Universidade Federal de Pernambuco

Centro de Informática

REVOLUTION ENGINE

ARQUITETURA DE UM MOTOR 3D PARA JOGOS

Aluno: Marco Túlio C. F. Albuquerque (mtcfa@cin.ufpe.br)
Orientador: Silvio de Barros Melo (sbm@cin.ufpe.br)

Recife, vinte e um de julho de 2005.

Dedicatória

Dedico este trabalho em sua grande parte a minha família: mãe, pai e irmãos. Pois, além de me darem todas as oportunidades sempre me apoiaram e aceitaram minhas escolhas mesmo quando não entendiam.

Agradecimentos

Primeiramente a minha mãe, Ligia por ter sido sempre tão compreensiva e dado todo tipo de apoio necessário durante todos esses anos. Agradecer também ao meu pai, Marco Túlio, por criar as todas as oportunidades que tive em minha vida e pela preocupação com meu futuro. Sem esquecer dos meus irmãos, Adriana e Bruno por estarem sempre ao meu lado.

À Carol, amiga e namorada a mais de cinco anos. Sempre companheira e que sempre falou o que eu precisava ouvir sem se importar como eu iria reagir. Sempre entendeu, mais do que a maioria, os horários e a dificuldade de estudar no Centro de Informática. Além de ter revisado, jornalista que é, diversos relatórios, textos e artigos meus e de meus colegas.

Agradeço aos Joeys: Felipe, Thiago e Duda. Sempre foram a melhor forma de diversão e distração de todo tipo de problema que precisei enfrentar, de sempre trazerem alegria e serem amigos em todos os momentos, bons ou ruins. Enfim, por terem dividido todas as experiências que vivemos e rachado todos aqueles churrasquinhos.

A todos os meus amigos de faculdade, principalmente os colegas mais próximos, o grupo de projetos: André, Afonso, Vicente, Cabelinho, Vilmar, Ives, mais recentemente Gênio, ao resto do Oxente e a Marco Aurélio por tanto tempo de trabalho na iniciação científica. Pois afinal foram as pessoas que estavam comigo nas aulas, nos projetos, nas madrugadas sem dormir, nos estágios, convivendo quase que diariamente durante toda a graduação.

Agradecer especialmente a Ives. Importante durante todo este trabalho e grande incentivador da pesquisa em computação gráfica no Cin. Pelas diversas discussões que tivemos sobre todos os aspectos da computação gráfica. E principalmente por que ele também me agradeceu.

Ao professor Silvio pelo suporte. Por estar sempre disposto a ajudar no que for preciso, mesmo quando ainda fazia parte do Departamento de Matemática. Além de suprir a necessidade de um orientador na área da computação gráfica tão adorada por diversos alunos.

Gostaria de agradecer a diversos professores do Centro de Informática, especialmente Chico, Aluízio e Geber. Pela formação recebida e pela orientação nos diversos projetos durante cadeiras e iniciação científica.

Agradecer, em especial, a dois tios Fúlvio e Luciano. Fúlvio por ter me ensinado tanto sobre programação durante um estágio em sua empresa antes mesmo de iniciar a faculdade. Luciano por ter me abrigado na sua casa e ajudado bastante durante o tempo que passei em Nova York. E por despertarem todo meu fascínio por computação, tecnologia e ficção científica em geral.

Agradecer ao meu colégio de tantos anos, o CPI. Por ter me dado uma formação boa além de tantos amigos. Especialmente ao professor André Paegle, pelo conhecimento passado, a amizade criada e a consciência, de ensinar o assunto e não a fazer prova de vestibular, tão incomum em professores de ensino médio nos dias de hoje.

Gostaria de agradecer a todos aqueles que participaram de forma positiva na minha formação. Aqueles que conheci no CPI, no Centro de Informática ou no CESAR pela amizade e companheirismo.

Queria expressar também meu muito obrigado ao Posto Br que fica ao lado da reitoria da Universidade e sua loja de conveniência Br Mania por estarem aberto 24 horas e servir o melhor bolovo[1] da cidade.

[1] Bolovo é originalmente o bolinho de carne moída, ovo e massa. Seu uso no cotidiano indica todos aqueles salgados de massa, duros que pingam óleo no papel.

Histórico de Revisões

Versão	Data	Alteração
0.1	21 de julho	Criação dos tópicos do documento e do Histórico de Revisões
0.1b	01 de agosto	Criação da capa e do índice do documento.
0.1c	02 de agosto	Texto do capítulo 1. Introdução e Inserção da Seção Referências.
0.1d	02 de agosto	Atualização das referências bibliográficas, reestruturação das seções, criação do capítulo 2. Motores Gráficos
0.2	03 de agosto	Texto do capítulo 2.
0.3	12 de agosto	Inserção dos diagramas de classe e da lista dos casos de uso
0.3b	13 de agosto	Texto e figuras do capítulo 3. Exceto seção Concorrência.
0.3c	14 de agosto	Finalização da Seção Análise do capítulo 4
0.3r	15 de agosto	Tabela de Resultados e Lista de Tabelas
0.5	17 de agosto	Refinamento dos casos de uso e modelo conceitual. Revisão final e correções do capítulo 2.
0.6	18 de agosto	Formatação segundo algumas regras da ABNT e de TG descritas em http://www.cin.ufpe.br/~tg
0.6a	18 de agosto	Numeração das páginas
0.9	19 de agosto	Diagramas de colaboração
1.0	19 de agosto	Resumo e finalização do documento

Resumo

Diversos motores gráficos para jogos têm sido criados com propósitos comerciais e também por comunidades Open Source. Apesar do grande número de sistemas e de muita pesquisa na área, não existe uma arquitetura padrão devido, principalmente, aos requisitos que diferem para cada tipo de jogo.

Entendendo a complexidade do problema e a diversidade de possibilidades para um motor gráfico, este trabalho apresenta uma arquitetura para motores de alto nível que possa ser refinada para adequar-se aos mais diversos tipos de motor e portável para os diversos ambientes de execução de jogos.

Sumário

Lista de Figuras

Lista de Tabelas

1

2 Introdução

O mercado de jogos eletrônicos possui, na indústria de entretenimento, a maior taxa de crescimento nos últimos anos. Segundo a Screen Digest7 entre 1997 e 2003 o mercado de software de entretenimento teve um crescimento de 100%. Desde 1995, mais de três bilhões e meio de jogos foram vendidos, culminando em um faturamento de 18.2 bilhões de dólares em 2003. Segundo Roger Bennett, diretor geral da Entertainment and Leisure Software Publishers Association (ELSPA), *"A indústria de jogos continua a demonstrar a mais dinâmica taxa de crescimento de todas as indústrias de criatividade. É gratificante ter prova da crescente popularidade do entretenimento interativo e de sua abrangente base de clientes*".

O crescimento da indústria transformou a maneira com que os jogos são criados. No inicio, os jogos eram criados por hobistas (hobbyist) e normalmente o software inteiro era feito pela mesma pessoa. Nos primeiros anos da década de 1990, com o crescimento da indústria, começaram a surgir empresas especializadas na área: as desenvolvedoras de jogos e os publishers[2].

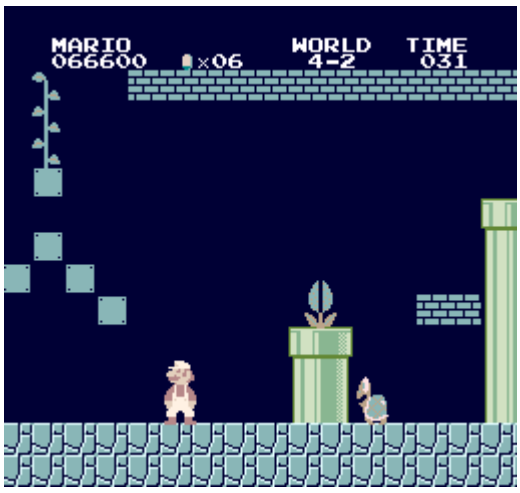

Figura 1 - Mario Brothers©, jogo da Nintendo. Falta realismo e imersão.

A capacidade computacional limitada dos computadores e consoles[3] criavam limites na criação de jogos, os quais, em sua maioria, eram do tipo plataforma em duas

[2] Responsável pelo marketing – promoção, estratégia de vendas, distribuição, etc. - do jogo, papel semelhante ao de uma editora na publicação de livros.

[3] Sistema criado para rodar jogos eletrônicos. Também conhecido como vídeo game.

dimensões. Esses jogos tinham como fator principal sua jogabilidade[4] e deixava de lado, por motivos tecnológicos, conceitos como imersão e realismo.

Em dezembro de 1993, a desenvolvedora id Software© lançou Doom7. Doom era um ótimo jogo e tinha um diferencial tecnológico: o motor (engine). A criação do motor de jogos foi crucial para o desenvolvimento e crescimento da indústria. O motor era um grande framework e concentrava os conceitos independentes do jogo - gráficos, som, rede, entre outros – aumentando consideravelmente o grau de reuso e a portabilidade por conseqüência a qualidade e desempenho dos jogos diminuindo custos e a quantidade de bugs.

Vários jogos, entre eles CounterStrike, Team Fortress, TacOps e Strike Force, foram criados utilizando o motor de Doom que evoluiu bastante nos últimos doze anos.

A evolução das técnicas de computação gráfica possibilitou a utilização de gráficos em três dimensões trazendo imersão e realismo para os jogos, agregando valor aos produtos e criando um novo mercado. Ao mesmo tempo, os requisitos do motor mudam com a evolução das plataformas e a necessidade da redução de custos.

Figura 2 – Doom© em 1993 e Doom3 em 2003. Evolução do motor.

O motor gráfico é o artefato, de software, chave na indústria de jogos. Tão importante quanto à qualidade artística do jogo.

O projeto desenvolvido é brevemente apresentado em 2.1 mostrando a relevância, escopo e objetivos deste trabalho e a metodologia empregada durante sua execução.

[4] Do inglês gameplay, é a maneira com a qual o jogador interage com o jogo.

O conceito motor gráfico é descrito no capítulo 2.1. Nesse capítulo são apresentados os diferentes tipos de motor assim como as diferenças entre o motor e o jogo. Os componentes que compõem um motor serão descritos brevemente[5].

Em Gerenciamento de Cenas são apresentados os conceitos de computação gráfica, comuns em jogos, para representar, gerenciar e renderizar uma cena. Bishop, em 7, frisa que técnicas de gerenciamento de cena são utilizadas há muito tempo em sistemas 3D - desde os primeiros simuladores de vôo - e apesar do avanço no hardware é necessário um gerenciamento de cena efetivo e de bom desempenho. Esse capítulo pretende familiarizar o leitor com os conceitos e apresentar os problemas.

O capítulo 5 descreve o motor gráfico criado. O problema de representação de cena é atacado com abordagens da engenharia de software e é apresentada a arquitetura - extensível, escalável e modularizada - do framework.

2.1 Este trabalho

Essa seção descreve o trabalho desenvolvido. Primeiramente é apresentada a motivação para a construção de um motor gráfico. Motivação necessária devido às diversas opções de motores existentes, mostrando assim a necessidade e a importância desse projeto tanto para os participantes como para o Centro de Informática.

A seção 2.1.2 apresenta os objetivos do trabalho. O escopo de um motor gráfico vai muito além do possível para o calendário do projeto. Apresentamos então o que é e o que não é contemplado e a preocupação com a continuidade do mesmo. Por fim, a seção 2.1.3 explica como o trabalho foi feito e como as responsabilidades foram divididas.

2.1.1 Relevância

Em um primeiro momento, a criação de um motor pode parecer uma alternativa ingênua e sem sentido. Existem diversos motores com ótimo desempenho e amplamente testados pela comunidade em diversos jogos, como o de Doom e ainda alternativas livre como Ogre3D 7.

Por outro lado, uma pesquisa com os motores existentes demonstram diversas fragilidades e empecilhos. O maior deles é o foco desses motores. Como descrito

[5] Mais detalhes podem ser encontrados no survey em 7, Capítulo 3 páginas: 33 - 54.

na seção anterior motores têm propósitos diferentes e um motor para jogos de esporte não possui os mesmos requisitos para os de jogos de estratégia.

Outro fator importante é o custo de personalização desses motores. Existe o custo de treinamento para desenvolver a personalização, familiarizar-se com o motor utilizado e ainda da integração propriamente dita. Com as atualizações do motor pela comunidade essa extensão precisa ser atualizada constantemente. Com o aumento do número de personalizações o custo dessas atualizações pode ser bastante alto e comparável com o da criação de um motor próprio.

Além disso, a criação de um motor cria nos desenvolvedores familiaridade com os diversos componentes que formam o framework e contribuirá para o desenvolvimento de pesquisas na área dentro do Centro de Informática, a exemplo do Forge7. A pesquisa na área de computação gráfica no Centro é bastante insipiente. A entrada do professor Silvio no Centro visa suprir essa deficiência e o projeto do motor é parte de uma estratégia para pesquisas de longo prazo.

2.1.2 Objetivos

Um motor gráfico envolve uma gama imensa de funcionalidades. Os motores comerciais custam alguns milhões de dólares e milhares de homens/hora. Relativo a esta complexidade, o objetivo deste trabalho é bastante humilde. Dos diversos problemas embutidos em um motor gráfico – descritos em 7, 7, 7 e 7 – pretendemos resolver os problemas da representação de cena e detecção de colisão[6].

Esses problemas foram escolhidos por formarem o núcleo, ou core, da engine. As funcionalidades não contempladas neste trabalho podem ser vistas como extensões na representação da cena e a seção 5.2.3 explica a maneira como isso deve ser feito na arquitetura proposta.

A estrutura de classes para a representação de cena tem impacto direto na detecção de colisão, por isso a preocupação com o mesmo. Mal projetado pode comprometer o desempenho da colisão e forçar a utilização de “remendos”.

Nosso trabalho pretende criar uma arquitetura, resolvendo o problema da representação de cena e da detecção de colisão, que proporcione boa performance e ao mesmo tempo facilite portabilidade. As diversas plataformas de execução de um jogo – PCs com Windows ou Linux, e os diversos consoles – impossibilitam um

[6] A representação de cena envolve a modelagem computacional da geometria e a renderização incluindo, os aspectos físicos e matemáticos envolvidos no processo.

motor de implementação única multi-plataforma. Assim é muito importante a facilidade de portabilidade da arquitetura.

2.1.3 Metodologia de Trabalho

Este trabalho foi desenvolvido utilizando a abordagem, em uma versão simplificada, para análise e projeto orientados a objetos descrita em 7. A pouca experiência em projetos orientados a objetos da equipe foi o fator principal para seguir esse caminho.

Inicialmente foi feita uma análise do projeto. Foram definidos os casos de uso e um diagrama conceitual dos conceitos inseridos no projeto. O projeto seguiu com a criação de diagramas de colaboração para auxiliar na divisão de responsabilidades entre as classes e por fim foi definida uma arquitetura.

Na arquitetura foram aplicados diversos padrões de projetos. Com os conceitos mapeados foram identificadas as opções e os padrões foram aplicados quando apropriado.

Além disso, existe um trabalho de graduação complementar sendo desenvolvido pelo aluno Leonardo Costa. O trabalho de Leonardo pretende traduzir a arquitetura de alto-nível descrita neste trabalho em componentes de código utilizando as tecnologias Windows e OpenGL.

3 O Motor Gráfico

Macedo Júnior, em 7, define motor gráfico como "*... sistema gráfico 3D que sintetiza as imagens em tempo real respondendo as chamadas da aplicação*", Macedo Júnior alerta para o fato de para a confecção de um motor "...*é necessário o conhecimento de uma ampla variedade de técnicas, algoritmos, estrutura de dados e diversas áreas da matemática pura"* como análise, álgebra linear e geometria analítica.

As próximas seções expões os diversos tipos de motor e suas classificação. A seção 3.2 mostra como, comumente, um motor é constituído e qual a responsabilidade de cada um dos componentes. E por fim, na última seção, uma pequena apresentação e análise de motores existente estudados durante o desenvolvimento deste trabalho com a finalidade de levantar requisitos.

3.1 Classificação

Diferentes tipos de jogos possuem requisitos, , diferentes. Assim, a classificação dos jogos também serve para classificar os motores. Os gêneros mais populares, e chaves para classificação, são os jogos de tiro em primeira pessoa – os FPS -, jogos de estratégia e jogos de "dirigir".

Em jogos do tipo FPS, o jogador, normalmente, encontra-se dentro de prédios e uma quantidade limitada de personagens aparece na tela. Esses jogos, portanto, focam nos detalhes visuais e na animação dos personagens. Doom é um exemplo desse tipo de jogo.

Figura 3 - Warcraft3 da Blizzard. Estratégia: Personagens, construções e terreno.

Nos jogos de estratégia o usuário possui, o que é chamado na área de jogos, uma “visão de deus”. O jogador ver o jogo “do céu” com acesso irrestrito as diversas localizações do mapa no jogo. O foco do motor é a habilidade de desenhar um grande número de objetos, com menos detalhes que os FPS, e o relevo – terra e água – do local onde o jogo acontece.

Por fim, temos os jogos de dirigir. O gênero engloba corridas de carro, simuladores de vôo, combates militares, etc. Esses jogos normalmente passam em locais abertos onde o ângulo de visão é quase ilimitado. Dessa maneira, os motores preocupam-se com técnicas especiais de nível de detalhe para reduzir o número de polígonos que precisam ser desenhados.

Figura 4 - Nascar Racing da EA Sports. Ângulo de visão sem restrições.

As diferenças têm impacto nos diversos componentes do motor. Cada gênero requer um cuidado especial com conjuntos distintos de requisitos. Técnicas de animação, por exemplo, os FPS precisam de modelos flexíveis e detalhadas enquanto jogos de estratégia utilizam modelos mais simples com um número limitado e previsível de animações.

Figura 5 - Age Of Empires 3 e Age of Mythology. Arte Diferente, mesmo motor.

Existem ainda outras categorias de jogos – plataforma, puzzle – que normalmente utilizam gráficos em duas dimensões ou isométricos7. Outros gêneros que utilizam gráficos 3D têm os mesmos requisitos para o motor que os gêneros citados anteriormente.

3.2 Componentes

Nessa seção os componentes serão descritos superficialmente. Para um detalhamento e estudo aprofundado recomendo 7,7,7,7 e 7.

É importante salientar que este trabalho pretende apenas criar uma arquitetura para representação de cena prevendo futuras adições. Essa seção é importante para situar o leitor e ajudá-lo a entender o contexto do trabalho.

3.2.1 Pipeline de Renderização

O pipeline de renderização concentra as operações de *back-end* em sistemas gráficos. Um dos objetivos do motor é alimentar, de maneira eficiente, esse pipeline com polígonos a serem renderizados.

As operações algébricas sobre os vértices dos polígonos – transformações, iluminação, texturização, rasterização[7], shading entre outros - são executadas durante o pipeline. Diversos fabricantes de placas de vídeo – como nVidia e ATI - implementam essas operações em seu processador gráfico, a GPU[8][9].

O processo de iluminação por ser de grande importância e ter diversas peculiaridades será descrito na seção 3.2.2.

Figura 6 - Processo de Texturização.

[7] Rasterização é o processo de transformar uma imagem de duas dimensões descrita como um conjunto de pontos e polígonos em pixels no formata apropriado para a saída – monitor ou impressora.

[8] Graphics Processing Unit. Processador de placas de vídeo utilizado em PCs e consoles ou vídeo games.

[9] Algumas placas são dedicadas às operações sobre vídeos: criar e editar. Com implementação de algoritmos de compressão e descompressão de vídeos e imagens como Mpeg-2, Mpeg-4, JPG, entre outros.

Texturização é o processo em que uma imagem é mapeada a um modelo em três dimensões na cena.

A texturização é normalmente utilizada para aumentar o realismo e também facilita a identificação de personagens conhecidos. Muito usado em jogos de esporte para identificar jogadores reais. Técnicas de mapas de iluminação utilizam modelos de iluminação mais realistas, e mais caros computacionalmente, para gerar uma imagem, também usam texturização para aplicar – a iluminação pré-computada – nos modelos nos sistemas interativos.

Figura 7 - Texturização. Rosto do jogador Raul criado por artista e aplicado ao modelo.

Até 2001, placas gráficas usavam um pipeline fixo em que os modelos de iluminação e de shading eram definidos pelo fabricante. Foram então criados os shaders programáveis. Shaders são programas utilizado para determinar a cor – a partir das fontes de luz, material do objeto e texturas – da superfície de um objeto.

Os shaders são programados utilizando a linguagem de máquina da placa de vídeo ou as linguagens de shading de alto-nível, HLSL. Existem diversas HLSLs e fica a cargo do programador qual utilizar.

Os shaders executados em GPUs podem ser de dois tipos: Pixel Shader ou Vertex Shader. Os pixel shaders operam sobre todos os pixels da imagem criada, enquanto que os vertex shaders operam somente sobre os vértices dos polígonos do modelo usando técnicas de interpolação para colorir pixels internos do polígono.

O vertex shader torna as funções de texturização e iluminação programáveis. Os pixel shader operam depois do pipeline geométrico – transformações, iluminação e textura – e antes da rasterização. Operam para produzir a cor final do pixel.

Independente do tipo de shader, a computação da cor é baseada em um modelo de iluminação que define a cor em um dado ponto do espaço. Iluminação e o modelo de iluminação mais comuns são descritos na próxima seção.

A qualidade e o desempenho dos GPUs e a facilidade de programação proporcionada pelos shaders têm levado pesquisadores de diversas áreas a utilizá-los com propósitos diferentes dos da computação gráfica, como em 7 e 7.

3.2.2 Iluminação e Materiais

Realismo é, normalmente, o objetivo principal quando criamos imagens digitais. Um elemento chave para o realismo é mostrar as cores corretas para os objetos na imagem.

Representação realística de cores em computação gráfica requer quantificação de luz e materiais e o desenvolvimento das regras que descrevem as interações – os chamados modelos de iluminação – entre o material e o meio. A determinação da cor deve ser baseada em na natureza da luz e da percepção das cores pelos humanos.

Dois fenômenos têm maior importância na geração de imagens. A interação da luz entre as bordas dos objetos e como a luz é absorvida ou espalhada dentro do material.

Técnicas de iluminação podem ser classificadas entre dois grandes grupos: os com técnicas de iluminação global, que contemplam algoritmos baseados na física da luz e suas propriedades e os de iluminação local, que levam em conta apenas a fonte de luz e sua incidência no objeto a ser iluminado.

Algoritmos de iluminação global levam em conta a incidência dos raios provenientes das fontes de luz assim como a influência dos raios refletidos por outros objetos na cena. O princípio básico que descreve o movimento da luz em um ambiente é a lei de conservação da energia. A energia proveniente de uma fonte de luz ao atingir a borda de um objeto é absorvida ou refletida. A energia refletida influencia então nos demais objetos na sala.

Em sistemas interativos de alta performance, como jogos, as técnicas de representação de luz e materiais são bastante simples, normalmente as de iluminação local. Devido à complexidade computacional de algoritmos de iluminação global.

Técnicas de iluminação local levam em conta somente a interação dos objetos com as fontes de luz e normalmente são modelos empíricos que não levam em conta a lei de conservação de energia.

O modelo mais comum é o modelo de iluminação de Phong. Nesse modelo a cor do material é formada por três componentes: ambiental, especular e difusa. De acordo com a equação abaixo:

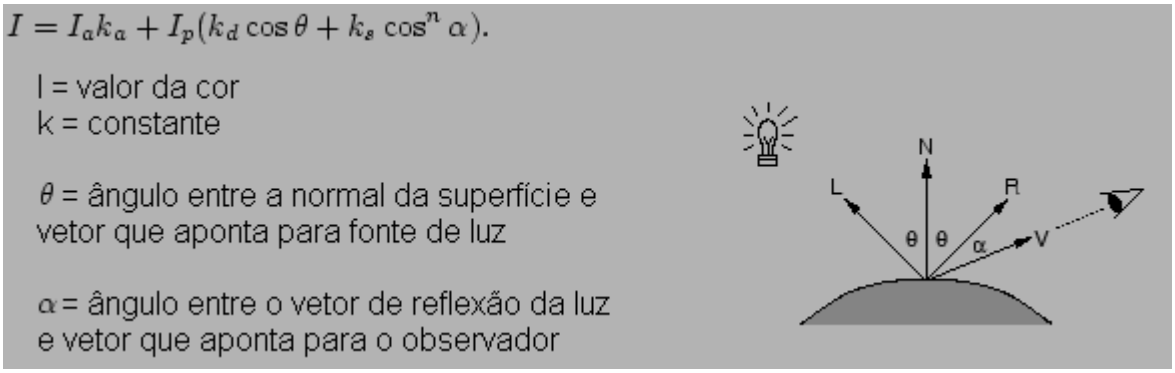

Figura 8 – Equação de iluminação de Phong.

O componente difuso ($I_p k_d \cos\theta$) e o ambiental ($I_a k_a$) são representações simplificadas do modelo ideal Lambertiano para superfícies difusas.

O modelo Lambertiano ideal é uma simplificação do fenômeno real. O modelo assume que a luz vem de todas as direções, não existem sombras e a superfície do modelo é uniforme – sem manchas - e como conseqüência a intensidade da luz refletida é igual em todas as direções para esses componentes.

A componente ambiental representa a quantidade de luz inerente ao ambiente. Pode ser interpretada como a luz do sol.

O componente especular ($I_p k_s \cos^n\theta$) é um componente modelado por uma função vetorial de reflexão. Esse componente produz brilho relacionado à geometria do modelo e a posição da fonte de luz.

Maiores detalhes podem ser encontrados em 7, 7 e 7.

3.2.3 Câmera

Os objetos em uma cena 3D, normalmente, são feitos por um modelador em um software à parte. As definições, dessa maneira, possuem sistemas de coordenadas e escalas próprias. Na montagem da cena, esses objetos precisam ser postos na mesma escala e em um mesmo sistema de coordenadas, chamado de sistema de coordenadas mundial. O passo seguinte é transformar esses objetos para a perspectiva de onde o usuário está localizado.

A câmera representa esse usuário, observador. A câmera define que porção da cena é visível pelo observador assim como a perspectiva dos objetos. Definem uma câmera os parâmetros: ângulo de visão horizontal, ângulo de visão vertical, localização da câmera na cena, sistema de coordenadas, plano de projeção, entre outros.

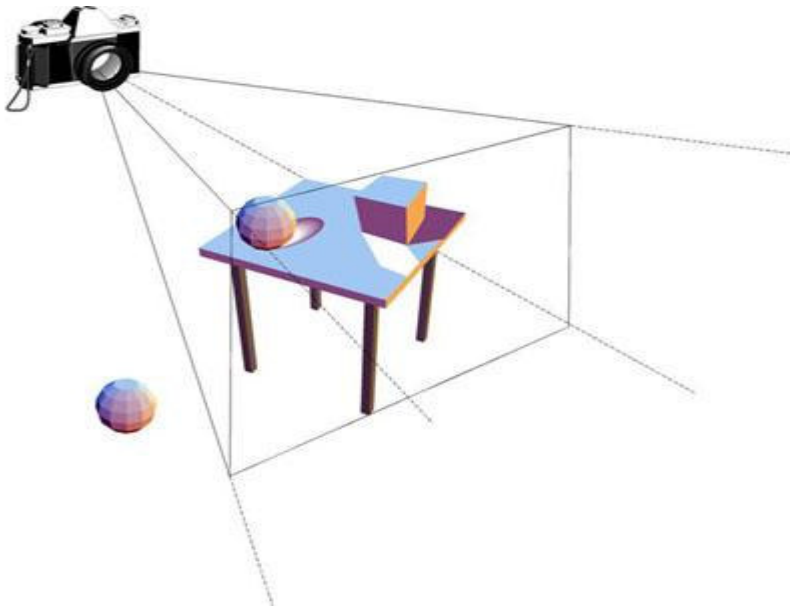

Figura 9 - Câmera Virtual. Definição da visibilidade.

A movimentação da câmera é feita pelo usuário do motor e a estratégia de movimentação da câmera é crucial para oferecer uma boa experiência ao jogador.

A localização da câmera é essencial para algoritmos de nível de detalhe 77. A maioria deles baseia-se nas distâncias da câmera para o objeto e da localização real do ponto e sua localização projetada. A eficiência dessas operações compromete a qualidade desses algoritmos.

Dessa maneira, a câmera deve ter uma boa representação, assim como ser acessível pelos diversos módulos da engine.

3.2.4 Gerenciamento de Cena

O pipeline é o *back-end* do motor, enquanto o gerenciamento de cena é o *front-end*. O gerenciamento de cena eficiente saberá enviar ao *back-end* somente os polígonos necessários para renderização.

O gerenciamento de cena tenta evitar que polígonos não visíveis pelo observador sejam enviados para o pipeline de renderização. Enviar esses polígonos para

renderização força o processador gráfico a fazer todas as operações em vértices desnecessariamente, já que esses não são vistos.

Para isso o gerenciador de cena deve, entre outras coisas, ser responsável pelas operações de *cliping* e *culling*.

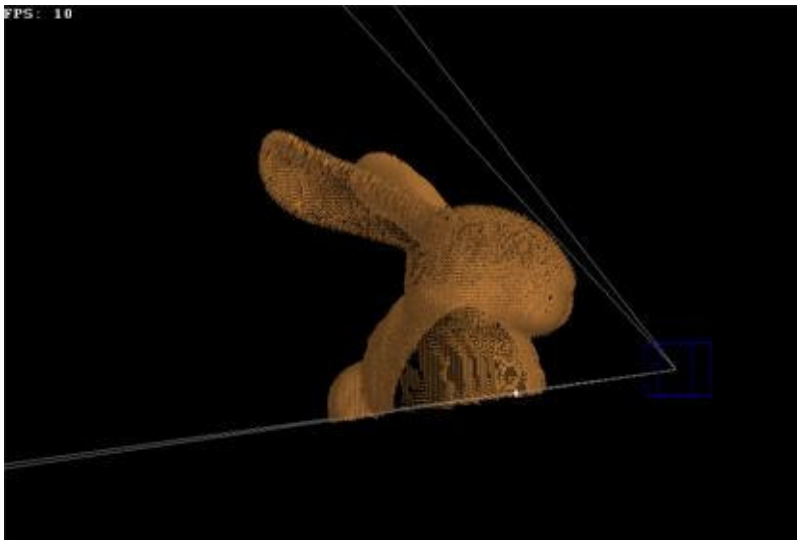

Figura 10 - Culling. Só a parte visível do modelo é enviada ao backend.

Esse componente do motor é o tema principal deste trabalho. Dessa maneira deixamos uma discussão com mais detalhes para um capítulo inteiramente dedicado ao assunto, o capítulo 4.

3.2.5 Modelagem Geométrica

Modelagem geométrica é a parte do motor responsável pela representação dos modelos em termos de geometria, ou seja, pontos e polígonos. De maneira geral é a construção ou uso de modelos geométricos na representação do modelo.

Modelos geométricos têm uma descrição da topologia do modelo. Diferente de abordagens procedurais que descrevem objetos por algoritmos e com modelos de superfícies matemáticas implícitas – como fractais, por exemplo.

Existem diversos aplicativos de suporte a criação de modelos geométricos altamente conhecidos na comunidade de jogos e cinema, como o Maya e o 3D Studio MAX. A facilidade de uso desses aplicativos ajudou a popularizar a modelagem geométrica.

É importante frisar que recentemente outras abordagens de modelagem, em especial a modelagem baseada em física, vêm ganhando espaço. Mas devido a sua complexidade computacional e imaturidade do campo somente as técnicas mais

simples de sistemas de partículas e de cordas são utilizadas em sistemas interativos em tempo real como jogos.

3.3 Motores

Nessa seção serão discutidos alguns motores estudados e avaliados, principalmente, para entender os requisitos implementados e funcionalidades existentes[10] em sistemas similares.

O Site devmaster.net7 gerencia uma lista[11] de motores gráficos existente – livres e proprietários. O site possui um ranking dos melhores motores de acordo com as opiniões dos visitantes. Baseado nessa lista utilizamos três das engines mais populares para estudo.

3.3.1 Ogre3D

A Ogre7 (Object-Oriented Graphics Rendering Engine) é o motor mais popular da lista do devmaster.net. Essa engine foi construída objetivando a facilidade de uso, fato que explica, parcialmente, a popularidade.

A facilidade de uso é com certeza o ponto mais forte do motor. Com isso, existe uma vasta comunidade de usuários e colaboradores, crucial e um projeto de software livre.

No entanto, algumas decisões de projeto[12] comprometeram a performance da engine e a independência de plataforma. A simplicidade com que a arquitetura foi tratada criou problemas de portabilidade. O motor tem um desempenho bom em ambientes Windows com Diret3D e é bastante debilitado para Windows ou Linux com OpenGL.

---

[10] Como explicitado na seção 2.1.3 o foco do trabalho é na análise e projeto descrito em 7.

[11] A lista possui 215 motores, em 14 linguagens de programação diferente utilizando até quatro APIs gráficas. O site possui uma média por volta de 10 mil visitantes por dia.

[12] Com o intuito de não inundar o leitor com tópicos ainda pouco discutidos as decisões serão devidamente indicadas quando apropriado, no momento em que forem discutidas em relação ao presente trabalho. Válido também para os demais motores.

3.3.2 Irrlicht Engine

O motor Irrlich7, seguindo o Ogre, privilegia facilidade de uso. No entanto, a arquitetura é mais extensível e as diferentes implementações possuem desempenho semelhante.

A representação de cena da Irrlich utiliza uma abordagem bastante aberta, no sentido que os nós assumem diversos papéis. Parte dessa idéia foi utilizada em nosso motor.

Essa liberdade cria problemas de portabilidade já que a implementação do mesmo é feita na própria classe e são necessário diversas implementações do mesmo nós, consequentemente dos mesmos algoritmos.

3.3.3 Open SceneGraph

O Open SceneGraph7, tem a arquitetura mais desenvolvida entre os motores pesquisados. Utiliza diversos padrões de projetos e teve seu projeto publicado e discutido em diversos congressos na área da computação gráfica.

Dessa maneira, nosso projeto herda diversos aspectos do Open SceneGraph e preocupa-se em estender a arquitetura de maneira a torná-la mais flexível e sofra menos impacto dos problemas descritos na seção 4.3 deste documento.

Open SceneGraph possui uma comunidade com mais de mil colaboradores ativos em sua lista de discussão e download do seu código. Existem diversos projetos sendo criados com Open SceneGraph nas áreas de simulação visual, jogos, realidade virtual, visualização científica e modelagem.

3.3.4 Outros

Alguns motores proprietários - 77777 - foram analisados, mas não foram levados em consideração por diversos motivos.

A quantidade de informação sobre a arquitetura e as tecnologias utilizadas é limitada comprometendo uma boa avaliação do sistema. O custo das licenças é bastante além dos recursos financeiros do grupo de pesquisa em computação gráfica no Centro do Informática, impossibilitando o teste desses motores.

Por fim, muitas das empresas responsáveis por esses motores possuem parcerias com empresas fabricantes de sistema operacional - Microsoft e Apple – e produtoras de placas de vídeo – ATI, NVIDIA, entre outras. Tais parcerias levam muitas vezes a decisões políticas sobre o produto e não tecnológicas.

4 Gerenciamento de Cenas

O principal requisito para um motor gráfico é a renderização de uma cena, descrita em três dimensões, na tela do usuário. A modelagem computacional da cena tem impacto direto no desempenho, resultado visual e consequentemente satisfação do usuário.

Nessa seção será descrita a representação hierárquica, em árvore, da cena por ser a mais comum no mercado e ter sido amplamente discutida pela comunidade acadêmica.

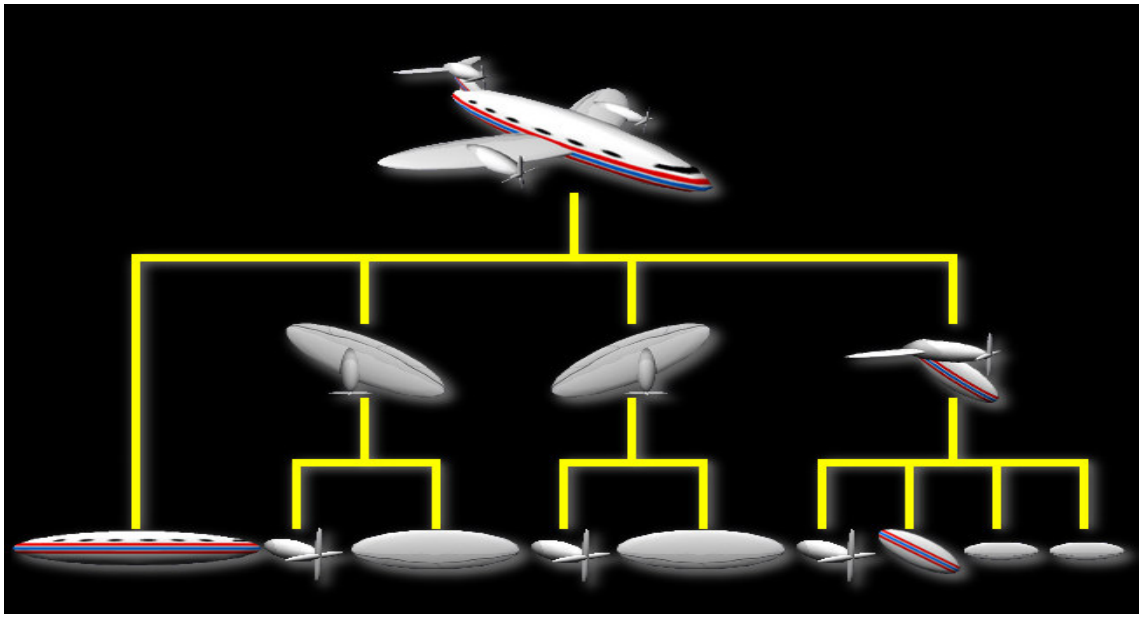

Figura 11 - Representação Hierárquica. Instância de um grafo de cena para um modelo de avião.

Os problemas inerentes à representação em árvore são discutidos na seção 4.3. Bethel et al, em 7, debate os diversos problemas dessa representação e as fragilidades conhecidas. Pretendemos então apresentar possíveis soluções e como serão aplicadas na engine.

A organização do conteúdo em um grafo de cena é muito importante para jogos em diversos aspectos. A quantidade de conteúdo normalmente é muito grande. Esse conteúdo normalmente é criado em partes pequenas pelos artistas – designers – e o grafo de cena é uma maneira de organizar e concentrar os diversos modelos em uma cena.

A organização hierárquica, do grafo de cena, é uma forma de explorar o principio da localidade. Objetos relacionados tendem a ficar fisicamente próxima em uma cena.

O grafo de cena, dessa maneira, facilita o culing, ou seja, eliminar objetos fora do ângulo de visão do observador diminuindo a quantidade de objetos a serem processados[13].

A representação hierárquica também é importante para sistemas de detecção de colisão. Um nó do grafo de cena contêm, espacialmente, seus filhos. Um teste de interseção pode determinar se toda subárvore daquele nó precisa ou não ser testada para colisão. Caso uma bala, por exemplo, não bate em uma casa, não baterá nos móveis dentro dela.

4.1 O Grafo de cena

O modelo de grafo de cena é a abordagem mais popular e poderosa para representação de cenas. Apesar do nome grafo, jargão na área, o grafo de cena é na realidade uma árvore. Cada nó pode ter vários filhos mas somente um pai.

O grafo de cena representa objetos de maneira hierárquica. Os nós representam à hierarquia e divisão da cena enquanto as folhas contêm a geometria, normalmente triângulos, que possuem os atributos – cor, localização, rotações, etc. - especificados no caminho desde a raiz e que são enviados para placa gráfica.

A hierarquia pode ser dependente dos modelos, como no exemplo da casa e dos móveis e do avião. Ou pode ser feita pela subdivisão do espaço com algoritmos como BSP e Octree explicados em 4.2.

Em APIs com grafo de cena, o desenvolvedor foca o trabalho na construção da cena[14] - com operações como criar cena, inserir objeto na cena, inserir fonte de luz, entre outras. O grafo de cena encapsula problemas de baixo nível e dependentes de plataforma como texturização, o pipeline gráfico, entre outros.

4.2 Estado da arte

Existem diversas técnicas de modelagem do grafo de cena. Entre elas podemos destacar as que utilização ordenação espacial e as que utilizam hierarquias de volumes, as *bounding hierarchies*.

---

[13] Nesse caso, processamento são as operações sobre os vértices – transformações, iluminação, aplicação de texturas, etc. - do pipeline gráfico.

[14] Os casos de uso utilizados na análise deste trabalho, mostrados em 5.1.1, refletem bem esse paradigma e esclarecem melhoro conceito de grafo de cena.

As árvores de particionamento binário (BSPs) subdividem recursivamente o espaço, utilizando hiperplanos, em conjuntos convexos. A subdivisão gera uma representação da cena em forma de árvore, conhecida como árvore BSP.

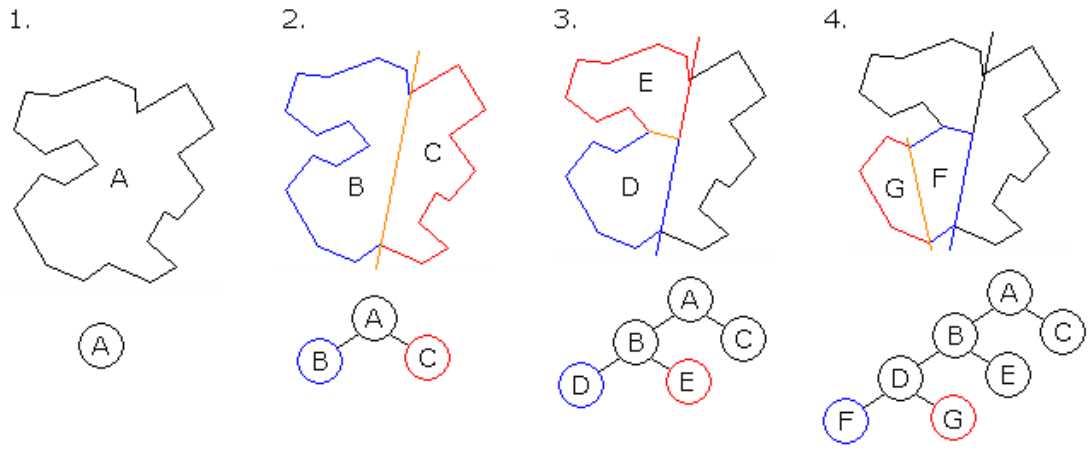

Figura 12 - Criação de uma árvore BSP, por Fredik.

A técnica BSP foi amplamente utilizada em diversos jogos, entre eles Quake, Doom, Unreal e atualmente é bastante comum em sistemas de modelagem assistida por computador utilizados por engenheiros e arquitetos.

Outra técnica de subdivisão são as Octrees. As Octrees são utilizadas, normalmente, para testes de visibilidade. A octree sempre particiona o espaço em oito cubos de mesmo tamanho. A divisão continua até que um nó seja denominado livre ou ocupado.

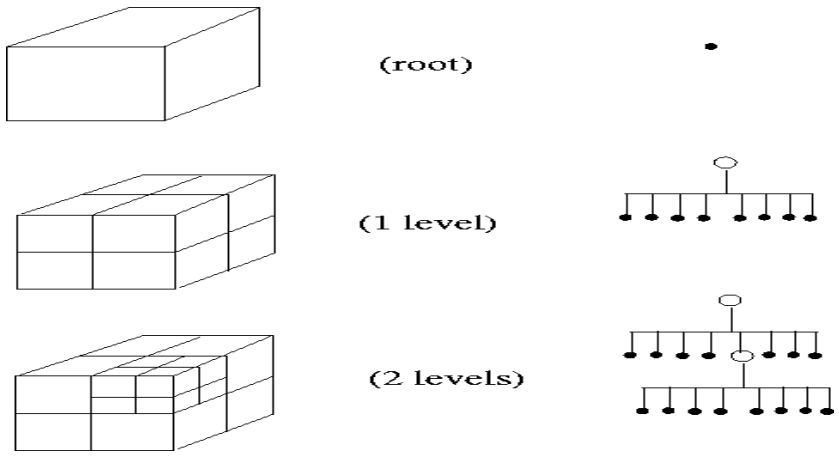

Figura 13 - Criação de uma Octree

O maior problema encontrado nessas representações por particionamento de espaço é o custo computacional de criação da árvore. Apesar dos ótimos resultados, quando utilizados em objetos estáticos o desempenho para objetos

dinâmicos é drasticamente menor, pois é necessário que a árvore – BSP ou Octree – seja recalculada e em cenas com um grande número de objetos dinâmicos pode comprometer a experiências dos usuários finais da aplicação devido à freqüência de criação da mesma.

As hierarquias de volume, bounding hierachies, são algoritmos que criam uma representação simplificada do modelo utilizando volumes com teste de interseção simples como cubos, esferas e cilindros.

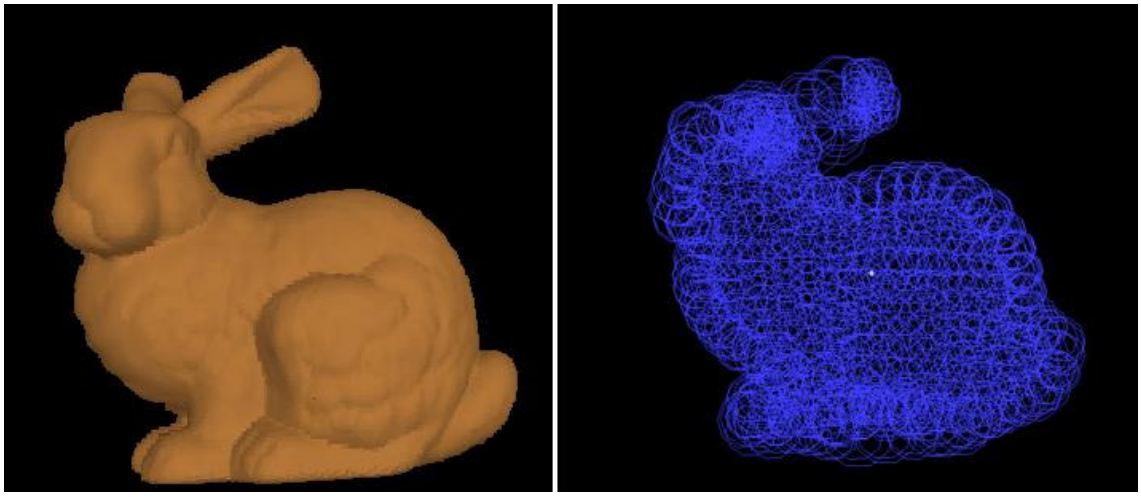

Figura 14 - Modelo de coelho à direita e sua representação com uma hierarquia de esferas.

Um exemplo são os Axis-Aligned Bounding Boxes (AABB), cubos alinhados com o sistema de coordenadas. Esses volumes são rapidamente construídos e algoritmos de união de bounding boxes de vários modelos para formar bounding boxes de cenas inteiras são bastantes simples. No entanto, modelos complexos podem ser mal representados por bounding boxes, detectando colisões que na realidade não existem.

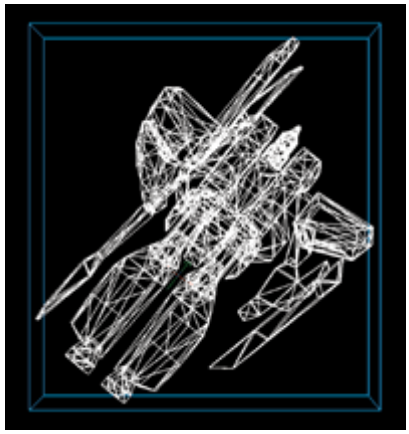

Figura 15 - AABB de um modelo. Note que bounding boxes menores (do braço, da perna, etc.) podem ser utilizados para criar um hierarquia.

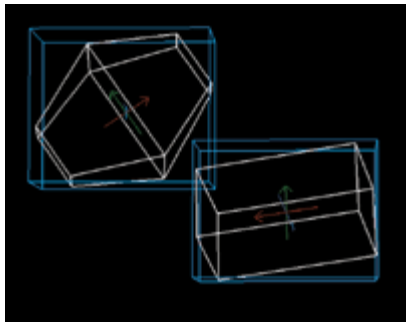

Figura 16 – Bounding Boxes contêm interseção, mas os modelos não colidem.

Com a melhora nos dispositivos de hardware, algoritmos mais elaborados puderam ser utilizado. A OBBTree7 é a estrutura com melhores resultados. Em tempo de execução é necessário apenas percorrer a árvore para os testes de visibilidade e colisão.

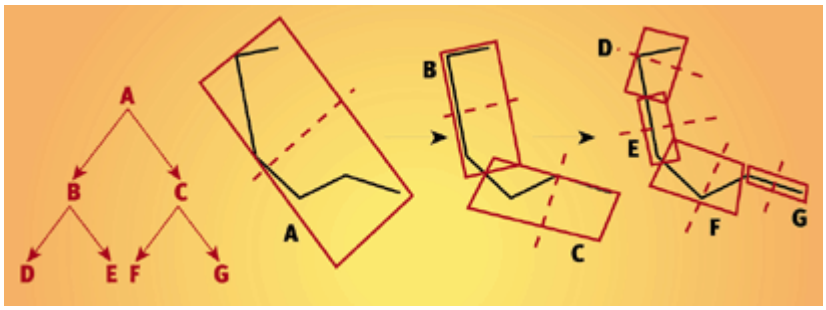

Figura 17 - Construção de uma OBBTree em uma curva de duas dimensões.

Muitas vezes utiliza-se AABB para um primeira teste de interseção e em caso positivo utiliza-se a OBBTree do mesmo modelo e refaz o teste. Por fim, em um segundo positivo, utiliza-se a intercessão com todos os triângulos do modelo.

Apesar de diminuir os problemas das técnicas de particionamento de espaço, as hierarquias de volume têm performance em tempo de execução para objetos estáticos aquém das técnicas de BSP e Octree para testes de colisão e visibilidade.

Nota-se claramente que as técnicas de particionamento de espaço e hierarquia de volume são complementares. Para a representação de construções e outros modelos naturalmente estáticos as abordagens de particionamento de volume são bem adequadas enquanto que para objetos bem dinâmicos

4.3 Problemas

O objetivo principal do grafo de cena é abstrair do desenvolvedor a complexidade das diversas APIs gráficas existente provendo também um ambiente multi-plataforma. Dessa maneira, a preocupação com portabilidade é importantíssima. A arquitetura deve prover maneiras simples de criar implementações diferentes para as mesmas operações.

4.3.1 Portabilidade

Devido à quantidade de possibilidades de ambientes de execução – consoles, PCs – e diversas APIs gráficas com desempenhos diferentes para cada configuração de hardware não faz sentido uma implementação multi-plataforma do motor.

Por exemplo, temos placas da ATI que funcionam melhor com códigos de Direct3D, enquanto computadores com placas nVidia têm melhor desempenho com OpenGL. Dessa maneira, não podemos prevê qual a melhor opção e temos que nos preocupar em criar duas implementações.

Jogos de computador criam mais uma complicação. Além de PCs, os jogos têm ambientes de execução peculiares, os consoles. Cada console tem seu próprio framework e APIs para vídeo, som, etc. Dessa maneira, é muito comum as desenvolvedores terem várias equipes para o mesmo projeto, um para cada plataforma. Um motor que privilegie código portável diminuirá o tamanho desses times.

Dessa maneira, devemos pensar em como minimizar o trabalho com portabilidade. A arquitetura deve privilegiar e facilitar códigos portáveis. Nosso motor faz isso separando a implementação em códigos portáveis, códigos dependentes de plataforma de execução e códigos dependentes de APIs gráficas. Essa separação é melhor explicada no capítulo 5.

4.3.2 Concorrência

Um problema inerente ao grado de cena são os empecilhos existentes para criar vários processos concorrentes e aumentar o desempenho.

Em 7, cria a discussão em torno da real eficiência dos grafos de cena. A abordagem facilita para composição de cena: localização, orientação, agrupamento, entre outras coisas. Essa visão da cena facilita para o programador, usuário do motor, a criação de sistemas interativos.

No entanto, não cria uma estrutura preparada para a renderização. A estrutura de grado de cena impõe diversas restrições de ordenamento. Essas restrições podem impossibilitar o paralelismo no processamento. O mesmo Bethel 7 , atenta para o fato da criação de operações de otimização do grafo de cena para facilitar paralelismo e propiciar um aumento de desempenho.

4.3.3 Colisão

Em sistemas de simulação realista nos quais se criam representações computacionais de máquinas, pessoas, animais e dos mais diversos objetos, como em um jogo. É necessário modelar as interações dos objetos precisamente. Dessa maneira obtém-se um resultado visual satisfatório, de extrema importância nesse tipo de aplicação. A forma de interação mais importante e tratável genericamente é a colisão.

Segundo Linn, em 7, uma colisão é dividida em três fases: detecção, cálculo dos pontos de contato e resposta. A resposta é dependente da aplicação e como nosso foca é na criação de um framework padrão, ficamos com a detecção e o cálculo dos pontos de contato. No entanto, mecanismos de callback foram implementados para que o usuário possa fornecer suas funções que tratarão a colisão, a reposta.

As cenas criadas em jogos eletrônicos têm milhares de polígonos que precisam ser renderizadas a uma taxa mínima de trinta quadros por segundo, para não causar desconforto na visão do usuário. Tais requisitos condenam algoritmos que testem interseções entre todos os polígonos, pois têm complexidade de O (n^n) testes por frame - sendo n o número de polígonos.

Dessa maneira, o grafo de cena precisa ser projetado de maneira a facilitar a detecção de colisão e o cálculo dos pontos de contato. Para isso, utilizaremos estruturas híbridas com particionamento espacial em objetos estáticos e *bounding hierarchies* em objetos dinâmicos.

5 REvolution Engine

Essa seção descreve o trabalho desenvolvido. De acordo com o descrito na seção 2.1.3, separamos o projeto em duas fases: análise e projeto. Durante a análise criamos os casos de uso dos cenários possíveis na utilização do motor e um diagrama[15] conceitual para conhecer melhor o problema.

Na fase de projeto criamos os diagramas de colaboração para identificarmos como os objetos iam se comunicar e qual o papel de cada um. Por fim, criamos os diagramas da arquitetura final do motor.

5.1 Análise

Durante os primeiros meses da construção da engine foram criados os casos de uso para identificar os cenários possíveis de utilização da mesma. E a partir dos casos de uso, como é aconselhado em 7, foi criado um diagrama conceitual.

Esses artefatos foram então utilizados para a fase de projeto e continuaram a ser refinados à medida que sentimos a necessidade – o que aconteceu em diversas oportunidades.

5.1.1 Casos de Uso

Os casos de uso aqui representados foram criados a partir da análise dos motores. Foram listados as features dos motores e diversos fóruns de discussão foram observados e analisados.

Durante o processo de análise – na descrição dos casos de uso e na construção do modelo conceitual – os casos de uso sofreram diversas mudanças. A discussão ajudou a entender melhor o problema refinando de maneira natural os mesmos.

[15] Diversos diagramas não ficaram com boa aparência na sua posição ideal do documento. Para melhor visualização, alguns deles, foram repetidos em tamanho maior no Apêndice A.

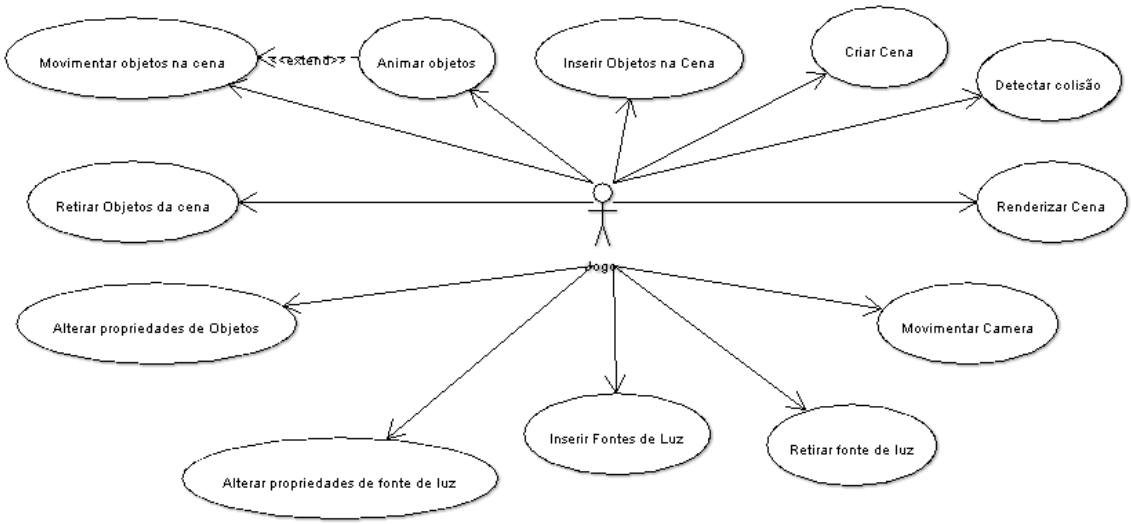

Figura 18 - Casos de Uso. Análise inicial baseada somente nos requisitos.

Durante a descrição dos casos de uso foram identificadas diversas similaridades entre os mesmos que geraram mais casos de uso e uma descrição mais real dos cenários.

Primeiro foi identificado que não existiam os casos de uso criar objeto nem o criar fontes de luz. Tais cenários são essenciais para o usuário e são necessários para a inserção de objetos e fontes de luz na cena.

Em seguida, casos de uso mais simples - recuperar objeto, recuperar fonte de luz - foram criados pois foram identificados diversos casos de uso com passos iniciais idênticos. Esses passos foram então embutidos nesses casos de uso.

Durante a fase de projeto, na criação dos diagramas de colaboração foi identificada a necessidade de refinar alguns casos de uso. Dessa maneira, o caso de uso movimentar câmera foi dividido em: modificar zoom da câmera, modificar posição da câmera e modificar sistema de coordenada da câmera.

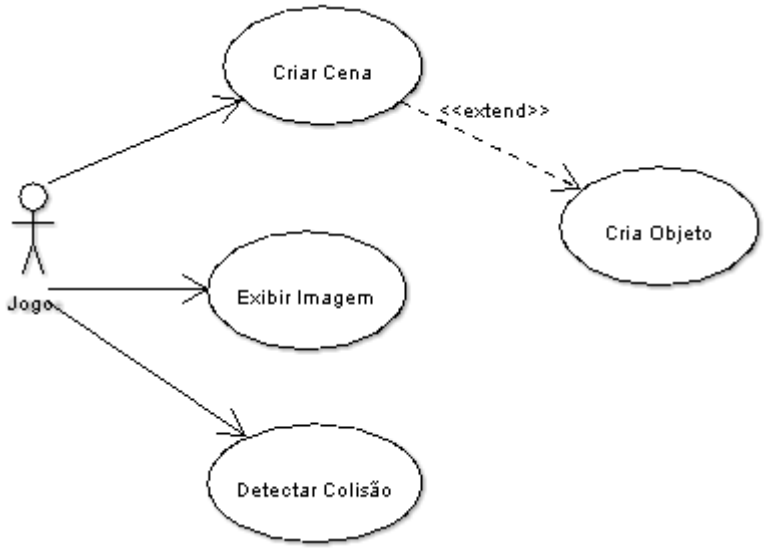

Figura 19 - Casos de Uso. Criar Objeto.

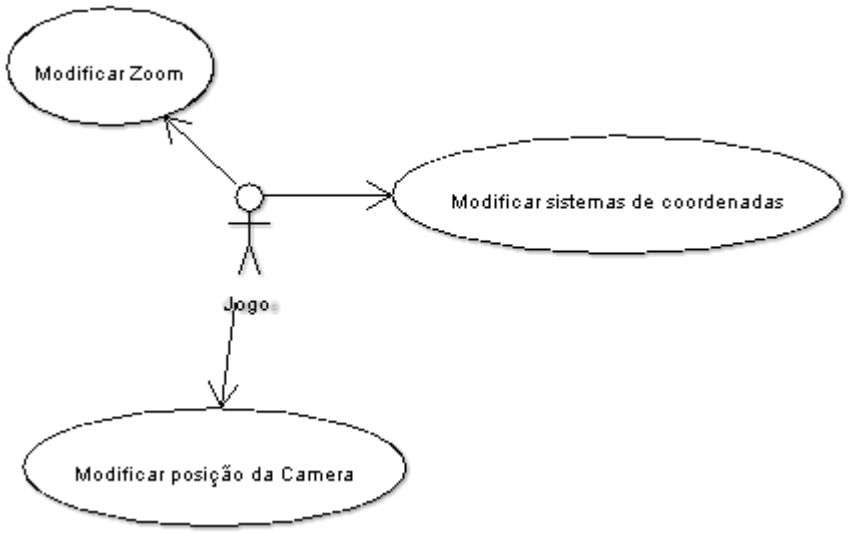

Figura 20 - Casos de Uso para Câmera Virtual.

Outro caso de uso dividido foi Alterar propriedades do objeto que foi dividido em modificar textura, modificar material e modificar orientação. O mesmo foi feito com Modificar fonte de luz, dividido em Mover fonte de luz e Mudar Cor da fonte de Luz.

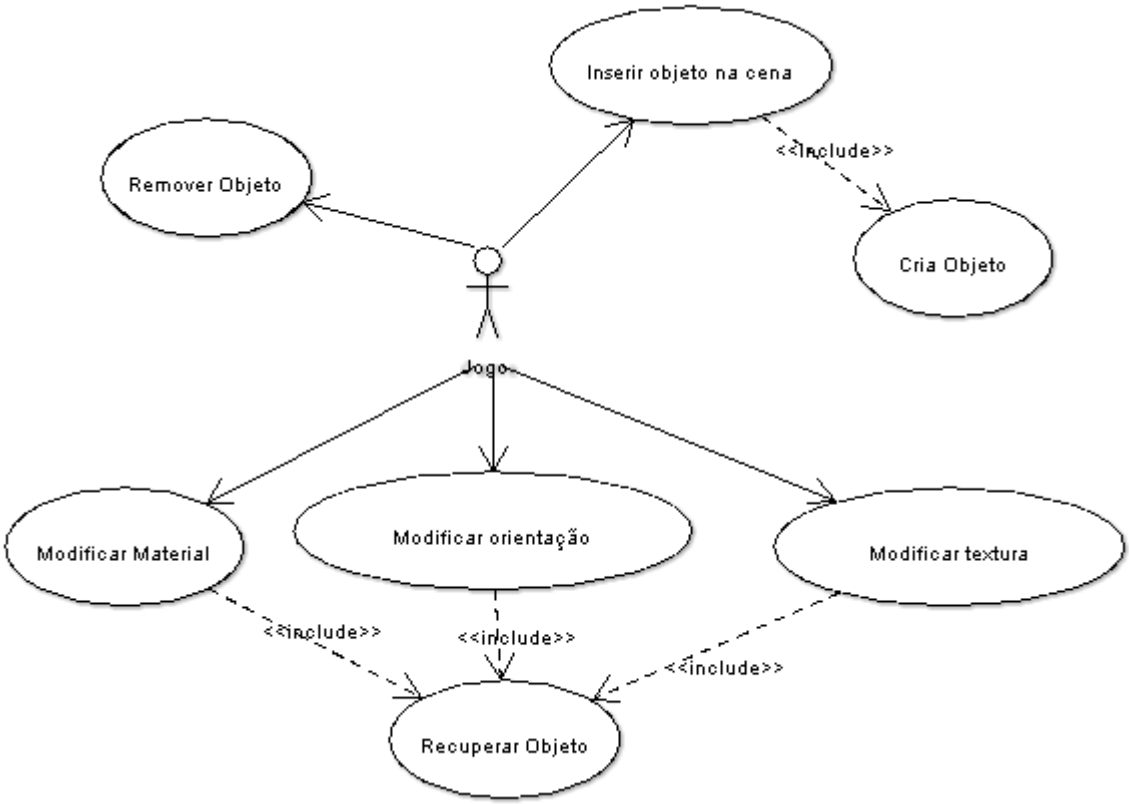

Figura 21 - Novos casos de uso de Objeto.

Importante salientar que durante todo o processo não foi identificado nenhuma necessidade de outro ator que não fosse o jogo. Outros sistemas podem vir a ter acesso à cena – como o sistema de som, a inteligência artificial do jogo, entre outros - mas não impactaram de nenhuma maneira nos casos de uso já existentes nem criariam a necessidade de outros cenários.

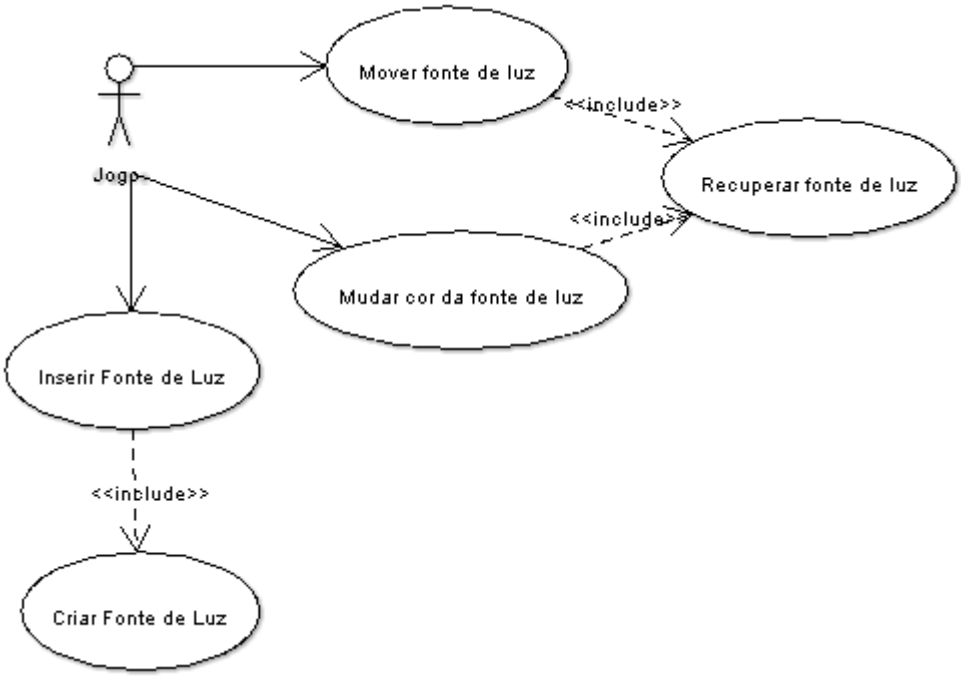

Figura 22 - Casos de uso para fontes de luz.

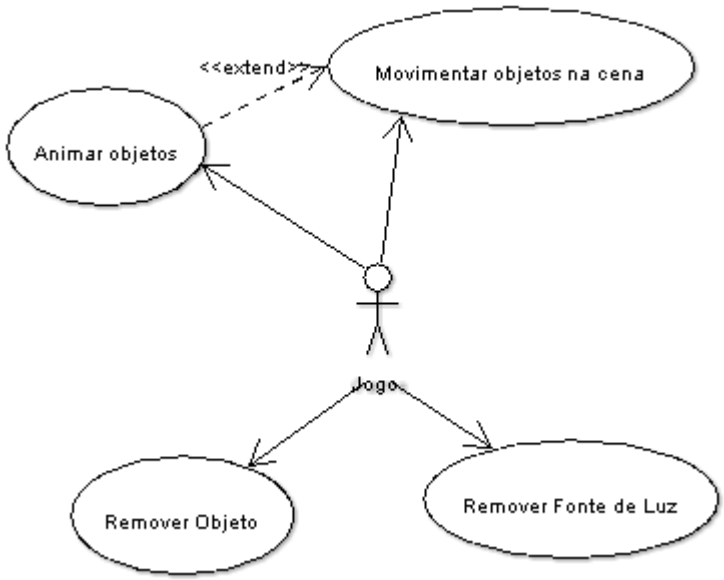

Figura 23 - Casos de uso mantidos.

A seguir a lista dos casos de uso levantados e uma descrição simples de cada um deles. Esses casos de uso foram utilizados para criar um modelo conceitual assim como os diagramas de colaboração.

O modelo utilizado para a descrição do casos de uso foi baseado no apresentado em 64. Outros modelos, talvez mais úteis, existem mas para manter o padrão utilizado durante todo o trabalho esse foi o modelo escolhido.

Tabela 1 - Caso de Uso: Criar Cena

Caso de uso	Criar Cena
Tipo	◆ primário □ secundário □ opcional
Finalidade	Cria a representação hierárquica de um ambiente.
Visão Geral	Usuário cria o nó raiz do grafo de cena.
Seqüência de Eventos	1. Criar uma nova instância de um objeto de cena 2. Insere esse objeto na cena

Tabela 2 - Caso de Uso: Criar objeto

Caso de uso	Criar objeto
Tipo	◆primário □ secundário □ opcional
Finalidade	Criar modelo computacional de personagens, animais, construções, entre outros.
Visão Geral	Usuário cria um objeto informando a geometria.
Seqüência de Eventos	1. Criar uma nova instância de um objeto de cena 2. Usuário especifica a geometria (lista de polígonos) que compõem o objeto. 3. Usuário especifica se o objeto é estático ou não 4. Usuário especifica posição do objeto 5. Usuário informa orientação do objeto 6. Usuário informa à função que trata colisões desse objeto

Tabela 3 - Caso de Uso: Inserir objetos na cena

Caso de uso	Inserir Objeto na cena
Tipo	◆primário □ secundário □ opcional
Finalidade	Adicionar objeto na cena.
Visão Geral	Usuário insere um novo objeto na cena.
Seqüência de Eventos	1. Criar objeto 2. Insere objeto na cena

Tabela 4 – Caso de Uso: Movimentar objetos na cena

Caso de uso	Movimenta objetos na cena
Tipo	◆primário □ secundário □ opcional
Finalidade	Da dinamismo a cena e a sensação de tempo.
Visão Geral	Usuário informa nova posição do objeto.

Seqüência de Eventos		1. Usuário recupera o objeto 2. Chama o método setPosition() 3. Sistema modifica sua posição 4. Sistema reposiciona o nó no grafo de cena

Tabela 5 - Caso de Uso: Animar objetos

Caso de uso	Animar objetos	
Tipo	□ primário	◆ secundário □ opcional
Finalidade	Aumentar o realismo e imersão do usuário final	
Visão Geral	Usuário informa a nova posição do objeto e o objeto "anda" na cena	
Seqüência de Eventos		1. Usuário recupera o objeto 2. Chama o método setPosition() 3. Sistema modifica sua posição 4. Sistema reposiciona o nó no grafo de cena utilizando técnicas de animação para gerar quadros intermediários.

Tabela 6 - Caso de Uso: modificar material

Caso de uso	Modificar material	
Tipo	□ primário	◆ secundário □ opcional
Finalidade	Modificar cor de um objeto	
Visão Geral	Usuário modifica o material do objeto a partir do nó que representa o mesmo no grafo de cena	
Seqüência de Eventos		1. Usuário recupera o objeto 2. Chama o método setMaterial() 3. Sistema identifica nó de material desse objeto 4. Sistema modifica o material do objeto

Tabela 7 - Caso de Uso: modificar orientação

Caso de uso	Modificar orientação	
Tipo	◆ primário	□ secundário □ opcional
Finalidade	Modificar orientação do modelo com relação ao sistema de coordenadas	
Visão Geral	Usuário informa os ângulos para rotação do modelo com relação a cada um dos eixos	

Seqüência de Eventos		1. Usuário recupera o objeto 2. Chama o método setRotation() informando o ângulo para cada eixo: x, y e z. 3. Sistema identifica matriz de rotação do objeto no grafo de cena e modifica de maneira a representar a orientação informada pelo usuário

Tabela 8 - Caso de Uso: modificar textura

Caso de uso	Modificar textura	
Tipo	□ primário ◆ secundário	□ opcional
Finalidade	Modificar textura aplicada a um objeto	
Visão Geral	Usuário identifica o arquivo de imagem que será mapeado no objeto	
Seqüência de Eventos	1. Usuário recupera o objeto 2. Chama o método setTextura() informando o nome do arquivo da imagem 3. Sistema identifica o nó de textura desse objeto e o modifica.	

Tabela 9 - Caso de Uso: Recuperar objeto

Caso de uso	Recuperar Objeto	
Tipo	◆ primário □ secundário	□ opcional
Finalidade	Dar acesso ao objeto para o usuário	
Visão Geral	Usuário informa id do objeto a ser recuperado	
Seqüência de Eventos	1. Usuário informa identificador 2. Sistema retorna referência para o objeto	

Tabela 10 - Caso de Uso: Remover objetos da cena

Caso de uso	Remover objeto da cena	
Tipo	□ primário ◆ secundário	□ opcional
Finalidade	Retirar objetos que não fazem mais parte da cena	
Visão Geral	O usuário remove o objeto pelo seu identificador	
Seqüência de Eventos	1. Usuário pede a retirada do objeto pelo identificador	

Tabela 11 – Caso de Uso: Inserir fonte de luz

Caso de uso	Inserir fonte de luz	
Tipo	◆ primário	□ secundário □ opcional
Finalidade	Aumentar realismo da cena, iluminando-a.	
Visão Geral	Usuário insere fonte de luz na cena	
Seqüência de Eventos	1. Usuário cria fonte de luz 2. Usuário insere objeto na cena 3. Sistema coloca fonte de luz em nó apropriado	

Tabela 12 - Caso de Uso: Criar fonte de luz

Caso de uso	Criar fonte de luz	
Tipo	◆ primário	□ secundário □ opcional
Finalidade	Cria fonte de luz	
Visão Geral	Usuário informa local e cor da fonte de luz	
Seqüência de Eventos	1. Criar uma nova instância de um objeto Luz 2. Usuário especifica o tipo de luz. 3. Usuário especifica a cor da luz 4. Usuário especifica posição da fonte de luz	

Tabela 13 - Caso de Uso: Recuperar fonte de luz

Caso de uso	Recuperar fonte de luz	
Tipo	◆ primário	□ secundário □ opcional
Finalidade	Usuário tem acesso à fonte de luz	
Visão Geral	Usuário informa id da fonte de luz e recebe referência	
Seqüência de Eventos	1. Usuário informa identificador 2. Sistema retorna referência para a fonte de luz correspondente	

Tabela 14 - Caso de Uso: Mover fonte de luz

Caso de uso	Mover fonte de luz	
Tipo	□ primário	◆ secundário □ opcional
Finalidade	Modificar posição da fonte de luz	
Visão Geral	Usuário informa nova posição da fonte de luz	
Seqüência de Eventos	1. Usuário recupera fonte de luz 2. Chama o método setPosition() 3. Sistema modifica a localização fonte de luz 4. Sistema reposiciona a fonte de luz no grafo de cena.	

Tabela 15 - Caso de Uso: Mudar cor da fonte de luz

Caso de uso	Mudar cor da fonte de luz	
Tipo	□ primário	◆ secundário □ opcional

Finalidade	Modificar cor de uma fonte de luz
Visão Geral	Usuário modifica a cor de uma fonte de luz
Seqüência de Eventos	1. Usuário recupera fonte de luz 2. Chama o método setColor() informando a nova cor 3. Sistema modifica a cor fonte de luz

Tabela 16 – Caso de Uso: Retirar fonte de luz

Caso de uso	Retirar fonte de luz
Tipo	□ primário ◆ secundário □ opcional
Finalidade	Retirar uma fonte de luz da cena
Visão Geral	Usuário retira fonte de luz pelo seu identificador
Seqüência de Eventos	1. Usuário informa identificador da fonte de luz 2. Sistema retira fonte de luz da cena

Tabela 17– Caso de Uso: Modificar Zoom

Caso de uso	Modificar zoom
Tipo	◆ primário □ secundário □ opcional
Finalidade	Afastar o plano de projeção do observador aumentando o tamanhos dos objetos na cena.
Visão Geral	Usuário informa a distância do observador ao plano de projeção
Seqüência de Eventos	1. Usuário informa distância do observador ao plano de projeção 2. Sistema atualiza o estado de renderização

Tabela 18 – Caso de Uso: Modificar sistema de coordenadas

Caso de uso	Modificar sistema de coordenadas
Tipo	□ primário ◆ secundário □ opcional
Finalidade	Redefinir como o observador está com relação à cena.
Visão Geral	Usuário informa novo sistema de coordenadas em coordenadas de mundo
Seqüência de Eventos	1. Usuário informa os três vetores do novo sistema coordenadas 2. Sistema atualiza estado de renderização

Tabela 19– Caso de Uso: Modificar posição da câmera

Caso de uso	Modificar posição da câmera
Tipo	◆ primário □ secundário □ opcional

Finalidade	Modificar posição do observador na cena
Visão Geral	Usuário informa nova localização do observador
Seqüência de Eventos	1. Usuário informa vetor de localização do observador 2. Sistema atualiza estado de renderização

Tabela 20 - Caso de Uso: Detectar colisão

Caso de uso	Detectar colisão
Tipo	□ primário □ secundário ◆ opcional
Finalidade	Avisar ao usuário de colisões entre objetos na cena
Visão Geral	Usuário deve informar que quer tratar colisão
Seqüência de Eventos	1. Usuário informa que precisa tratar colisão 2. Para cada iteração do sistema o motor executa o algoritmo de colisão em objetos dinâmicos 3. Sistema chama função de resposta de colisão para os objetos que colidam

Tabela 21– Caso de Uso: Exibir Imagem

Caso de uso	Exibir imagem
Tipo	◆ primário □ secundário □ opcional
Finalidade	Criar imagem da cena para ser exibida.
Visão Geral	Usuário pede a renderização da cena
Seqüência de Eventos	1. Usuário pede a renderização da cena 2. Cena é exibida na tela

5.1.2 Modelo Conceitual

Larman, em 7, afirma que "*O passo mais essencial orientado a objetos na análise ou na investigação é a decomposição do problema em conceitos e objetos individuais*".

Inicialmente criamos um diagrama de classes para o modelo conceitual baseado nos casos de uso e à medida que avançamos no desenvolvimento algumas modificações foram feitas e o diagrama foi naturalmente refinado.

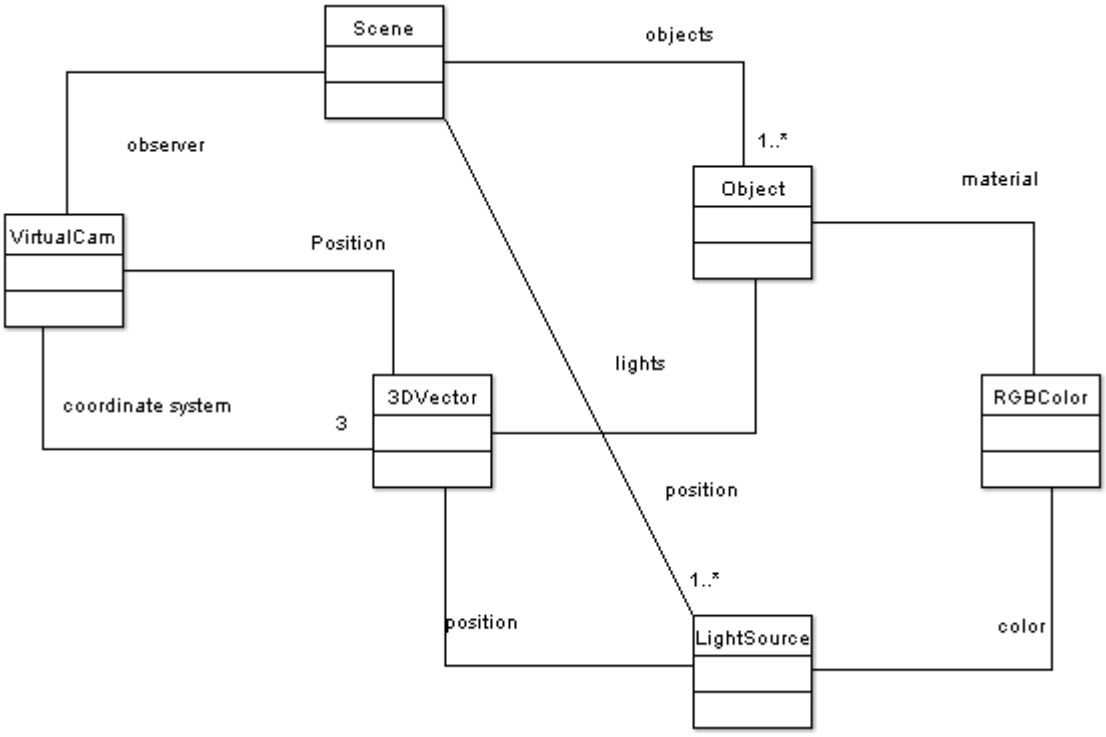

Figura 24 - Diagrama Conceitual Inicial. Conceitos presentes em um grafo de cena.

Além disso, seguindo Larman 7, criamos um modelo apenas com conceitos e depois adicionamos atributos, associações e métodos em uma segunda iteração depois da primeira versão do projeto.

Os dois passos de refinamento modificaram bastante o modelo inicial. Diversos atributos e métodos foram adicionados identificados nas seqüências de eventos dos casos de uso. E alguns conceitos foram deixados de lado no primeiro diagrama, como polígonos e geometria.

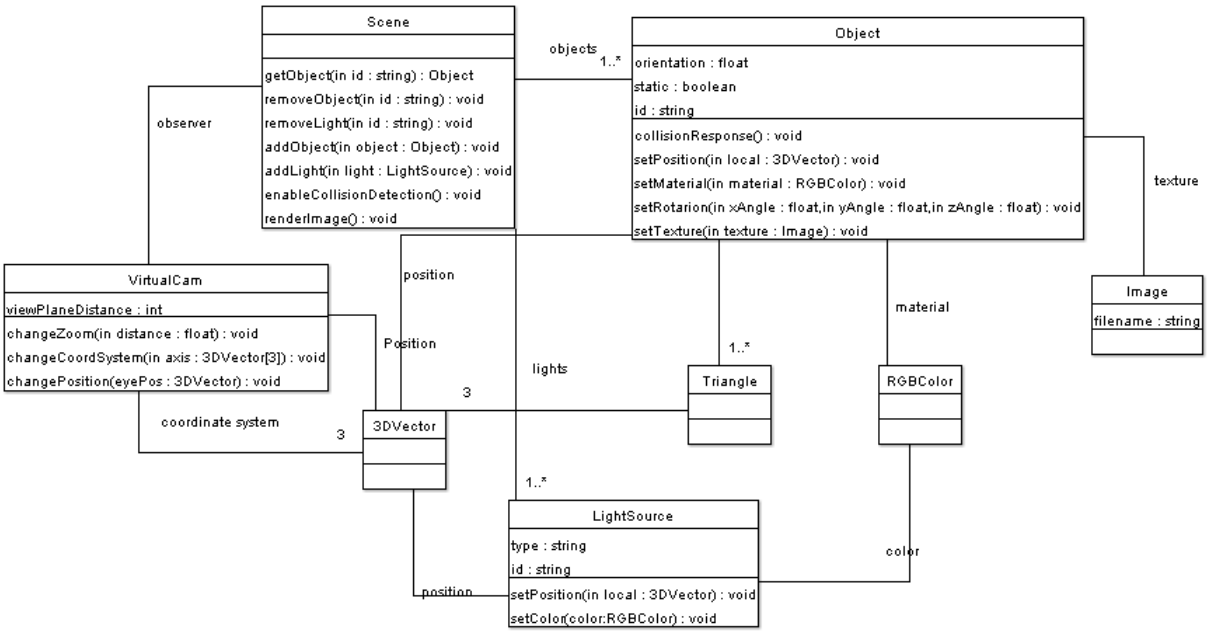

Figura 25 – Diagrama conceitual refinado.

Ainda foram encontrados alguns outros conceitos durante a criação dos diagramas de colaboração. Importante salientar que a equipe achou os diagramas de colaboração bastante valiosos, na verdade, foi a parte da análise e projeto que melhor ajudou no entendimento do problema e na proposta de solução.

5.2 Projeto

Como especificado na seção 2.1.2, nosso objetivo é criar uma arquitetura portável e que permita ótimo desempenho. Com o exposto nas próximas seções ficará claro que dividimos a engine de maneira que uma classe contenha apenas código: padrão que precisa apensa de recompilação para as diferentes plataformas, código dependente de sistema operacional – fazendo chamadas do mesmo e utilizando periféricos do hardware como som e rede - e código dependente de API gráfica contendo algoritmos executados no processador da placa de vídeo.

As classes de representação de cena utilizam algoritmos específicos – construção e testes em BSPs, Octrees, hierarquias de volume - executados pelo processador. Dessa maneira utilizamos apenas código padrão da linguagem C++ para sua implementação. Essa código precisa apenas ser recompilado para as diversas plataformas de execução.

Os códigos dependentes de sistema operacional são implementados pelos Visitors[16] enquanto que códigos específicos para uma API gráfica são implementados pelos Handlers. Dessa maneira, uma implementação Linux e outra Windows dos Visitors podem utilizar as mesmas implementações OpenGL dos Handler diminuindo o trabalho ao portar códigos e aumentamos o reuso.

Apenas uma implementação das chamadas a API gráfica também contribuem para a melhoria de performance. A manutenção de apenas uma implementação é mais simples e pode ser melhor trabalhada que a de várias implementações.

5.2.1 Responsabilidades dos Objetos

Larman, em 7, afirma que "*... um passo essencial (na decomposição dos requisitos funcionais em classes) é a alocação de responsabilidades aos objetos e a ilustração de como eles interagem através de mensagens expressas em diagramas de colaboração*".

Nessa seção, serão apresentados os diagramas das principais operações-renderização e detecção de colisão - ilustrando mensagens entre os objetos com o intuito de identificar as responsabilidades de cada um deles.

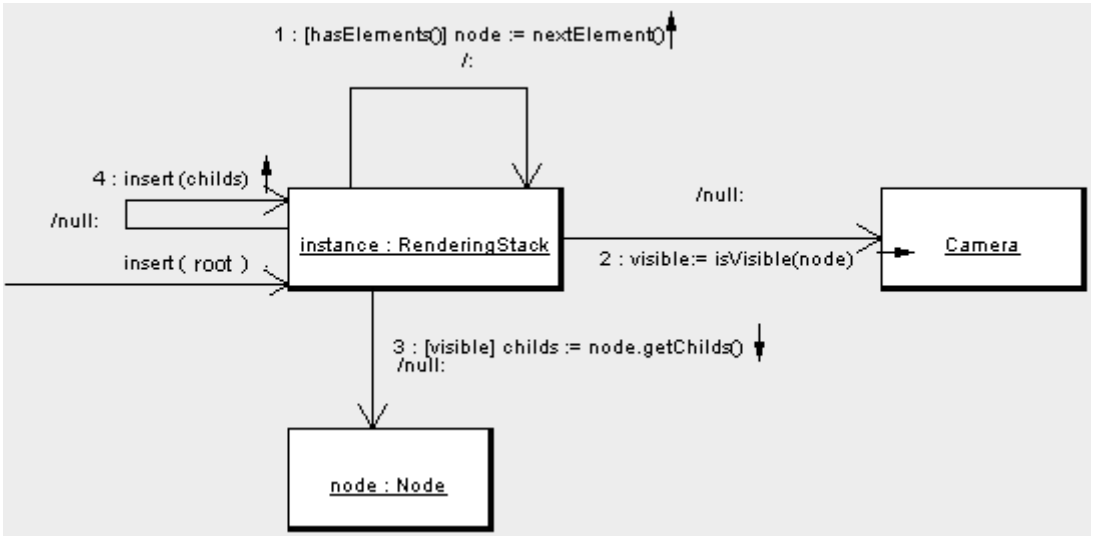

Figura 26 – Diagrama de Colaboração. Renderização.

O diagrama de colaboração da operação de renderização mostrou a necessidade da criação de uma pilha para os nós possivelmente visíveis. A cada nó visitado, faz-se o teste de visibilidade e em caso positivo adicionamos o nó à pilha.

[16] Os papeis de Visitors e Handlers na arquitetura do motor é explicado nas próximas seções.

Enquanto ainda estiverem nós na pilha precisamos visitá-los. Os nós de geometria que estiverem na pilha são renderizados.

Além disso, a renderização precisa de dois passos. O primeiro para identificar os nós realmente visíveis – operação de *culling*. Depois, os nós são visitados com uma abordagem *bottom-up*, para aplicarmos os decoradores da raiz ao nó de geometria.

A detecção de colisão ocorre sempre que um objeto dinâmico modifica sua posição. Dessa maneira, mantemos uma lista de objetos que se moveram e que devem fazer o teste de colisão.

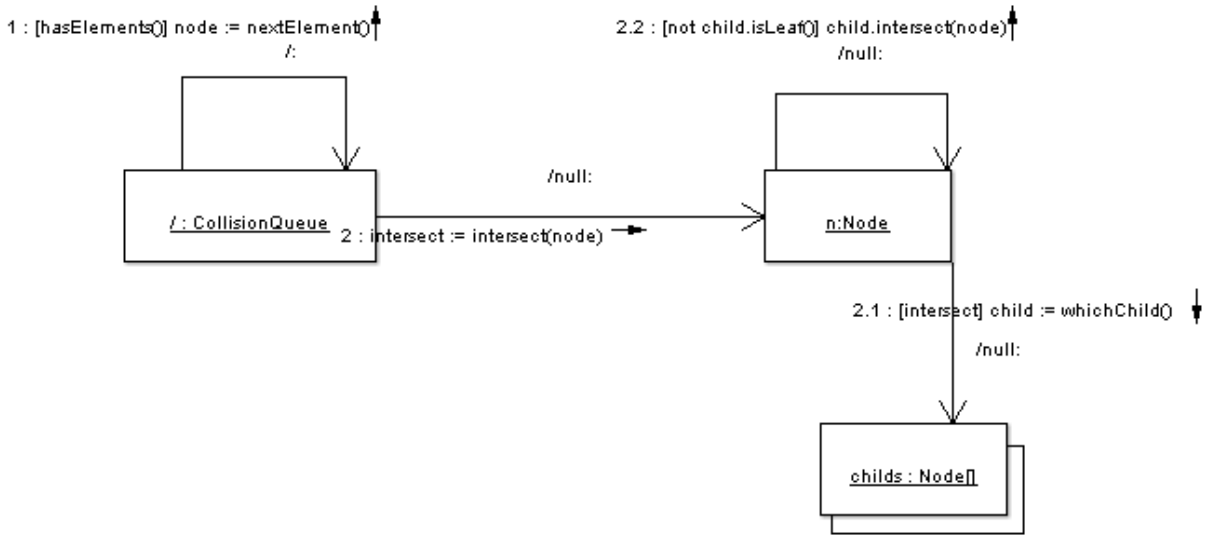

Figura 27. Diagrama de Colaboração. Colisão.

O teste de colisão é feito comparando a subárvore do objeto movido com toda hierarquia da cena. Caso encontre alguma colisão, o sistema faz a chamada da função de *callback* determinada para aquele objeto.

5.2.2 Arquitetura

O motor possui basicamente três componentes centrais. O SceneManager gerencia a cena e é o componente do motor mais visível para o usuário. Os visitors são utilizados para executar as operações necessárias à renderização da cena. O HandlerRegistry contém as implementações dos Handlers dos decoradores da cena.

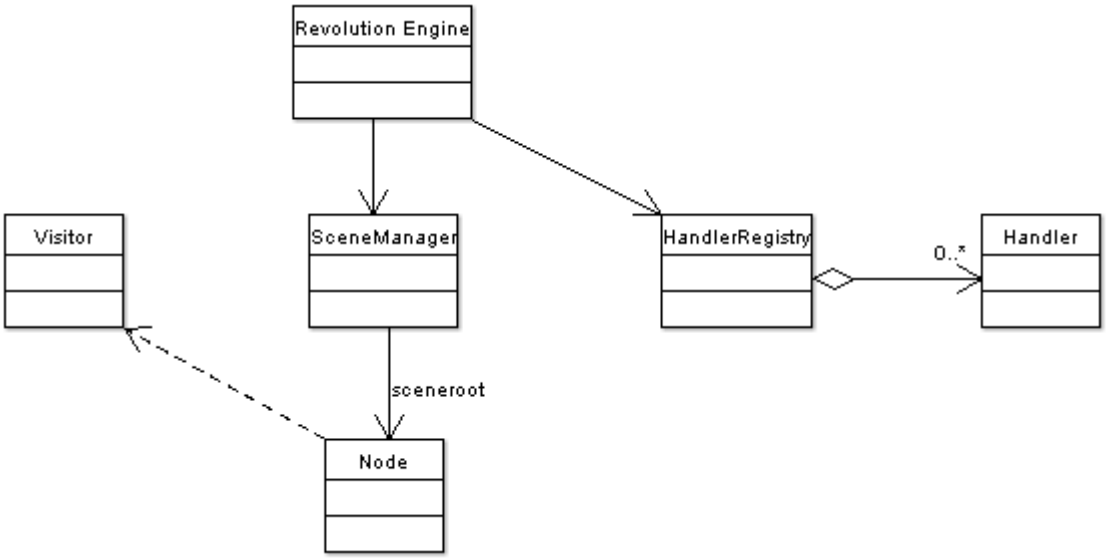

Figura 28 - Arquitetura. Visão Geral antes do refinamento.

O SceneManager possui o grafo de cena e é responsável pelas operações de inserção, remoção e alteração de objetos no mesmo. O SceneManager gerencia as visitações no grafo de cena. Todas as operações sobre o mesmo são feitas por visitors.

Os visitors padrões são implementados pelo motor. Outras operações podem ser feitas criando novos visitors e adicionando-o a lista de operações do SceneManager.

A operação de renderização faz duas visitas a hierarquia. Por esse motivo, decidimos dividi-la em dois visitors: RenderingVisitor e Culling Visitor.

O grafo de cena possui basicamente três tipos de nós: nós de cena, nós decoradores e nós de geometria.

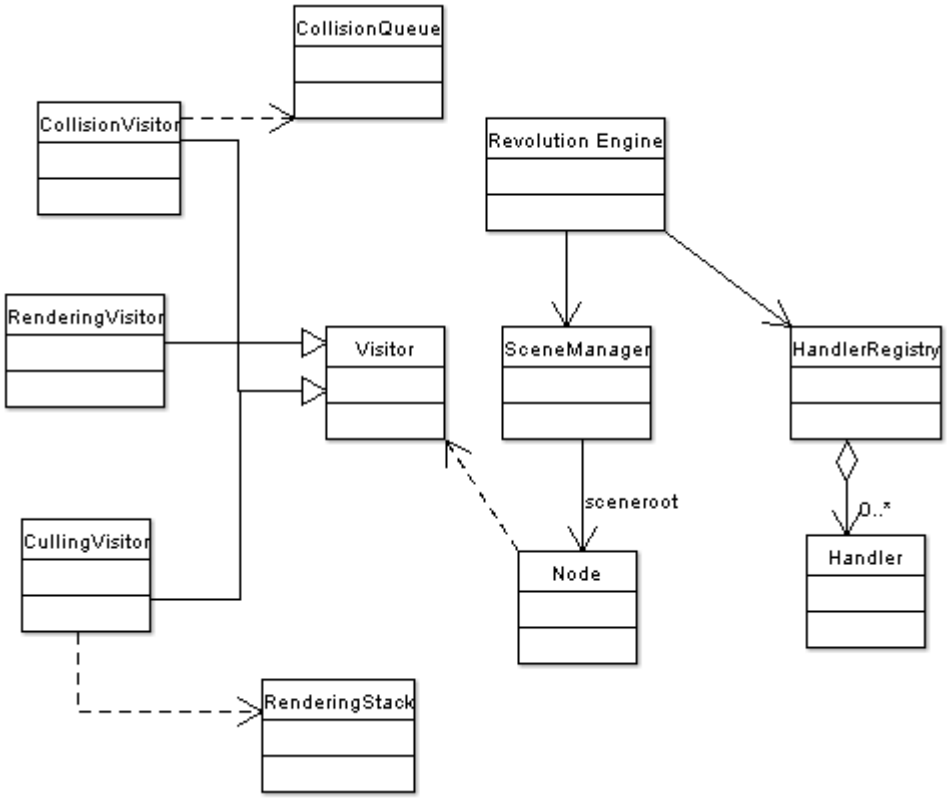

Figura 29 – Diagrama Geral Refinado após diagramas de colaboração.

Os nós de cena (SceneNode) compõem a cena em si e podem implementar algoritmos de ordenação espacial ou de hierarquias de objetos. Dado os pontos positivos das duas abordagens o motor da à possibilidade da utilização de uma abordagem híbrida. Fica, dessa maneira, a cargo do usuário do motor – o desenvolvedor do jogo – utilizar as técnicas da melhor maneira possível visto que o mesmo conhece o contexto no qual está trabalhando.

Objetos de cena são modelos em três dimensões e dessa maneira são descritos como um conjunto de polígonos. Tais objetos podem ser classificados entre dinâmicos e estáticos. Os objetos estáticos não modificam sua posição durante o tempo de execução da aplicação e por isso as operações de detecção de colisão e atualizações não são aplicadas ao mesmo.

Por outro lado, objetos dinâmicos sofrem essas operações. Essas operações muitas vezes são diferentes para os diferentes objetos. A atualização da posição de um modelo humanóide – animação – não acontece da mesma maneira que a atualização de um modelo de carro por exemplo. Da mesma maneira, a resposta de uma bala na colisão com uma parede é bem diferente na sua colisão com a água. Assim, essas operações devem ser implementadas pelo usuário.

Para essas operações o motor utilizará a técnica de delegates. Os métodos dos objetos de cena para resposta de colisão e atualização são apenas wrappers para a implementação dos mesmo pelo usuário. Importante frisar que esses algoritmos, normalmente, utilizam apenas código padrão. Assim, essa abordagem, não afeta de maneira considerável a portabilidade.

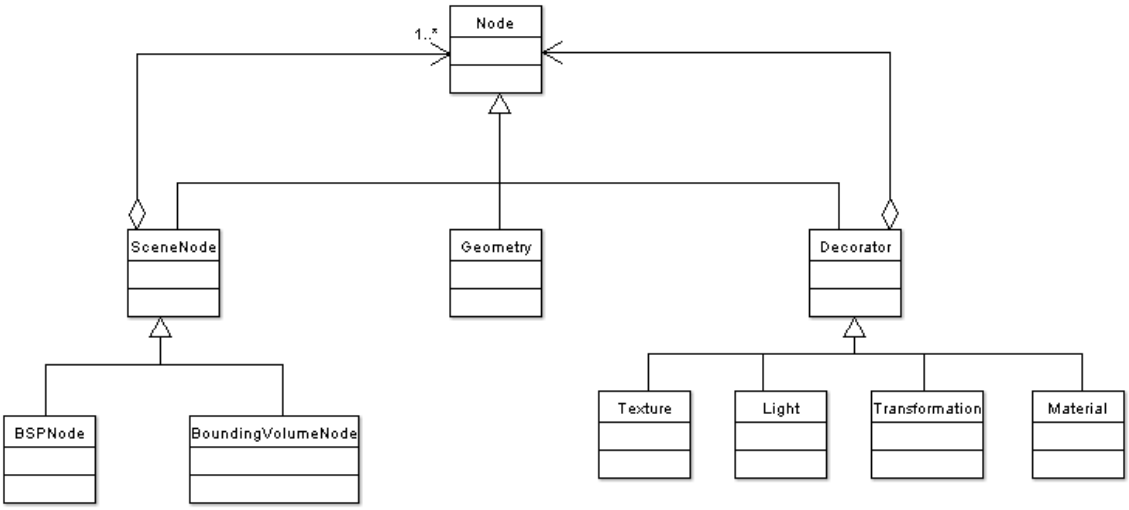

Figura 30 – Arquitetura final do Grafo de Cena.

Nós decoradores incluem os elementos físicos, matemáticos e computacionais na cena. Portanto, não possuem geometria, mas afetam de diferentes maneiras a configuração da cena.

Por fim, os nós de geometria possuem o conjunto de triângulos que formam um modelo a ser renderizado. Esses nós são sempre as folhas da árvore no grafo de cena.

Um polígono renderizado possui diversas propriedades que definem a cor, localização e textura de seus vértices. Tais propriedades são especificadas no caminho da árvore desde a raiz até o nó folha que possui o vértice.

Tais propriedades formam o que chamamos de estado de renderização. Um triângulo é desenhado na tela de acordo com o estado atual da renderização. Dessa maneira a travessia do visitor pelo grafo aplica essas propriedades – especificadas nos decoradores – ao estado de renderização atual.

Cada caminho da raiz até uma folha pode ser visto como a especificação de um estado de renderização da geometria da folha. Importante salientar que muitas vezes caminhos diferentes podem criar a mesma especificação. Existem diversos algoritmos para otimização do grafo de cena que se preocupam em minimizar o

grafo maximizando o uso de memória. Tais algoritmos estão fora do escopo deste trabalho, mas a implementação deles pode ser feita com a implementação de visitors adicionais.

Para a implementação todos esses diagramas foram refinados para refletir classes de implementação. O objetivo deste trabalho era somente a criação da arquitetura e para ater o foco em problemas nesse nível de abstração esses diagramas foram omitidos.

5.2.2.1 Padrões de Projeto

Diversos padrões de projetos foram utilizados na arquitetura proposta com o objetivo de criar mais flexibilidade no motor e de ter uma arquitetura com componentes bem separados para facilitar atualizações independentes.

O grafo de cena é conceitualmente uma estrutura hierárquica e para isso utilizamos o padrão de projeto Composite na sua modelagem.

Para conceitos adicionais utilizamos o padrão Decorator. O decorator não só representa esses conceitos como é facilmente integrado com o padrão composite utilizado para o grafo de cena. Poderíamos ter utilizado o padrão Observer para atualização da hierarquia, mas avaliamos que essa era uma decisão de implementação, e já utilizamos um operador de atualização na forma de Visitor.

O padrão mais impactante é o Visitor. Todas as operações para o caso de uso mais importante, renderizar cena, são implementados com visitors. Importante salientar que grande parte da separação dos códigos para proporcionar melhor portabilidade foi garantido com os Visitors. Idéia também usada no OpenSceneGraph e deixada de lado, duplicando códigos, no Ogre3D.

Além disso, padrões tradicionais foram utilizados. O Facade foi utilizado para a entrada do motor. A classe principal RevolutionEngine, visível pelo programado, é um facade. E diversos Singletons foram utilizados, o principal deles é o RegistryHandler.

Por ser uma fonte comum de recursos, o RegistryHandler precisa ter somente uma instância para que não ocorre divergências. Evita, por exemplo, que o usuário registre um Handler em um registro diferente do acessado pelos visitors.

5.2.2.2 Register & Handlers

Para cada decorador na árvore é necessário um handler que aplicará o decorador ao estado da renderização. Dessa maneira, códigos específicos da API gráfica – OpenGL ou Direct3D- são implementados no Handler e não no nó decorador.

Para decoradores padrões – transformações, texturas, fontes de luz, etc. – o motor terá implementação própria, mas o usuário pode modificar o handler responsável utilizando o HandlerRegistry. Para isso basta registrar outro Handler para aquele decorador. Dessa maneira o usuário pode, por exemplo, utilizar um modelo de iluminação específico, para a subárvore filha desse decorador, que contém uma simulação de oceano[17].

Para a utilização de um Handler, é necessário o registro do mesmo. O usuário deve para isso, utilizar o HandlerRegistry, responsável por delegar a responsabilidade ao Handler apropriado. Caso o HandlerRegistry não conheça um Handler apropriado, uma exceção é levantada.

A implementação do Register utiliza mecanismos baseados na tecnologia Component Object Model (COM) 7 da Microsoft. Em COM, cada componente faz o registro no registro do Windows normalmente durante a instalação da aplicação que o utilizada. Cada componente tem um nome para identificá-lo e dessa maneira evitar duas instalações diferentes do mesmo componente. O nome também é utilizado pela aplicação para carregar o componente.

Analogamente, os handlers terão identificadores. Assim o usuário pode ter diversas implementações de handlers para o mesmo decorador mas deve especificar ao HandlerRegistry qual utilizar. Se acontecer algum conflito entre os Handler uma exceção é lançado.

Todos os handlers utilizados devem ser implementados utilizando à mesma API Gráfica. O jogo deve usar somente uma API gráfica devido ao ambiente de execução utilizado. Em Windows, por exemplo, cada janela tem um contexto no qual são criados os componentes de interface. Para as diferentes APIs – OpenGL, Direct3D, etc. – o contexto tem configuração diferente o que impossibilita a utilização de API diferentes no mesmo contexto. O mesmo vale para outros ambientes de execução.

---

[17] Oceanos utilizam modelos de iluminação bem diferentes dos padrões por terem um comportamento próprio com relação às leis da física de refração e reflexão da luz.

5.2.3 Extensões

O Motor pode ter sua funcionalidade entendida basicamente de duas maneiras: novos objetos de cena ou novos decoradores. Objetos de cena aumentam as possibilidades de representação da cena enquanto novos decoradores podem ser usados para utilizar diferentes modelos de iluminação bem como fenômenos físicos mais específicos do contexto simulado.

Novos objetos de cena podem ser adicionados para utilizar novas técnicas de representação caso essas venham a surgir. Dessa maneira, a implementação dos visitors deve ser entendido para tratar os novos nós.

Os operadores – nos visitors – padrões, implementados no motor, podem ser reimplementados ou estendidos por meio de herança. Na reimplementação o usuário deve refazer as operações inclusive para os nós pré-existentes. Na herança basta a implementação dos nós adicionados.

Uma extensão interessante é, por exemplo, a de animação de humanóides. Pode ser criado um nó AnimatedMesh, herdeiro de SceneNode, que executa animações durante a atualização da sua localização. Assim, um AnimationUpdateVistor que utiliza técnicas de animação para criar o efeito correto implementa o método visit do Visitor para objetos do tipo AnimatedMesh, o polimorfismo garante que esse será o método utilizado.

Para novos decoradores a extensão é bastante simples e foi explicada na seção 5.2.2.2. Basta que o usuário crie o decorator e o handler e registre o Handler no HandlerRegistry. Os visitor automaticamente procuram por Handlers no HandlerRegistry e a unicidade do registro evita que o handler não seja encontrado.

5.3 Implementação

Como citado na seção 2.1.3, existe um trabalho de graduação complementar, de Leonardo Costa, responsável pelo refinamento do modelo aqui apresentado e ainda a identificação das melhores tecnologias para a implementação. Dessa maneira, serão apresentadas as escolhas, com uma breve motivação, para maiores informações consultar o trabalho de Leonardo Costa.

A escolha padrão na indústria para implementação é utilizar a linguagem de programação C++. Apesar dos avanços de performance de Java, a máquina virtual ainda é deficiente no gerenciamento de memória e ainda tem desempenho aquém do possível com boas práticas utilizando C++.

A indústria ainda vê com certa desconfiança a plataforma .NET7 da Microsoft devido a diversos problemas de instabilidade. No entanto, a estratégia da Microsoft na criação do XNA7 é visto com bons olhos já que pretende ter compiladores para PCs assim como XboX7, evitando assim duas implementações ou códigos portáveis. Mas são apenas promessas para o futuro.

Na primeira versão, foi implementado um protótipo de motor para Windows utilizando a API gráfica OpenGL7.

O sistema operacional Windows foi escolhido por ter a maior base de usuários para PC[18] e ser a plataforma padrão dos usuários de jogos. O Windows possui também documentação vasta para suas diversas interfaces de programação (API) além de ambientes de desenvolvimento consolidados como Visual Studio . NET 20037.

A API gráfica OpenGL é utilizada por ter uma curva de aprendizado menor e ter ainda uma plataforma melhor elaborada que as demais opções. Além disso, experiências na indústria mostram que portar código OpenGL para APIs como Direct3D é menos trabalhoso que o contrário.

Em vários pontos da implementação foi utilizada a tecnologia Component Object Model (COM)7 da Microsoft para prover uma maior componentização do código e, consequentemente, facilidade de atualização.

As atualizações pós-venda de jogos são muito comuns na indústria de jogos. O time-to-market é bastante pequeno e o calendário apertado impede um jogo sem bugs. Além disso, diversos jogos possuem expansões que estendem o conceito do jogo em si com novas fases, novos personagens, entre outras coisas. Faz-se então necessária utilização de tecnologias para componentização do código que facilitem atualizações sem recompilação, não só por motivos tecnológicos como pelo próprio modelo de negócios praticado na indústria.

5.4 Resultados

Durante o desenvolvimento deste trabalho, um pequeno protótipo da engine foi implementado para um ambiente de execução com Windows e Direct3D. O propósito desse protótipo é testar o desempenho da engine já que não houve tempo

[18] A plataforma ideal seria para consoles, os vídeo games. No entanto, os ambientes de programação dos mesmos têm um custo alto e não são comercializados no Brasil.

hábil para criar diferentes implementações e testar a facilidade de extensão e portabilidade do motor. Apesar de que alguns testes ainda foram feitos.

Foram implementados, até agora, somente nós de hierarquia de volume utilizando volumes alinhados com os eixos (AABB), nenhum algoritmo de ordenação espacial foi feito, BSP ou Octree. Preferimos focar nossos esforços nos decoradores para podermos criar testes de extensão nos mesmos.

Foi implementado, para testes do registro de handlers, alguns decoradores como fonte de luz e texturas e, para testes de extensão, efeitos como sistemas de partículas[19] e "lens flare"[20]. Um skybox[21] também foi criado para cenas abertas.

Nas imagens apresentadas podem ser vistos o flare e o skybox. O céu no horizonte foi criado utilizando o skybox.

Um pequeno jogo, o 7Seas, foi criado para demonstrar algumas das qualidades técnicas – performance e extensibilidade – do motor. Foi o primeiro jogo em três dimensões da cadeira de jogos do Centro de Informática e o grupo obteve nota máxima na disciplina.

Tabela 22 - Máquinas de teste

Máquina	Processador	Memória	Placa de Vídeo
1	Athlon XP 2000+	512 MB	geForce 4 MX440 64MB
2	Athlon XP 1800+	512 MB	-
3	Pentium III 800Mhz	128 MB	ATI Mobility 8MB

Foram feitos os testes de performance e qualidade em poucas máquinas. A quantidade de máquinas é limitada devido às máquinas que tivemos acesso durante a produção do jogo, no entanto os resultados têm alguma expressividade e são

[19] Um sistema de partículas é uma técnica que simula certos fenômenos "caóticos" – como fogo, fumaça, água, etc. – que são fisicamente complexos e uma abordagem realista são complicadas de implementar.

[20] Fenômeno que acontece em imagens produzidas por sistemas óticos – câmera fotográfica, por exemplo – quando apontados para uma fonte de luz brilhante. Normalmente se manifesta na forma de estrelas, anéis ou círculos em uma linha cortando a imagem.

[21] Um skybox é uma caixa pré-renderizada utilizada para criar a ilusão de fronteiras distantes. Normalmente são imagens de montanhas ou do próprio céu.

analisados de acordo com a Tabela 23 - Benchmarks. Testes de Performance qualidade.

As máquinas 1 e 2 tinham Windows XP instalado enquanto que a máquina 3 tinha o Windows 2000. Todas as máquinas tinham o ambiente de execução 9.0c April Update do DirectX.

Figura 31 - 7Seas na Máquina 1. Lens Flare aumentam realismo da cena.

Infelizmente não houve tempo suficiente antes da confecção desse documento implementar os Handlers do motor em OpenGL para testes com outra API gráfica. No entanto, essa implementação está sendo feita para avaliar melhor a qualidade do motor.

Tabela 23 - Benchmarks. Testes de Performance qualidade

Máquina	Quadros por segundo	Qualidade Visual
1	31.39	Cena renderizada com o lens flare, sombra dos barcos e ainda reflexão da iluminação na água.
2	15.06	Lens flare renderizado mas sem sombra dos barcos e reflexão na água.

3	1.80	Lens flare renderizado mas sem sombra dos barcos e reflexão na água e com quadros por segundo insuficientes.

Durante os testes todo o processamento de rede, som e lógica do jogo foram desabilitados para demonstrar a performance real do motor sem influência de algoritmos não relacionados.

De acordo com os resultados dos testes podemos ver que o motor é bastante escalável devido ao aumento de performance e da qualidade visual[22] com o aumento do poder de processamento da máquina.

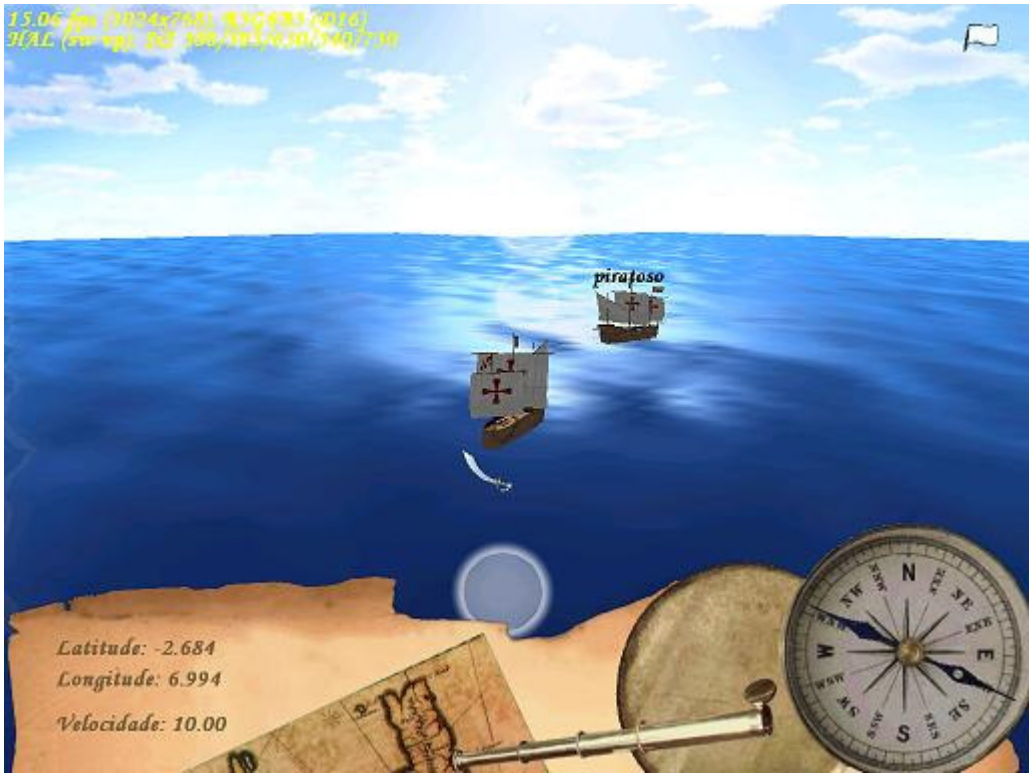

Figura 32 - 7Seas na Máquina 2.

Apesar dos bons resultados, inclusive pioneiros no Centro de Informática, diversos aspectos devem ser melhorados e diversas extensões são necessárias para contemplar todos os requisitos básicos de um motor tradicional e este trabalho deve ser feito nos próximos anos em outros trabalhos de graduação e dissertações de mestrado. Mais detalhes sobre extensões no capítulo 6.

[22] Qualidade visual é um conceito bastante subjetivo mas devido a grande diferença nas imagens e as diversas pessoas com a mesma opinião temos um resultado bem preciso.

Figura 33 - 7Seas na Máquina 3.

6 Conclusões e Trabalhos Futuros

Ficou claro pelas citações anteriores que este trabalho compreende apenas uma pequena parte de um motor gráfico. No entanto, foi criado o arcabouço do motor e as extensões podem ser facilmente inseridas. Dessa maneira, pretende-se utilizar o motor em cadeiras do Centro de Informática para incentivar a produção de sistemas interativos em três dimensões bem como a pesquisa na área.

Como salientamos em 4.3, e enfatizado por Bethel et al, em 7, a representação utilizando grafos de cena tem limitações diversas e que talvez tenham impacto no desenvolvimento dessa abordagem. Devido ao tempo hábil para confecção deste trabalho nosso objetivo foi apenas de estender e tentar melhorar mesmo sabendo das limitações e que uma mudança de paradigma talvez fosse mais apropriada.

A criação de visitors de otimização é outra possibilidade para contornar os problemas de concorrência inerentes ao grafo de cena. Essa possibilidade está sendo explorar com bastante sucesso na implementação de Java 3 d 7 . Esses visitors de otimização podem localizar e agendar operações de mudança de estado para possibilitar concorrência. Estruturas bem estruturadas agilizam diversas operações como detecção de colisão e picking[23].

Por falta de usuário e experts na área nossa elicitação de requisitos foi bastante limitada, mas compreendemos que o resultado geral do trabalho foi pouco afetado por esse problema. No entanto, uma atualização constante do motor é necessária para renovação e aperfeiçoamento do mesmo a medida que novos requisitos e casos de uso forem surgindo.

[23] Operação de seleção de um objeto em três dimensões na cena a partir de uma posição de duas dimensões na tela.

7 Referências Bibliográficas

[1] Eberly, David H.; 3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics. Morgan Kaufmann, 2000.

[2] Eberly, David H.; 3D Game Engine Architecture: Engineering Real-Time Applications with Wild Magic. Morgan Kaufmann, 2003.

[3] Id Software: Doom 3. http://www.idsoftware.com/games/doom/doom3/ , Maio 2005.

[4] Bishop, Lars; Eberly, Dave; Whitted, Turner; Finch, Mark; Shantz, Michael; Designing a PC Game Engine. IEEE Computer Graphics and Applications Volume 18, Issue 1, 1998. Pages: 46-53.

[5] Macedo Júnior, Ives J. A. mOGE: Mobile Game Engine. Universidade Federal de Pernambuco, 2005.

[6] Eduardo Rocha, Talita Menezes, Mauro Florêncio, Geber Ramalho e André Santos, Forge 16v: um Framework para Jogos Isométricos, II Workshop on Games and Digital Entertainment (WJogos'03), SBC, Salvador, 4-5 novembro 2003.

[7] OGRE 3D: Open source graphics engine. http://www.ogre3d.org/, Agosto 2005.

[8] OpenSceneGraph. http://www.openscenegraph.org/, Agosto 2005.

[9] Irrlicht Engine – A free open source 3d engine. http://irrlicht.sourceforge.net/, Agosto 2005.

[10]Harrison, Lynn T., Introduction to 3D Game Engine Design Using DirectX 9 and C#. Apress, 2003.

[11]Bethel, Wes; et. al. Scene Graph API: Wired or Tired? Panel Proposal, Siggraph 99.

[12]Larman, Craig. Utilizando UML e Padrões: Uma introdução a analise e ao projeto orientado a objetos. Bookman, 2000.

[13]Screen Digest, Global Media market research and analysis. http://www.screendigest.com/, Agosto 2005.

[14]Duchaineau, Mark; et al. ROAMing Terrain: Real-time Optimally Adapting Meshes. SIGGRAPH 1997.

[15]Lindstrom, Peter; Pascucci, Valerio. Visualization of large terrains made easy. Proceedings of IEEE Visualization 2001.

[16]Devmaster.net – Your source for game development. http://www.devmaster.net/, Agosto 2005.

[17]OpenGL – The Industry Standard for High Performance Graphics. http://www.opengl.org/, Agosto 2005.

[18]Microsoft DirectX: Home Page. http://www.microsoft.com/windows/directx/default.aspx, Agosto 2005.

[19]DirectX Graphics. http://www.msdn.microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/graphicsmultimedia.asp, Agosto 2005.

[20]Torque Store:. Garage Games. http://www.garagegames.com/makegames/, Agosto 2005.

[21]Renderware. http://www.csl.com/, Agosto 2005.

[22]Unreal Engine 3 – Unreal Technology. http://www.unrealtechnology.com/html/technology/ue3.shtml/, Agosto 2005.

[23]Reality Engine. http://www.artificialstudios.com/products.php, Agosto 2005.

[24]Gamebrio from NDL – Free your mind. http://www.ndl.com/, Agosto 2005.

[25]COM: Component Object Model Technologies. http://www.microsoft.com/com/default.mspx, Agosto 2005.

[26]Microsoft.NET Home Page. http://www.microsoft.com/net/default.mspx, Agosto 2005.

[27]Microsoft XboX. http://www.microsoft.com/xbox/, Agosto 2005.

[28]Microsoft XNA. http://www.microsoft.com/xna/, Agosto 2005.

[29]Java Technology. http://java.sun.com/, Agosto 2005.

[30]Visual Studio Home. http://msdn.microsoft.com/vstudio/, Agosto 2005.

[31]Charalambous, M.; Trancoso, P.;Stamatakis, A. Initial Experiences Porting a Bioinformatics Application to a Graphics Processor. Proceedings of the 10th Panhellenic Conference in Informatics (PCI 2005).

[32]Galoppo, N. et al.; LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware To appear in Proceedings of the 2005 ACM/IEEE Super Computing Conference. November 12-18, 2005.)

[33]M Lin, D Manocha, J Cohen, S Gottschalk. “Collision detection: Algorithms and applications”. Proceedings of the 2nd Workshop on Algorithmic Foundations, 1996

[34]MC Lin, S Gottschalk, D Manocha. “Obbtree: A hierarchical structure for rapid interference detection.” Computer Graphics (SIGGRAPH Conference Proceedings), 1996.

[35]Hall, Roy; Illumination and Color in Computer Generated Imagery. Springer-Verlag.1989.

[36]Bouknight, W. J. (1970), “A procedure for Generation of Three-dimensional Half-toned Computer Graphics Presentations,” Communications of ACM, vol. 13, no. 9, pp. 527-536.

[37]Sun’s Java3D home Page. http://java.sun.com/products/java-media/3D/ , Agosto 2005.

8 Apêndice A

Diagramas no tamanho original. Esse Apêndice tem a página com orientação diferente para que os diagramas possam ser vistos claramente.

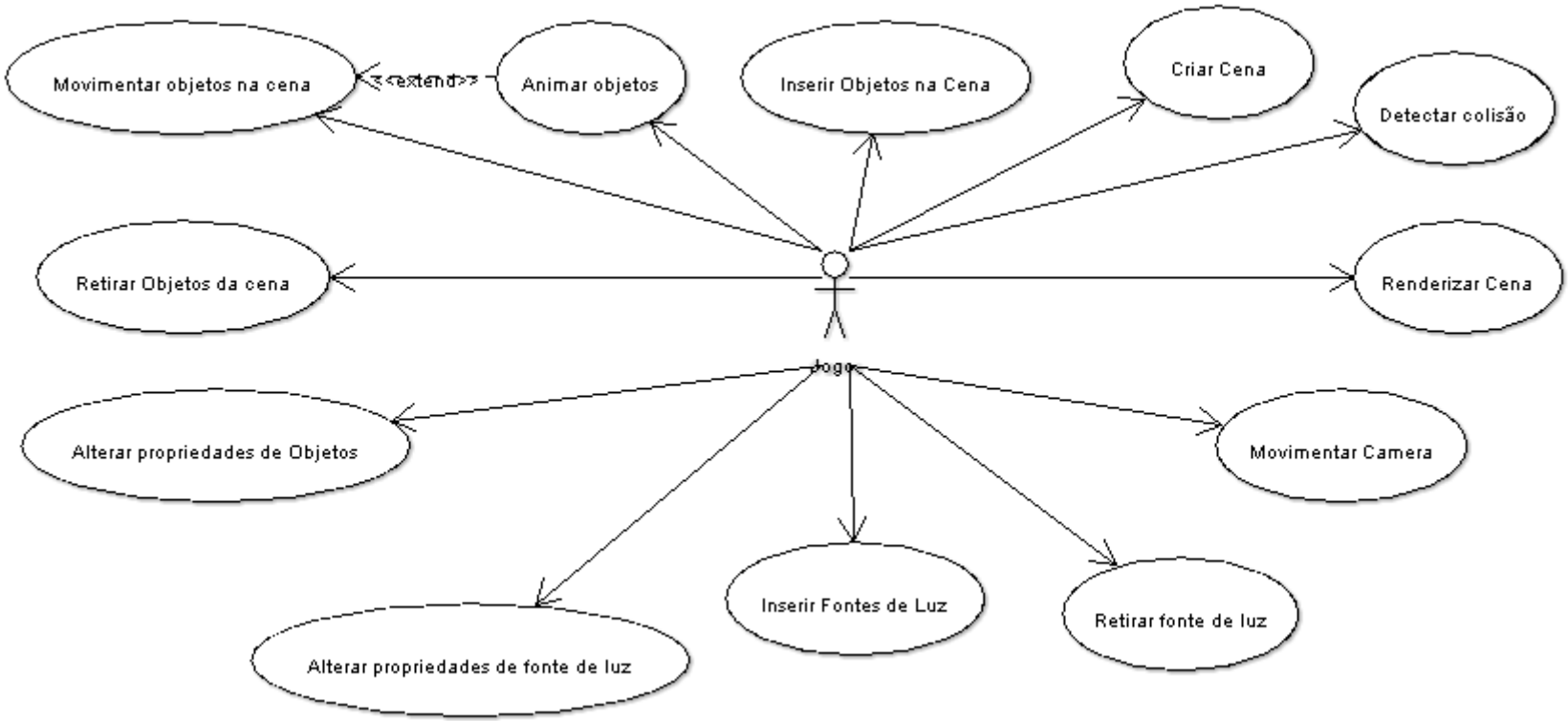

Figura 34 – Reconstrução de Figura 18 - Casos de Uso. Análise inicial baseada somente nos requisitos.

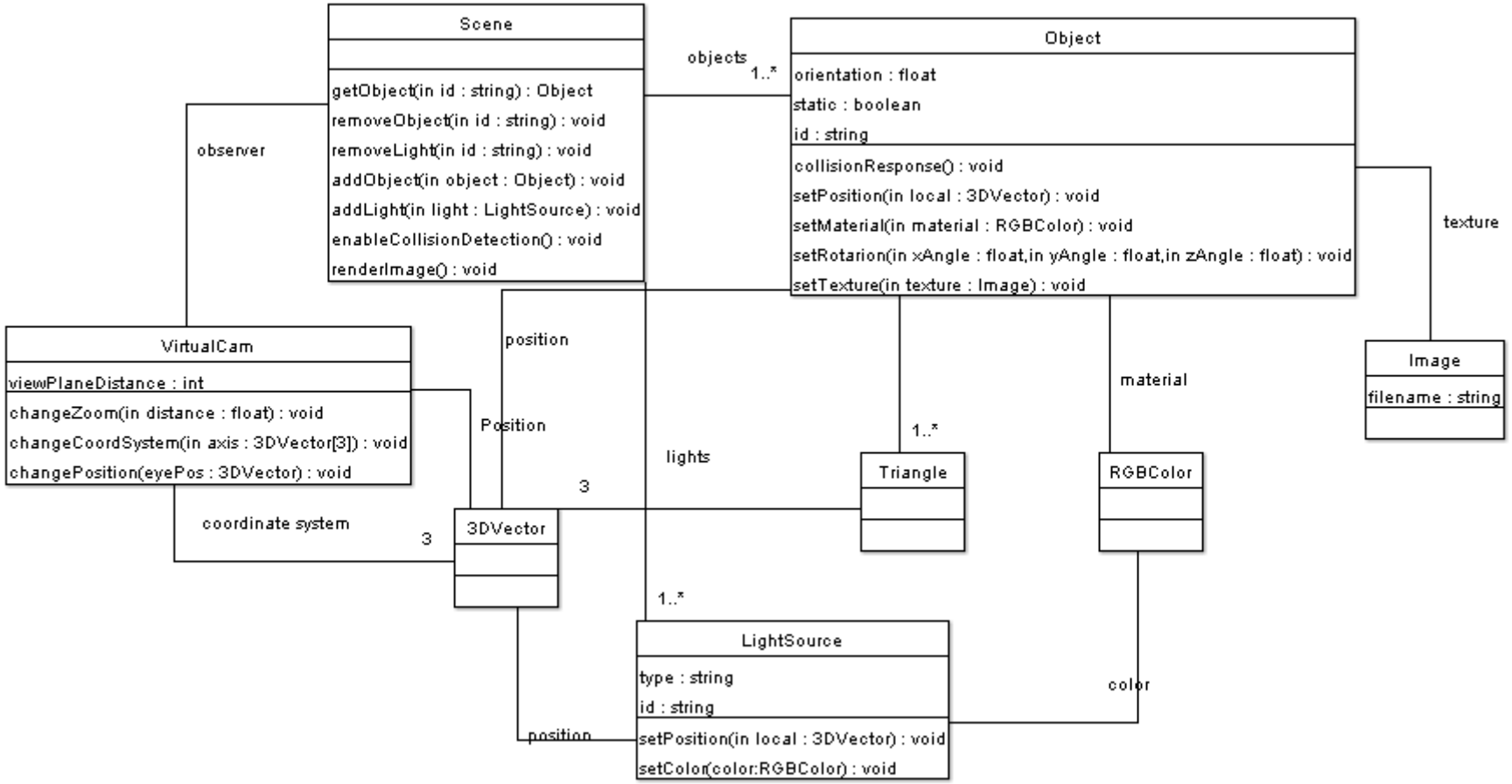

Figura 35 – Reconstrução de Figura 25 – Diagrama conceitual refinado.

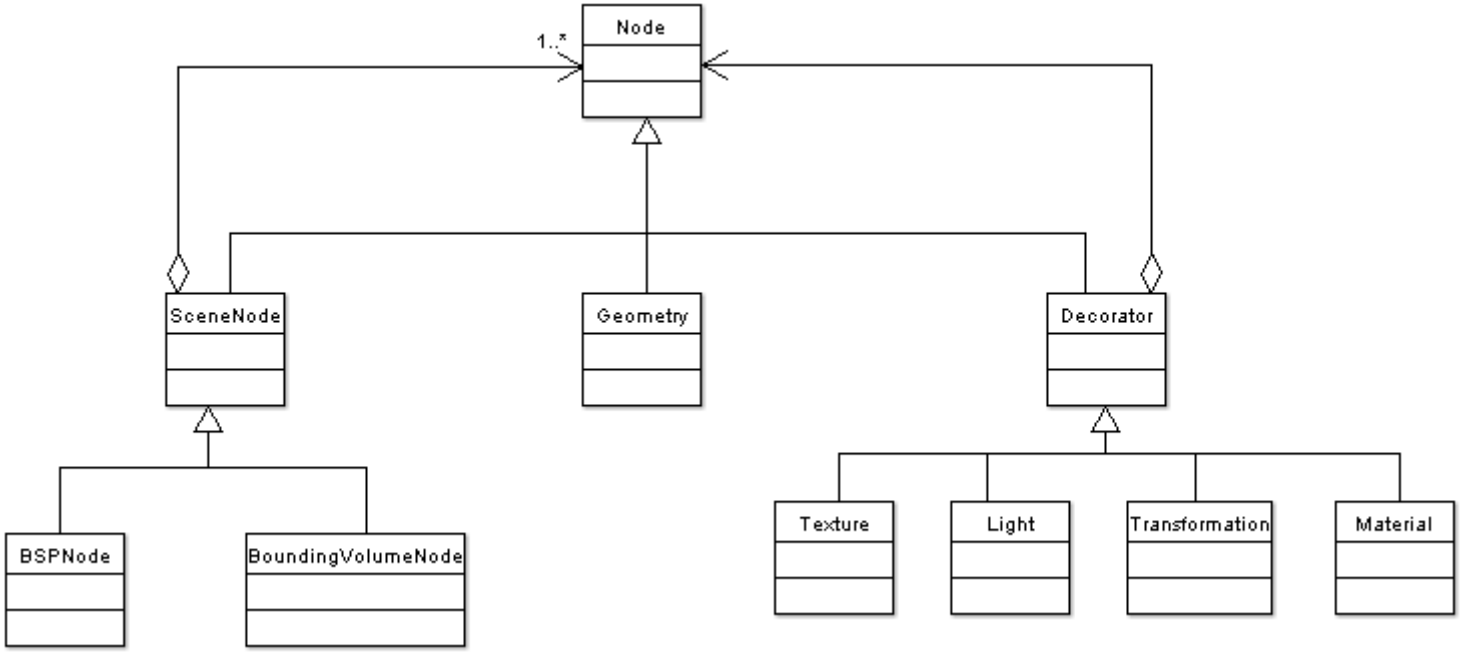

Figura 36 – Reconstrução de Figura 28 - Arquitetura. Visão Geral antes do refinamento.