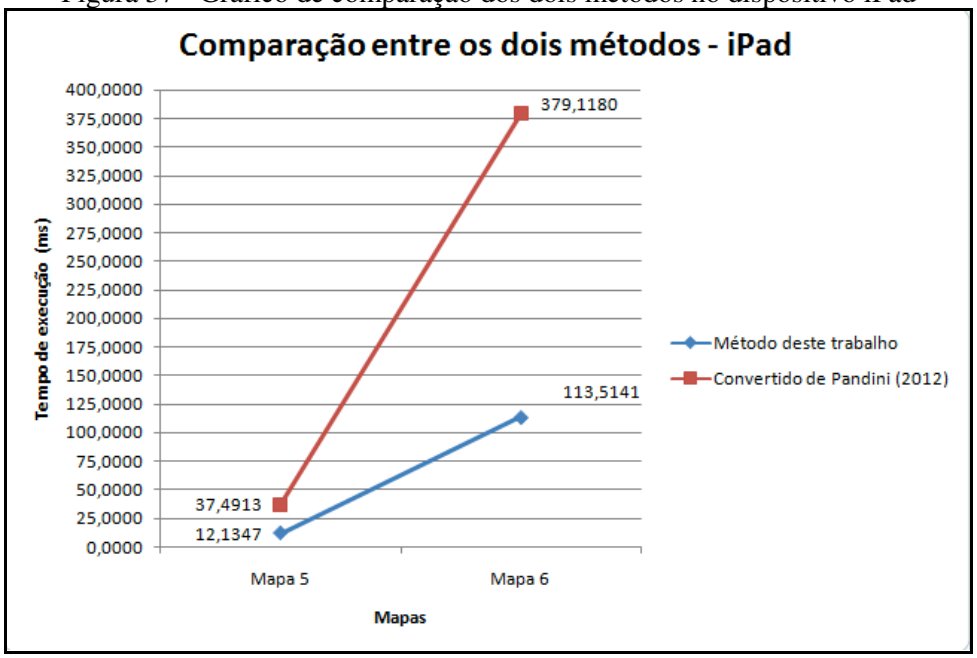
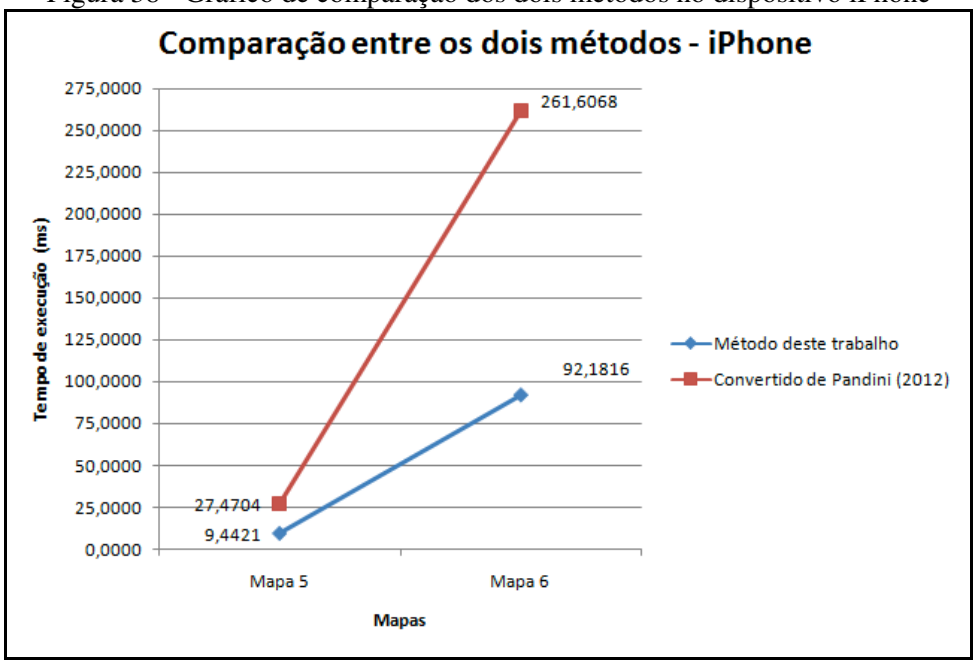


Figura 38 - Gráfico de comparação dos dois métodos no dispositivo iPhone



Nos gráficos acima, há uma comparação entre os dois métodos. Os tempos de execução do método desenvolvido neste trabalho são representados pelos pontos e linhas azuis. O desenvolvido por Pandini (2012) são representados pelos pontos e linhas vermelhas. Conforme observado, o método desenvolvido neste trabalho teve um ganho expressivo em tempo de execução em ambos os dispositivos.

3.4.1.3 Comparação com a biblioteca desenvolvida por Pandini (2012) e trabalhos correlatos

O presente trabalho teve como objetivo converter a biblioteca de algoritmos de portais em Android desenvolvida por Pandini (2012) para a plataforma iOS, permitindo a utilização

de cenários 3D (mantendo testes do algoritmo de portais em 2D) e testar outras formas de verificar se um ponto está dentro de um triângulo para melhorar o desempenho do teste de visibilidade. Os resultados obtidos ao término do trabalho são satisfatórios, a biblioteca desenvolvida conseguiu atingir os objetivos propostos.

No Quadro 39 é apresentada uma comparação entre a biblioteca desenvolvida neste trabalho, a biblioteca desenvolvida por Pandini (2012) e os trabalhos correlatos.

Quadro 39 - Comparativo das características da biblioteca desenvolvida com a biblioteca desenvolvida por Pandini (2012) e trabalhos correlatos

	Biblioteca desenvolvida	PANDINI (2012)	CARLOS; RAPOSO; SOARES (2010)	SILVA (2009)	SILVA et al. (2003)
Aceita modelos 3D	✓	⊘	✓	✓	✓
Importação de modelos	✓	⊘	✓	✓	✓
Executa em dispositivo móvel	✓	✓	⊘	✓	⊘
Algoritmo híbrido (GPU e CPU)	⊘	⊘	✓	⊘	⊘
Controle manual do observador	✓	⊘	⊘	✓	✓
Plataforma iOS	✓	⊘	⊘	⊘	⊘
Testes em 3D	⊘	⊘	✓	✓	✓

O trabalho de Carlos, Raposo e Soares (2010) foi o único entre os comparados que utilizou algoritmos híbridos que executam em GPU e CPU. Já o trabalho de Pandini (2012) foi o único que não aceita importar modelos de mapas. A biblioteca desenvolvida, o trabalho de Pandini (2012) e o de Silva (2009) rodam em dispositivos móveis. A biblioteca desenvolvida também é a única que roda na plataforma iOS. O controle manual do observador está presente na biblioteca desenvolvida, no trabalho de Silva (2009) e Silva et al. (2003). Por fim, as bibliotecas que realizam testes em 3D são as de Carlos, Raposo e Soares (2010), Silva (2009) e Silva et al. (2003).

Pode-se considerar que a biblioteca desenvolvida, nas características apresentadas, apresentou um melhor resultado em relação a biblioteca de Pandini (2012) e um bom resultado em relação aos trabalhos correlatos. A ferramenta desenvolvida destaca-se pela plataforma para qual foi desenvolvida, sendo a única entre as comparadas com tal característica.

4 CONCLUSÕES

O trabalho aqui apresentado discorre sobre o desenvolvimento de uma biblioteca de algoritmos de portais para a plataforma iOS. O estudo das técnicas e algoritmos de visibilidade possibilitou um melhor entendimento de como funcionam estas técnicas. Foi este estudo que possibilitou o entendimento do algoritmo de portais e contribuiu com o desenvolvimento deste trabalho, atendendo um dos objetivos iniciais.

Outro objetivo atendido refere-se à utilização de cenários 3D. Para que este objetivo fosse atendido, foi necessário obter conhecimento na linguagem de programação Objective-C e da biblioteca gráfica OPENGL ES, a fim de utilizar e modificar a biblioteca desenvolvida por Imianowsky (2013), resultando na possibilidade de utilização de cenários em 3D pela biblioteca. O estudo da linguagem de programação Objective-C ajudou também na conversão e criação das várias classes que compõe a biblioteca.

A utilização da biblioteca de Pandini (2012) como referência para a criação de uma biblioteca para a plataforma iOS, mostrou ser um ponto positivo tendo sido utilizada como referência em grande parte do desenvolvimento deste trabalho. A técnica empregada por Pandini (2012) para verificar se um ponto estava contido em um triângulo foi utilizada como base para a busca de novas técnicas que resolvessem esse problema. A nova técnica encontrada mostrou ter um desempenho superior à desenvolvida por Pandini (2012).

Durante o desenvolvimento da biblioteca, encontraram-se alguns problemas na conversão da biblioteca, sendo necessário encontrar outras soluções para resolver o mesmo problema.

A biblioteca desenvolvida apresentou algumas limitações, como por exemplo a obrigatoriedade na utilização da ferramenta de modelagem Blender na criação dos mapas. Além da limitação na criação dos mapas, a biblioteca apresentou uma limitação na quantidade de portais (que apontam para a mesma sala) vistos pelo observador. Em locais do mapa onde mais de um portal que apontam para a mesma sala são vistos, a biblioteca não mostrou-se capaz de reconhecer todos os pontos de interesse que estavam no campo de visão do observador. Entretanto, como situações iguais a essa não são comuns, a biblioteca mostrou ter resultados que permitissem atingir os objetivos propostos.

4.1 EXTENSÕES

Alguns pontos da biblioteca podem ser melhorados. Como sugestão para trabalhos futuros, citam-se os seguintes itens:

a) implementar os testes de visibilidade em 3D;

b) possibilitar a utilização de mais de um observador no mapa;
c) implementar os testes de visibilidade com objetos e não pontos;
d) tratar para que o observador consiga enxergar mais de um portal que aponte para a mesma sala;
e) possibilitar a utilização de outras ferramentas de modelagem 3D para criação de mapas.

REFERÊNCIAS BIBLIOGRÁFICAS

AGUIARI, Vinicius. **Venda de tablets deve crescer 98% em 2012.** [S.l.], 2012. Disponível em: <http://info.abril.com.br/noticias/mercado/venda-de-tablets-deve-crescer-98-em-2012-11042012-11.shl>. Acesso em: 8 set. 2012.

APPLE INC. **About iOS development.** [S.l.], 2012. Disponível em: <http://developer.apple.com/library/ios/#Documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/IPhoneOSOverview/IPhoneOSOverview.html>. Acesso em: 8 set. 2012.

BLACKPAWN. **Point in triangle test.** [S.l.], 2011. Disponível em: <http://www.blackpawn.com/texts/pointinpoly/default.html>. Acesso em: 18 maio 2013.

BLENDER. **Add-Ons.** [S.l.], 2013. Disponível em: <http://wiki.blender.org/index.php/Doc:2.6/Manual/Extensions/Python/Add-Ons>. Acesso em: 01 jun. 2013.

BOE, Bryce. **Line segment intersection algorithm.** [S.l.], 2006. Disponível em: <http://www.bryceboe.com/2006/10/23/line-segment-intersection-algorithm/> Acesso em: 18 maio 2013.

CARLOS. Eduardo T. **Frustum culling híbrido utilizando CPU e GPU.** 2009, 99 f. Dissertação (Mestrado em Informática) – Departamento de Informática, Pontífica Universidade Católica do Rio de Janeiro, Rio de Janeiro.

CARLOS, Eduardo T.; RAPOSO, Alberto; SOARES, Luciano. Frustum culling híbrido em CPU e GPU para visualização de modelos massivos em tempo real. In: SIMPOSIUM ON VIRTUAL AND AUGMENTED REALITY SVR 2010, 12., 2010, Natal. **Proceedings...** Porto Alegre: SBC, 2010. p. 174-183.

FINLEY, Darel R. **Point-in-polygon algorithm —** determining whether a point is inside a complex polygon. [S.l], [2007?]. Disponível em: <http://alienryderflex.com/polygon/>. Acesso em: 18 maio 2013.

GAME RENDERING. **Basic culling techniques.** [S.l.], 2008. Disponível em: <http://www.gamerendering.com/2008/11/02/basic-culling-techniques/>. Acesso em: 7 set. 2012.

GSMARENA. **Apple iPhone 4S**. [S.l.], 2013a. Disponível em: <http://www.gsmarena.com/apple_iphone_4s-4212.php>. Acesso em: 09 jun. 2013.

_____. **Apple iPad Wi-Fi.** [S.l.], 2013b. Disponível em: <http://www.gsmarena.com/apple_ipad_wi_fi-3828.php>. Acesso em: 09 jun. 2013.

HADWIGER, Markus; VARGAS, Andreas. **Visibility culling.** [S.l.], [1998?]. Disponível em: <http://www.cg.tuwien.ac.at/~msh/viscull.pdf>. Acesso em: 8 set. 2012.

IMIANOWSKY, Felipe A. **iOBJ**. Blumenau, 2013. Disponível em: <https://github.com/felipowsky/iOBJ>. Acesso em: 10 abr. 2013.

PANDINI, Ana P. **Biblioteca de algoritmos de portais para a plataforma Android.** 2012. 50 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

PETRY, Rodrigo. **Vendas de smartphones devem crescer 73% em 2012, aponta IDC.** [S.l.], 2012. Disponível em: <http://economia.estadao.com.br/noticias/economia,vendas-de-smartphones-devem-crescer-73-em-2012-aponta-idc,106637,0.htm>. Acesso em: 7 set. 2012.

PULLI, Kari et al. **Mobile 3D graphics:** with OpenGL ES and M3G. Burlington: Elsevier, 2008.

SADUN, Erica. **The iPhone™ developer's cookbook:** building applications with the iPhone 3.0 SDK. 2nd ed. Boston: Addison-Wesley Professional, 2009.

SILVA, Romano J. M. et al. Experiência de portais em ambientes arquitetônicos virtuais. In: SYMPOSIUM ON VIRTUAL REALITY - SVR 2003, 6., 2003, Ribeirão Preto. **Anais...** Ribeirão Preto: Creto Vidal e C. Kirner, 2003. p. 117-128.

SILVA, Wendel B. **Renderização iterativa em dispositivos móveis utilizando algoritmos de visibilidade e estruturas de particionamento espacial.** 2008. 109 f. Dissertação (Mestrado em Informática) – Curso de Mestrado em Informática Aplicada, Universidade de Fortaleza – UNIFOR, Fortaleza.

SILVA, Wendel B. Renderização iterativa em dispositivos móveis utilizando algoritmos de visibilidade e estruturas de particionamento espacial. In: CONCURSO DE TESES E DISSERTAÇÕES, 22., [2009], Fortaleza. **Anais...** Porto Alegre: SBC, 2009. p. 121-128.

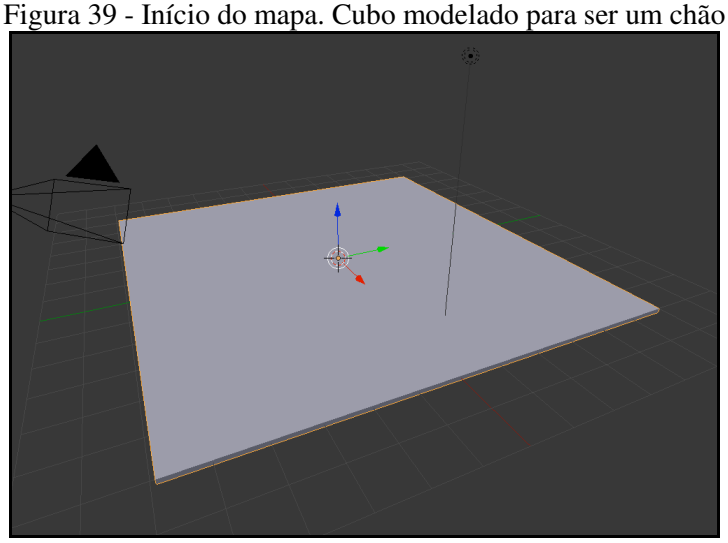
SUGRUE, James. **Building iOS 5 games:** develop and design. Berkeley: Peachpit Press, 2011.

TAVARES, Denison L. M. **Aproximação eficiente de visibilidade para nuvem de pontos utilizando GPU**. 2009. 73 f. Dissertação (Mestrado em Ciência da Computação) – Programa de Pós-Graduação em Computação, Universidade Federal do Rio Grande do Sul, Porto Alegre.

APÊNDICE A – Criar mapa no Blender

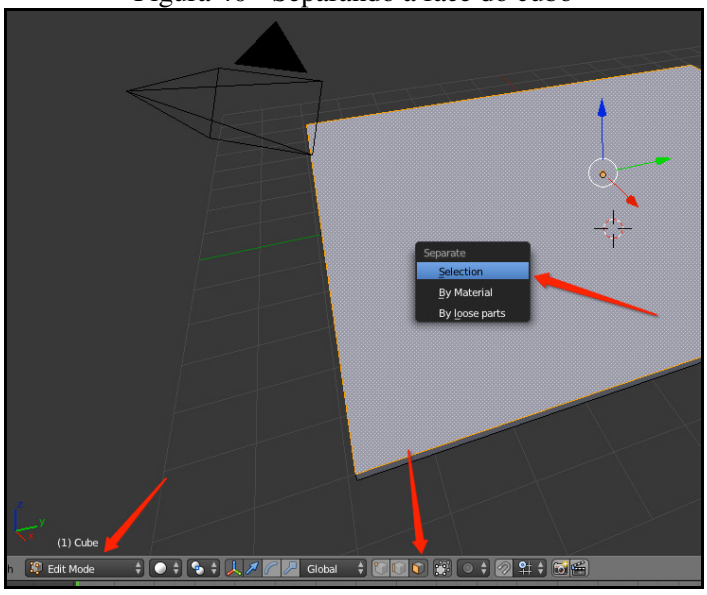
Para criar um mapa no Blender primeiramente deve-se inicializar um novo projeto contendo apenas um cubo e modelar o mesmo para que se pareça com um chão. A Figura 39 demonstra este passo.

Figura 39 - Início do mapa. Cubo modelado para ser um chão



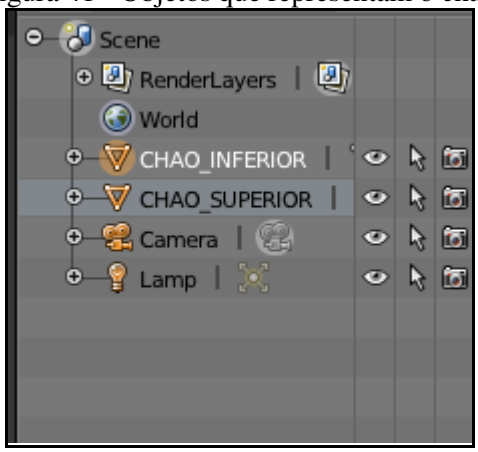
Após modelar o cubo o usuário deve separar sua face superior. Para isso é necessário entrar no modo de edição (`Edit Mode`), alterar para o modo de seleção de faces (`Face Select`), selecionar a parte superior do cubo clicando com o botão esquerdo do mouse sobre ela e separar a face (pressionar a tecla `P` do teclado) clicando na opção `Selection`. A Figura 40 mostra este passo.

Figura 40 - Separando a face do cubo



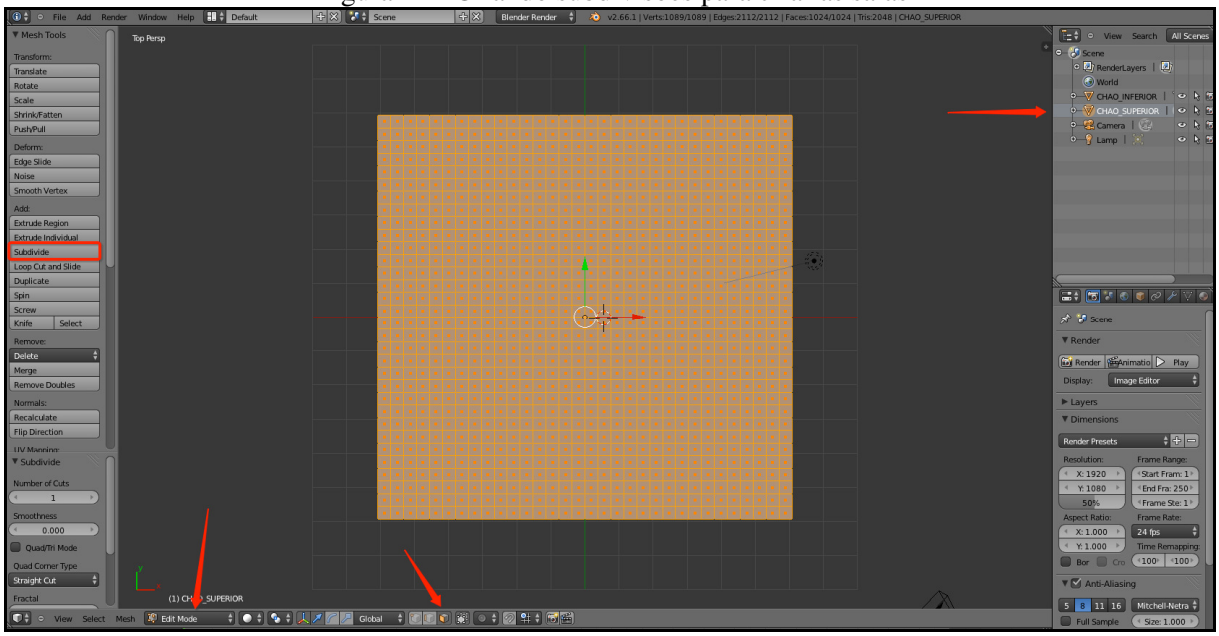
Após realizar estes passo, será criado um novo objeto. Este objeto deve ser renomeado para `CHAO_SUPERIOR`. O objeto que deu origem a este deve ser renomeado para `CHAO_INFERIOR`. A Figura 41 ilustra esse passo.

Figura 41 - Objetos que representam o chão



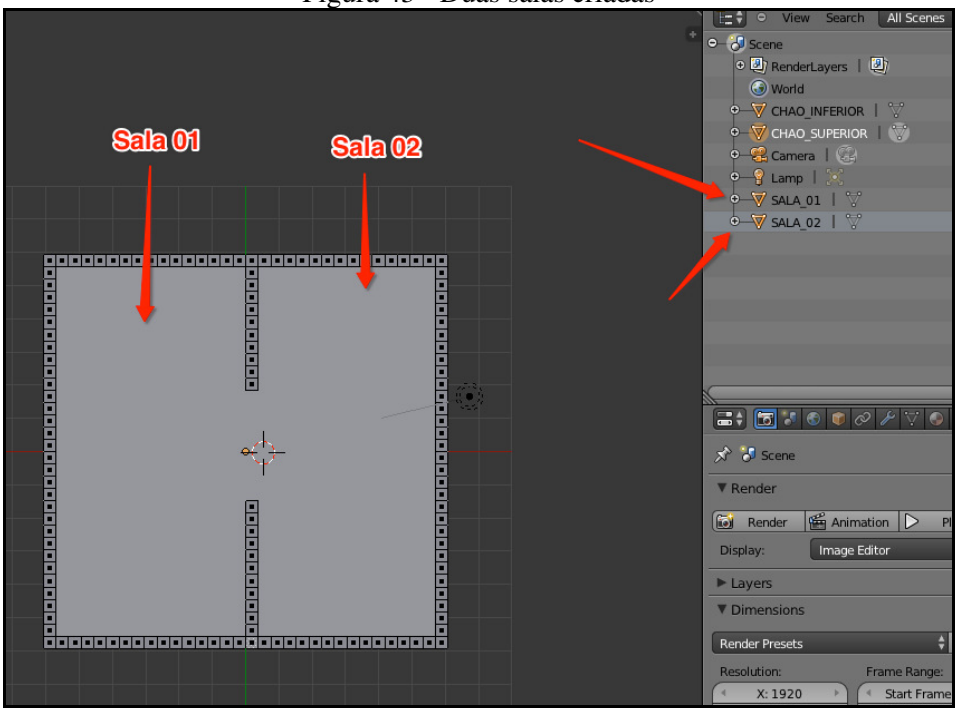
Após isso, é necessário criar as salas. Para isso deve-se selecionar o objeto `CHAO_SUPERIOR`, entrar em modo de edição e seleção de faces e aplicar o comando `Subdivide`. A Figura 42 mostra o resultado.

Figura 42 - Criando subdivisões para criar as salas



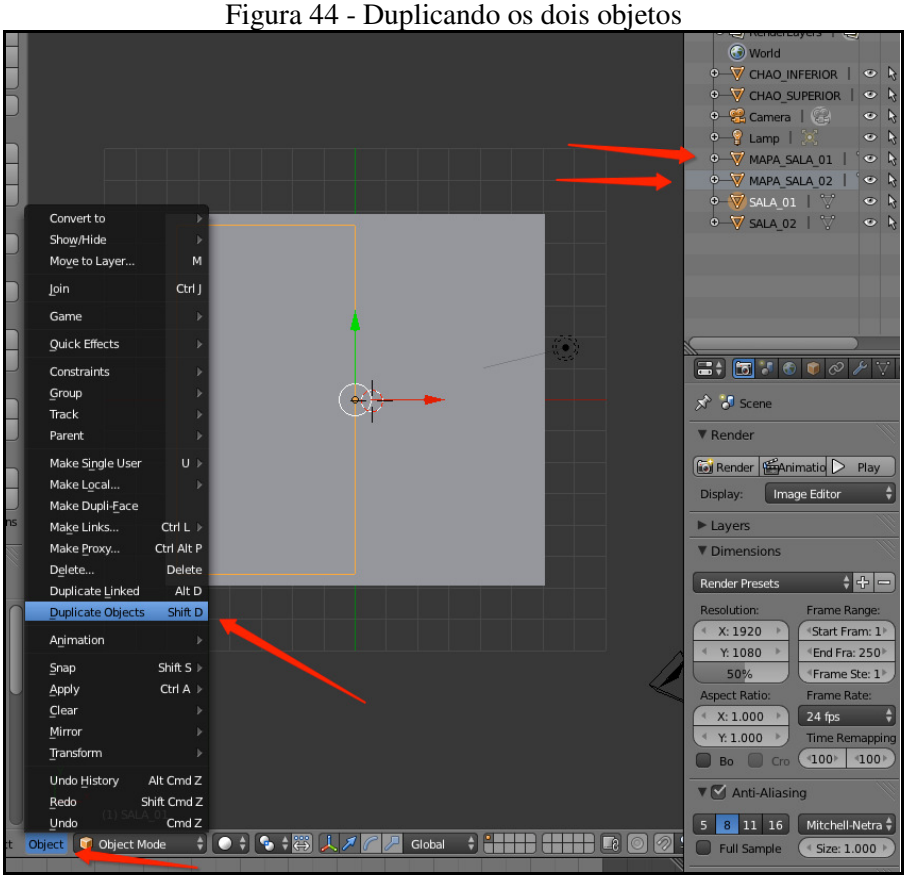
Para cada sala que será criada deve ser transformada em um novo objeto. Deve-se selecionar as faces que vão compor a sala e separá-las em um novo objeto (da mesma forma que foi separada a face superior do cubo). Cada objeto novo desse deve possuir um prefixo `SALA_` e um número sequencial. A Figura 43 mostra duas salas criadas.

Figura 43 - Duas salas criadas



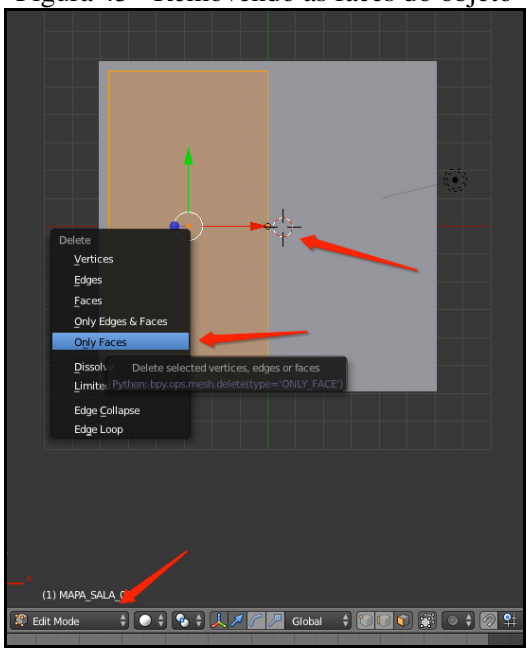
Os novos objetos devem ser duplicados e em cada objeto duplicado, adicionado o prefixo `MAPA_`. Para duplicar um objeto, é necessário selecionar o mesmo e ir no menu `Object`, `Duplicate Objects`. A Figura 44 mostra o resultado da duplicação dos objetos.

Figura 44 - Duplicando os dois objetos



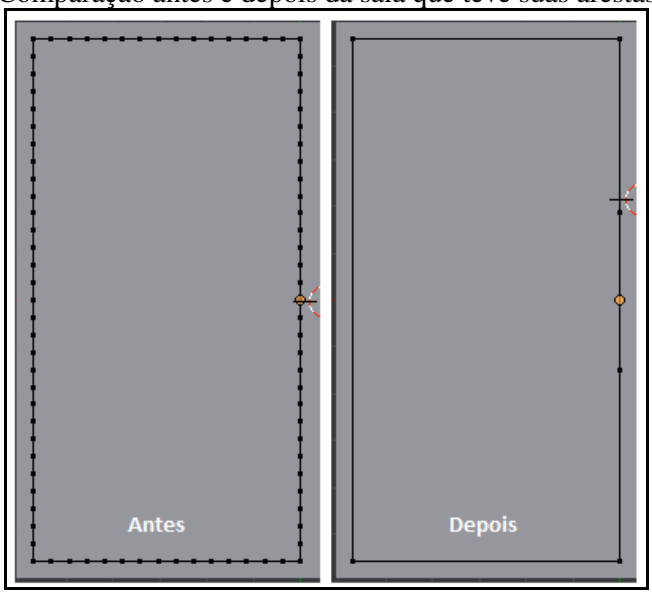
Após duplicar os objetos, é necessário remover suas faces e juntar as arestas. Para remover as faces, deve-se selecionar o objeto, entrar em modo de edição (`Edit Mode`), selecionar as faces e clicar na tecla `F`. Com isso todas as faces se transformarão em uma só. Após isso, deve-se clicar na tecla `X` e remover apenas as faces clicando na opção `Only Faces`. A Figura 45 mostra o resultado.

Figura 45 - Removendo as faces do objeto



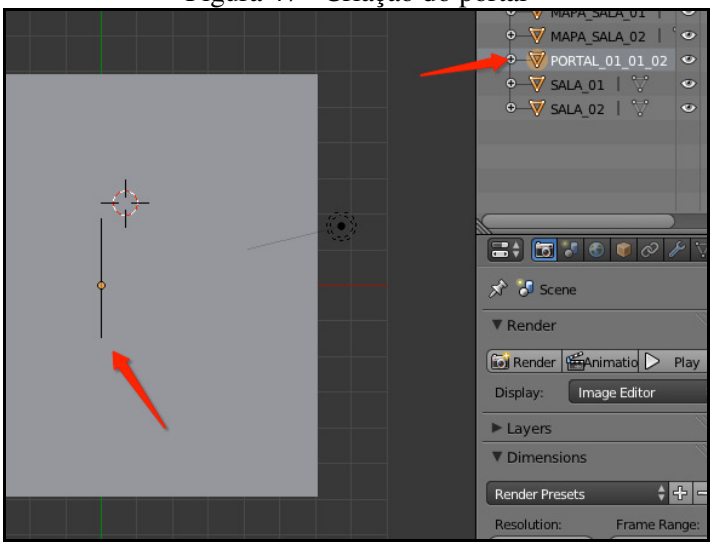
Após remover as faces dos objetos com prefixo `MAPA_` deve-se juntar as arestas diminuindo a quantidade de vértices. Para entrar no modo de seleção de vértices, deve-se selecionar a opção `Vertex select` e selecionar os vértices que serão ligados. Os mapas devem conter vértices apenas em locais onde há uma curva (lados da sala). Para realizar a junção de vários vértices é necessário utilizar o comando `Merge`. O comando `Merge` pode ser acessado através do menu `Mesh`, `Vértices`, opção `Merge`. A Figura 46 mostra o antes e depois de uma sala que teve suas arestas diminuidas.

Figura 46 - Comparação antes e depois da sala que teve suas arestas diminuídas



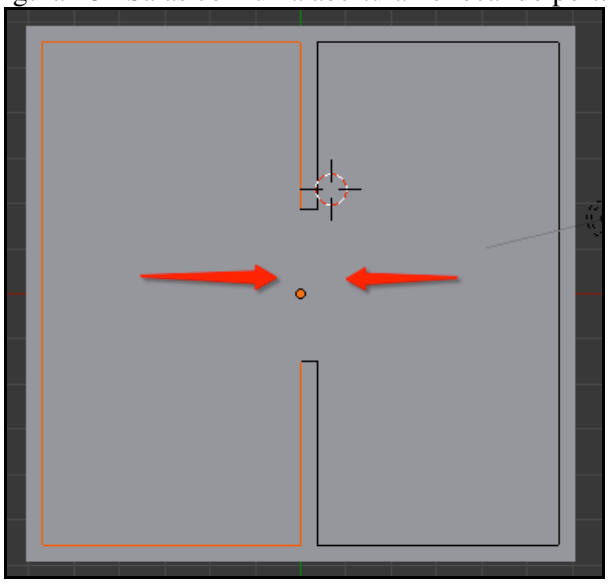
Após remover as arestas desnecessárias é preciso criar um portal que ligue as duas salas. Um portal é representado por uma aresta. Deve-se selecionar a aresta que representará o portal e separar em um novo objeto. Este novo objeto deve possuir o nome que identifique o portal e as salas que são ligadas por ele. A Figura 47 mostra um portal criado.

Figura 47 - Criação do portal



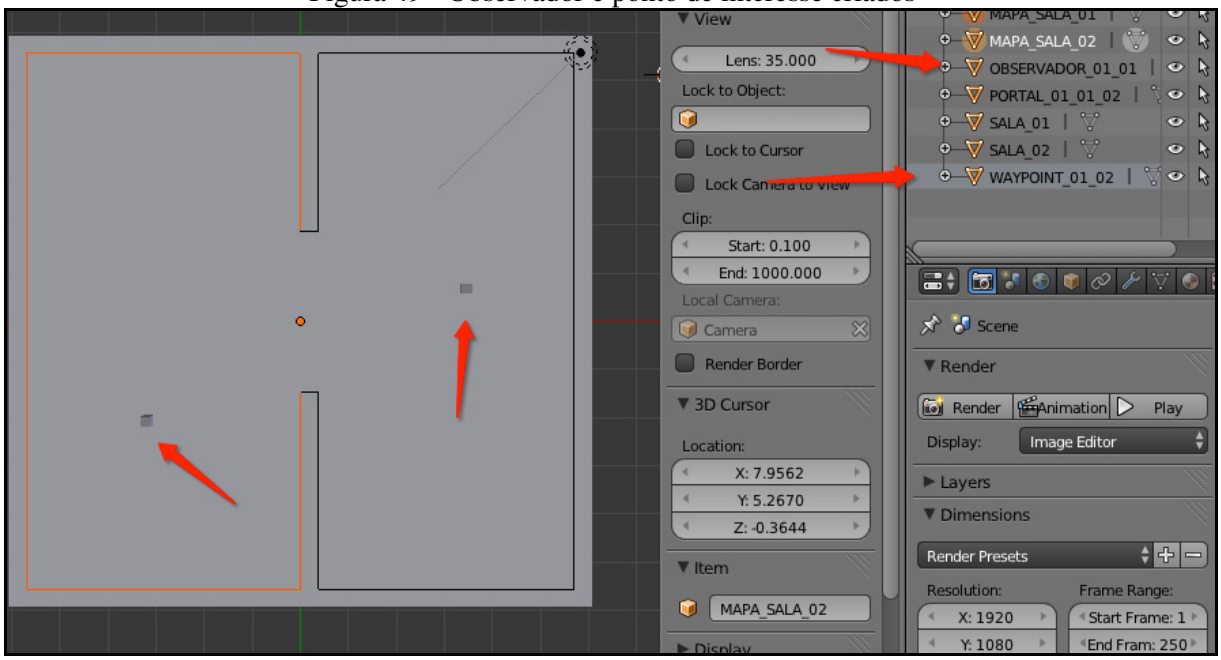
As salas que são ligadas pelo portal devem possuir uma abertura (onde o portal ficará). A Figura 48 demonstra esta abertura.

Figura 48 - Salas com uma abertura no local do portal



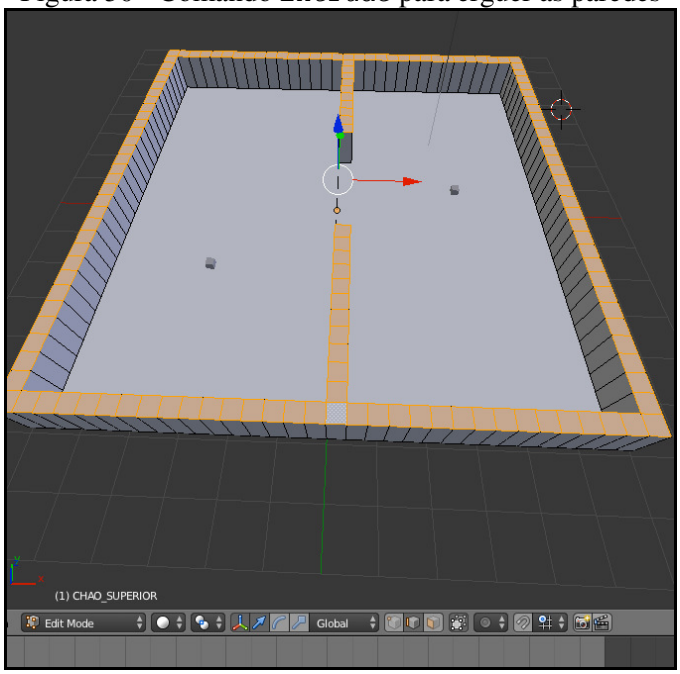
Após estes passos é necessário apenas adicionar o observador e os pontos de interesses. Ambos são representados por um objeto do tipo cubo. A Figura 49 mostra dois objetos (observador e um ponto de interesse) criados.

Figura 49 - Observador e ponto de interesse criados



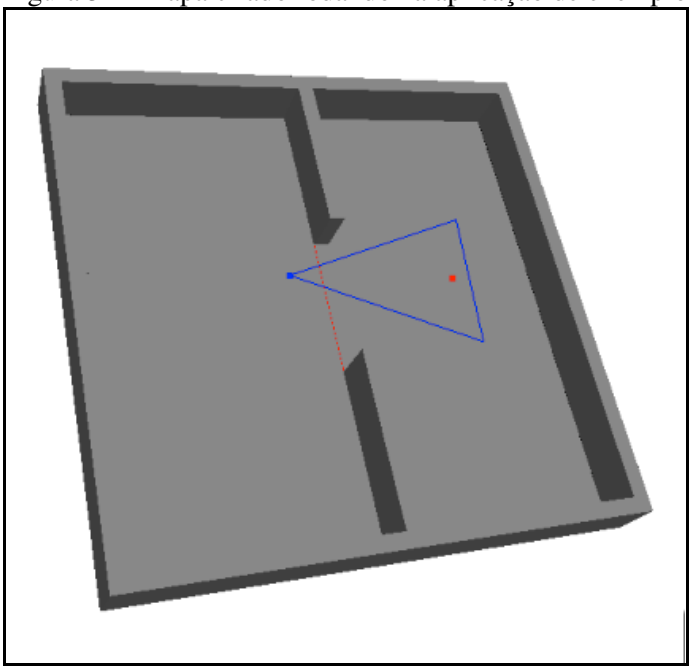
Após criar todos os objetos que compõe o mapa é necessário realizar o comando `Extrude` no objeto `CHAO_SUPERIOR`. Esse comando fará com que as paredes do modelo 3D sejam criadas. A Figura 50 mostra o resultado.

Figura 50 - Comando `Extrude` para erguer as paredes



A Figura 51 mostra o mapa sendo executado na aplicação de exemplo.

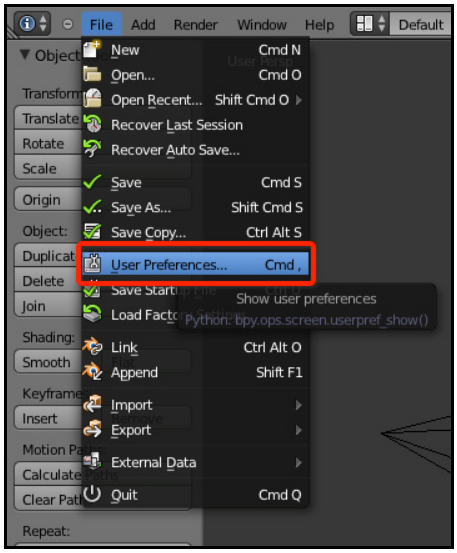
Figura 51 - Mapa criado rodando na aplicação de exemplo



APÊNDICE B – Instalar o *add-on* no Blender

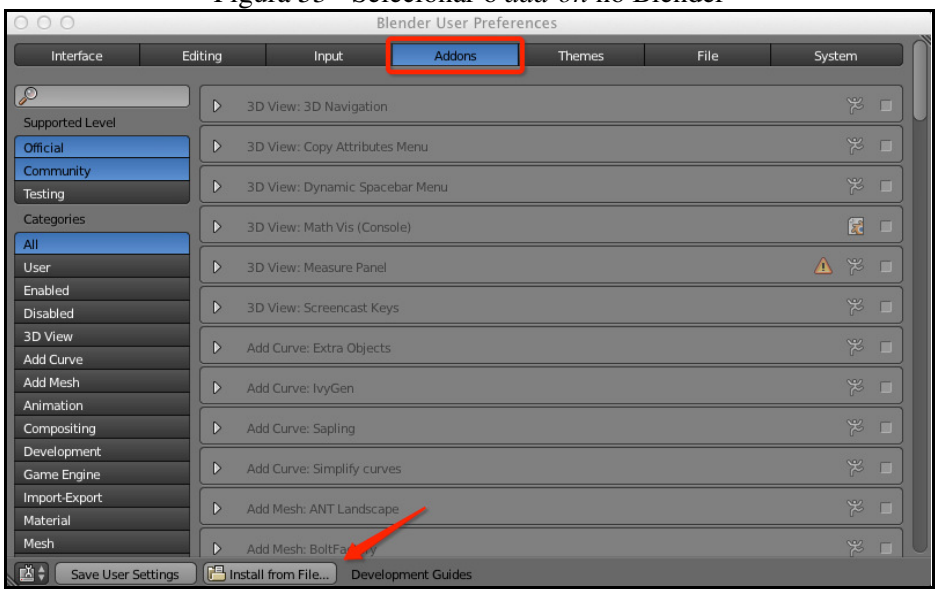
Para instalar o *add-on* desenvolvido no Blender é necessário entrar nas preferências do usuário. A Figura 52 ilustra este passo.

Figura 52 - Preferências do usuário no Blender



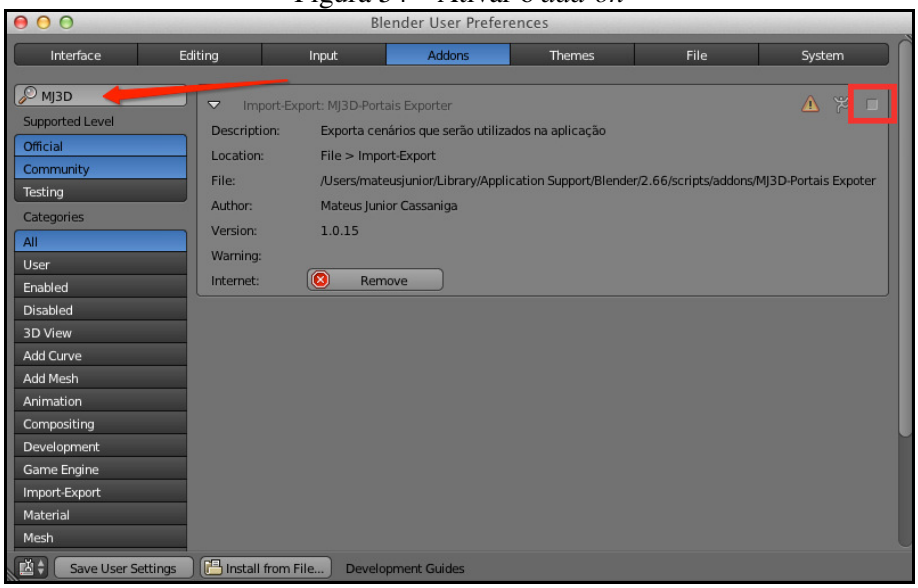
Após abrir as preferências do usuário é necesrário selecionar o arquivo contendo o *add-on* desenvolvido. Para isso, deve-se selecionar a aba `Addons` e clicar em `Install from File...` A Figura 53 demonstra esse passo.

Figura 53 - Selecionar o *add-on* no Blender



Após selecionar o arquivo contendo o *add-on*, é preciso ativá-lo. Para isso é necessário selecionar o *add-on* (`MJ3D Portais`) e ativar o mesmo clicando na caixa de seleção contornada com o quadrado vermelho, conforme Figura 54.

Figura 54 – Ativar o *add-on*



Após realizar estes passos, o *add-on* ficará disponível no menu de exportação. A Figura 55 mostra o resultado.

Figura 55 - *Add-on* instalado