DEPARTAMENTO DE COMPUTAÇÃO

D E C O M

CIC391 - Monografia - 2007/2

"VRScript: Interatividade em tempo-real com simulações tridimensionais"

Aluno:
Douglas Oliveira Matoso

Orientador:
Prof. Dr. Tiago Garcia de Senna Carneiro

U F O P

UNIVERSIDADE FEDERAL DE OURO PRETO

FOLHA DE APROVAÇÃO DA BANCA EXAMINADORA

VRScript: Interatividade em Tempo-Real com Simulações Tridimensionais

Douglas Oliveira Matoso

Monografia apresentada ao Departamento de Computação do Instituto de Ciências Exatas e Biológicas da Universidade Federal de Ouro Preto como requisito parcial da disciplina CIC391 – MONOGRAFIA, do curso de Bacharelado em Ciência da Computação, e aprovada pela Banca Examinadora abaixo assinada

_____

Prof. Dr. Tiago Garcia de Senna Carneiro
Doutor em Computação Aplicada pelo
INPE - Instituto Nacional de Pesquisas Espaciais
Orientador
Departamento de Computação – UFOP

_____

Prof. Dr. Marcone Jamilson Freitas Souza
Doutor em Computação pela
UFRJ - Universidade Federal do Rio de Janeiro
Examinador
Departamento de Computação – UFOP

_____

Prof. Dr. José Romildo Malaquias
Doutor em Ciências pela
UFU - Universidade Federal de Uberlândia
Examinador
Departamento de Computação – UFOP

Resumo

O presente trabalho objetiva dar continuidade ao desenvolvimento de um ambiente de modelagem e execução de simulações tridimensionais interativas em tempo-real e georeferenciadas, iniciado no semestre anterior (trabalho disponível em [1]).

Abstract

The aim of this work is to continue the development of a modeling and simulation environment for real-time interactive and georeferenced 3D simulations, started at the previous semester (work available at [1]).

Agradecimentos

Agradeço primeiramente a Deus.

Agradeço à minha mãe e amiga Aurora, que sempre esteve do meu lado, e de quem tenho muito orgulho, e ao meu candidato a padrasto Cor Jesus.

Aos familiares em Curvelo, em especial a vovó Ilda, tio Paulo, tia Maria Helena, o pequeno grande Henrique, e aos familiares de consideração tia Marina e tia Lourdes, que mesmo longe sei que posso contar com eles.

Aos amigos de verdade, espalhados pelo mundo. São poucos, mas significam muito. Amigos de jogar videogame nos fins de semana, a galera dos shows de Heavy Metal, os doentes da Unidev, os irmãos da cabin 21 em Cherokee.

Agradeço ao professor Tiago Carneiro pela grande oportunidade dada, e pelos ensinamentos compartilhados. Aos colegas de curso, pelo companheirismo e disposição em ajudar, e à UFOP, onde amadureci nestes últimos anos.

E por último, mas não menos importante, agradeço à minha namorada Alexandra pelo amor, apoio e compreensão, e aos Ruccini de Poá.

Índice

Lista de Figuras

Lista de Exemplos

1. Introdução

O ambiente *TerraME* (*Terra Modelling Environment*) [2][3] permite o desenvolvimento e simulação de modelos dinâmicos georeferenciados por meio de uma linguagem de programação de alto-nível, também chamada TerraME, que fornece meios para a descrição formal, rápida e clara desses modelos. No entanto, a análise exploratória dos resultados das simulações pode ser trazida para um patamar onde as sensações intuitivas utilizadas pelos seres humanos na decodificação de cenas do mundo real podem ser utilizadas para o entendimento dos comportamentos resultantes. A visualização tridimensional e interativa desses resultados, visualmente fiel aos processos modelados, facilita esse aprendizado e constitui o primeiro passo para se atingir tal objetivo.

Com o avanço da tecnologia da computação gráfica e com o aumento da capacidade de processamento e armazenamento computacional é possível recriar ambientes e situações com um grau de fidelidade cada vez maior em relação ao mundo real. Estes ambientes podem ser interativos, permitindo que o usuário não só observe uma cena a partir de diferentes pontos, mas também que exerça influência sobre a simulação observada.

Este trabalho objetiva o desenvolvimento de um ambiente de modelagem e execução, destinado à representação em três dimensões de modelos dinâmicos georeferenciados modelados no ambiente *TerraME*, que permita ao usuário interagir, em tempo-real, com o modelo durante sua simulação, facilitando tanto o entendimento do modelo quanto o diagnóstico de problemas.

Algumas aplicações atualmente disponíveis no mercado que já implementam funcionalidades semelhantes ao que é proposto neste trabalho são o *Walkinside* [4], da empresa *VRContext*, o *Quest3D* [5], da *Act-3D* e o *VRWorx* [6] da *VRToolbox*.

O *Walkinside* permite a importação de modelos 3D de aplicativos *CAD* e permite fazer um "*passeio virtual*" pelo ambiente importado, mas não oferece possibilidades de customização das regras de comportamento e da interatividade. O *Quest3D* possui uma série de componentes prontos, que dá bastante flexibilidade ao usuário, e que permite criar praticamente qualquer tipo de cena, onde o usuário pode descrever graficamente uma simulação na forma de um grafo, cujos nós (chamados de canais) representam entidades, como câmeras, objetos tridimensionais e/ou funções. No entanto, para criar novas funcionalidades (novos canais) além dos oferecidos pelo pacote, é preciso conhecimento avançado em programação e para a criação de *DLL's (Dymanic Link Library,* da plataforma *Microsoft Windows)*. O *VRWorx* não trabalha com ambientes 3D de fato, mas permite criar visões panorâmicas através de concatenação de fotos.

Neste semestre, foi dada continuidade ao desenvolvimento da aplicação *VRScript*, iniciado no semestre anterior. Esta segunda parte do trabalho visa a implementação dos módulos: GUI (interface gráfica com o usuário) e áudio (manipulação de músicas e efeitos sonoros), o tratamento de joysticks, a criação de um sistema de rotas e a adição de funcionalidades no módulo de física e tratamento de colisões.

2. Revisão da Literatura

O *VRScript* é um framework em linguagem C++, sobre o qual aplicações tridimensionais interativas podem ser desenvolvidas utilizando a linguagem de script *LUA*. Foram integradas ao framework diversas bibliotecas, visando oferecer funcionalidades ao usuário do *VRScript*.

A linguagem *LUA* é o principal componente do sistema, pois é através dela que o usuário (modelador da simulação) irá descrever sua aplicação. Para que o framework em C++ possa se comunicar com o código *LUA*, é utilizada a *API C* de *LUA*, que cria uma máquina virtual onde o código *LUA* é executado.

O motor gráfico *OGRE 3D* é responsável pelo carregamento, manipulação e visualização do mundo 3D e dos objetos que compõem a simulação, como objetos 3D, terrenos, câmeras, luzes, imagens.

A biblioteca *OIS* faz o reconhecimento e tratamento de eventos de teclado, mouse e joystick, o que permite dar interatividade às simulações desenvolvidas.

O framework *Qt*, é responsável por criar a janela da aplicação, e permite adicionar componentes de interface gráfica, como botões, caixas de diálogos e menus.

A biblioteca *Newton Game Dynamics* atua na simulação física e tratamento de colisão. Ela executa cálculos baseados em massa, posição, velocidades, forças aplicadas ao corpos, o que permite que os objetos no mundo 3D possuam comportamentos mais próximos do mundo real.

Por fim, a *API OpenAL* faz o tratamento de áudio. Ela permite carregar arquivos de som e executá-los durante uma simulação.

Todas estas ferramentas, integradas no framework VRScript, permitem oferecer ao usuário do framework diversas funcionalidades que podem ser utilizadas na modelagem de uma simulação.

2.1. A Linguagem Lua

O *TerraME* provê uma linguagem de programação de alto nível para a descrição do modelo. A linguagem *TerraME* é uma extensão da linguagem *Lua* (Ierusalimschy, Figueiredo et al. 1996) [7][8] que, por sua vez, é uma linguagem extensível que possui uma sintaxe simples e ambiente de execução eficiente. *Lua* possui uma grande quantidade de programadores na indústria de desenvolvimento de jogos, uma atividade que possui muitos requisitos em comum com a de modelagem e simulação de processos.

Lua foi criada com o objetivo de resolver problemas com poucas linhas de código. Para isso, ela se baseia em extensibilidade. Ao contrário de outras linguagens, *Lua* é facilmente estendida não apenas com softwares escritos na própria linguagem *Lua*, mas também escritos em outras linguagens, como *C* e *C++*. A extensibilidade de *Lua* é tão notável que muitos a

consideram não como uma linguagem, mas um kit para construção de linguagens específicas aos seus domínios.

Lua é uma linguagem dinamicamente tipada. Qualquer variável pode conter um valor de qualquer tipo (exemplo 2.1.1).

```
a = 10            -- a é do tipo número (number)
a = "uma string"  -- a agora é do tipo string
a = print         -- a é do tipo função (function)
```

Exemplo 2.1.1 - Lua é uma linguagem dinamicamente tipada

Existem oito tipos básicos em Lua: *nil*, *boolean*, *number*, *string*, *userdata*, *function*, *thread* e *table*.

Vale destacar o tipo tabela (*table)*, que implementa vetores associativos. Um vetor associativo é um vetor que pode ser indexado não apenas por números, mas também por *strings* ou qualquer valor da linguagem, exceto *nil*. Além disso, tabelas não possuem tamanho fixo, elas podem crescer ou diminuir dinamicamente. Tabelas são a principal (de fato a única) estrutura de dados em *Lua*, mas uma estrutura poderosa. Tabelas são usadas para criar vetores comuns, tabelas de símbolos, conjuntos, registros, listas e outras estruturas, de forma simples e uniforme (exemplo 2.1.2).

```
a = {}            -- declaração de tabela
k = "x"
a[k] = 10         -- nova entrada, índice "x", valor 10
a[20] = "ótimo"   -- nova entrada, índice 20, valor "ótimo"
a.cor = "azul"    -- nova entrada, utilizando a notação de ponto,
                  --    índice "cor", valor "azul"
print(a["cor"])   -- imprime "azul"
```

Exemplo 2.1.2 - Tabelas em Lua

Lua foi desenvolvida, desde o princípio, para ser integrada a softwares escritos em *C* e outras linguagens convencionais. A *API C* de *Lua* é um conjunto de funções que permitem código em *C* interagir com *Lua*. Ela é composta de funções para ler e escrever variáveis globais de *Lua*, chamar funções de *Lua*, executar trechos de código *Lua*, registrar funções em *C* para que possam ser posteriormente chamadas a partir de código *Lua*, etc.

O principal componente na comunicação entre *Lua* e *C* é a onipresente pilha de execução virtual. Quase todas as chamadas da *API* operam sobre esta pilha. Todas as trocas de dados de *Lua* para *C* e vice-versa ocorrem através desta pilha.

2.2. O Motor Gráfico Ogre 3D

Uma *game engine* (motor de jogo) é o componente de *software* central de um *video game* (jogo eletrônico) ou outra aplicação gráfica interativa em tempo-real. Ela provê as funcionalidades geralmente necessárias a esse tipo de aplicação, e simplifica o desenvolvimento. As funcionalidades tipicamente fornecidas por uma *game engine* incluem um módulo de renderização (transformação de informações geométricas, de textura e iluminação, em imagens projetadas na tela) para gráficos bi ou tridimensionais,

gerenciamento de cena, simulação física, detecção de colisão, áudio, animação, inteligência artificial, rede, reconhecimento de dispositivos de entrada (como joysticks). [9][10]

Muito freqüentemente, as *engines* são construídas sobre uma *API* gráfica como *OpenGL* ou *Direct3D*, provendo um nível acima das funções básicas de manipulação de primitivas dessas *APIs*.

Atualmente existe uma enorme quantidade de *engines* disponíveis (o banco de dados de engines do site *DevMaster.net* [11] conta atualmente com 284 *engines* catalogadas), variando de *engines* gratuitas e *open-source*, a algumas que cobram centenas de milhares de dólares para a sua utilização.

Dentre as *engines open-source* as mais populares são: *OGRE 3D* [12][13], *Irrlicht* [14] e *Crystal Space* [15]. Para este projeto escolhemos utilizar a *OGRE 3D* (*Object-Oriented Graphics Rendering Engine* – Motor para Renderização Gráfica Orientado a Objetos) pelos seguintes motivos:

- Gratuita, *open-source*, sob a licença *LGPL* que permite a sua utilização sem nenhuma restrição, mesmo em projetos comerciais.

- Multi-plataforma, trabalhando tanto com a *API OpenGL* como *Direct3D*, permitindo sua utilização em *Windows*, *Linux* e *Mac OSX*.

- *Engine* madura, com mais de 5 anos de desenvolvimento, comprovadamente estável e usada em um grande número de produtos comerciais entre jogos, visualização arquitetural e simuladores.

- Possui uma excelente documentação que descreve toda a sua arquitetura e classes, grande quantidade de tutoriais, e um livro publicado (*Pro OGRE 3D Programming*) [13], além de uma comunidade ativa e crescente.

- Possui uma interface orientada a objetos "limpa", bem organizada e fácil de usar, com uma arquitetura de *plugins* que permite que suas funcionalidades sejam estendidas sem a recompilação do código-fonte da *engine*.

A desvantagem entre a *OGRE 3D* e outras *engines* é que a *OGRE* é apenas uma *engine de renderização*, ou seja, cuida apenas de organizar os elementos da cena de forma a otimizar o processo de desenhar na tela (renderizar) a cena, atualizando a cada quadro de animação. Por ser apenas uma *engine* de renderização a *OGRE 3D* não possui outras funcionalidades essenciais ao desenvolvimento de simulações interativas, como tratamento de dispositivos de entrada (teclado, mouse, joystick), manipulação de áudio (músicas, efeitos, som ambiente), tratamento de colisão, simulação de física e funções de rede (para a criação de jogos ou simulações multi-usuário através de rede local ou Internet). Entretanto a *engine* permite e facilita a integração de outras bibliotecas para fornecerem essas funcionalidades que a *engine* não oferece.

2.3. A Biblioteca OIS

O motor gráfico faz o trabalho de organizar e desenhar a cena virtual, mas sem interação restaria ao usuário apenas assistir ao que é desenhado na tela. Para inserir o elemento interatividade a uma simulação é preciso reconhecer dispositivos de entrada de dados e reagir a seus comandos.

Teclado e mouse são os dispositivos mais comuns, acessíveis a praticamente qualquer usuário e normalmente utilizados em qualquer aplicação, portanto o reconhecimento destes dispositivos é obrigatório para uma aplicação interativa.

No mundo da realidade virtual, e principalmente em jogos, outro dispositivo bastante utilizado é o joystick (ou controlador de jogo).

Ainda para utilização em realidade virtual existem outros dispositivos especiais, mas que têm sua utilização ainda reduzida, principalmente devido ao alto custo. Neste grupo entram luvas que reconhecem movimentos da mão e dos dedos, detectores de movimentos da cabeça e visores que permitem maior imersão na cena.

Para tratamento de eventos de teclado, mouse e joystick um uma aplicação o programador pode utilizar de funções das bibliotecas fornecidas pelo próprio sistema operacional, como a *API* do *Windows* ou a *API DirectX*, voltada para o desenvolvimento multimídia.

Existe ainda a possibilidade de se usar bibliotecas específicas, que são construídas sobre a *API* do sistema operacional, mas que provê funcionalidades extras. Um exemplo é a biblioteca *OIS* (*Object-Oriented Input System* – Sistema de Entrada Orientado a Objetos) [16]. Suas principais características são:

- É gratuita, *open-source* e sua licença permite sua utilização livre em qualquer projeto.
- Multi-plataforma: Permite a utilização em sistemas Windows e Linux.
- Reconhece teclados, mouses e joysticks.

2.4. O Framework Qt

O framework Qt [17], da empresa Trolltech, é um framework de código-fonte aberto, composto por uma ampla biblioteca de classes e um conjunto de ferramentas voltadas para o desenvolvimento de aplicações multi-plataforma.

Na biblioteca de classes do Qt destacam-se os componentes de interface gráfica (*GUI - Graphical User Interface*, ou Interface Gráfica com o Usuário), que inclui desde botões, menus, caixas de diálogos, campos de texto, até tabelas, árvores, componentes de desenho, etc. O Qt permite ainda a extensão dos componentes padrões, permitindo personalizar seu visual e comportamento.

As funcionalidades do Qt vão além dos componentes *GUI*. A biblioteca fornece classes para manipulação de arquivos, rede, threads, acesso a banco de dados, processamento de arquivos XML, integração com *OpenGL*, etc.

Um dos maiores diferenciais da biblioteca de classes do Qt é o seu sistema de sinais e *slots* para comunicação entre componentes. Um sinal é emitido quando um evento em particular ocorre. Os componentes do Qt possuem vários sinais predefinidos, mas que podem ser estendidos e novos sinais adicionados. Um *slot* é uma função chamada em resposta a determinado sinal. Assim como sinais, novos *slots* podem ser criados. Por exemplo, o sinal *valueChanged* (valor alterado) em um componente *slider* pode ser conectado ao *slot setValue* (definir valor) de uma caixa de texto, assim sempre que o usuário arrastar o *slider*, ele automaticamente altera o valor na caixa de texto.

Além da biblioteca de classes, o Qt fornece um conjunto de ferramentas, onde se destacam o *Qt Designer*, um ambiente gráfico para modelagem de interfaces e definição de sinais e slots, e o *qmake*, para a criação de arquivos *makefile*.

2.5. A Biblioteca Newton

Outra funcionalidade imprescindível em simulações interativas é o tratamento de colisão. Uma cena normalmente possui objetos estáticos (um terreno, casas, prédios, árvores) e objetos dinâmicos (carros, pessoas). Estes objetos dinâmicos se movem, seja pela interação do usuário ou por outros fatores definidos na aplicação, e fatalmente entram em contato uns com os outros e com objetos estáticos. Como, na maioria das vezes, estes objetos representam objetos do mundo real, espera-se que eles não apenas se pareçam visualmente com a realidade, mas se comportem como tal. Sendo assim, é desejável, pelo menos, que objetos não "atravessem" outros objetos, respeitando a forma de cada um.

Ainda mais interessante que o fato de objetos não atravessarem uns aos outros, passando a impressão de que são sólidos, é o fato de que objetos quando colidem uns com outros reajam realisticamente, levando em consideração suas massas, velocidade no momento da colisão, forma, ângulo de colisão, atrito, elasticidade e outras variáveis que controlam o comportamento dos corpos no mundo real.

O primeiro requisito (objetos não atravessarem uns aos outros) pode ser atendido através de um sistema de detecção de colisão. Tal sistema irá realizar constantes cálculos para

determinar quais objetos estão colidindo com quais, em quais pontos, permitindo que o programa reaja a essa colisão.

O segundo requisito (objetos terem reações fisicamente realistas) já é um pouco mais complicado de ser atendido, uma vez que envolve cálculos avançados levando-se em conta todos os fatores que de uma forma ou outra interferem nos movimentos dos corpos em um sistema dinâmico. Além disso, uma simulação em tempo-real requer que tais cálculos sejam feitos no menor tempo possível. Uma simulação que rode a 30 quadros por segundo (que é um valor típico), por exemplo, dispõe de 33 milésimos de segundo para fazer todos os cálculos (não só de física, mas de toda a lógica da simulação), atualizar e renderizar a cena. Por esse motivo, simulações em tempo-real tendem a utilizar aproximações e simplificar os cálculos de física. O comportamento precisa parecer real, mas não precisa envolver todas as variáveis existentes no mundo real. Há uma troca de precisão por desempenho.

Existem algumas bibliotecas criadas exatamente com esse fim. Dentre elas podemos destacar a *Newton Game Dynamics* [18]. A *Newton* é uma biblioteca gratuita, de uso livre e multi-plataforma (Windows e Linux), voltada especialmente para jogos ou qualquer simulação em tempo-real.

2.6. A API OpenAL

A *OpenAL* (*Open Audio Layer*, ou Camada de Áudio Aberta) [19] é uma API multi-plataforma para manipulação de áudio 3D voltada para o uso em jogos (embora possa ser utilizada em outros tipos de aplicações que necessitem manipular sons). A *OpenAL* se assemelha a *OpenGL* em sua nomenclatura de funções e definições de tipos. Assim como *OpenGL*, a *OpenAL* é uma especificação mantida por um grupo de pessoas e empresas, onde atualmente se destaca a fabricante de hardware de áudio Creative Labs. O SDK da *OpenAL* tem código-fonte aberto, licença livre e pode ser usado nas plataformas Windows, Linux, Mac e XBox.

A API modela um conjunto de fontes geradoras de sons se movendo em um espaço tridimensional, e um ouvinte também posicionado no espaço 3D. A *OpenAL* calcula a intensidade, freqüência e posicionamento do som que chega ao ouvinte, de acordo com a distância entre ele e a fonte geradora de som, e também de acordo com as velocidades com que eles se movem. No mundo real, a velocidade relativa entre a fonte e o ouvinte causa uma distorção na freqüência percebida do som. Se a fonte está aproximando do ouvinte, o som gerado é percebido com uma maior freqüência que o som original, ou se a fonte está se afastando, o som é percebido com uma freqüência maior. Esta alteração na percepção do som com a velocidade é conhecida como efeito *Doppler* [20], e este efeito é simulado pela API *OpenAL*.

Os objetos básicos presentes na *OpenAL* são: `Listener` (ouvinte), `Source` (fonte geradora de som) e `Buffer` (informação de áudio). Podem existir vários *buffers*, que contém a informação do áudio (geralmente carregada a partir de um arquivo WAV ou outro formato de áudio). Cada *buffer* é ligado a uma ou mais fontes, que representam pontos no espaço 3D que emitem som. Existe sempre um ouvinte, que representa a posição de onde os sons serão captados.

3. Metodologia

3.1. Integrando as Ferramentas

Uma característica deste trabalho é o requerimento por um conjunto de diferentes funcionalidades, como manipulação de objetos 3D, desenho na tela, tratamento de dispositivos de entrada, simulações físicas, etc. Ao invés de desenvolver todas as funcionalidades do zero, decidimos por procurar ferramentas existentes que executem estas funções. A vantagem desta abordagem, é que podemos utilizar funcionalidades que já foram implementadas, testadas e otimizadas, e podemos trocar experiências com comunidades que já utilizam estas ferramentas. Ao optarmos por ferramentas livres, de código aberto, temos um maior controle do código que estamos adicionando ao nosso sistema, e temos a possibilidade de modificá-lo, caso haja necessidade.

Uma desvantagem em utilizar bibliotecas de terceiros é que nem sempre estas bibliotecas foram projetadas para funcionar em conjunto umas com as outras. Isto requer um trabalho de integração entre as diferentes ferramentas.

O aplicativo desenvolvido tem suas funcionalidades divididas em módulos, que são:

- Renderização: responsável pelo carregamento, manipulação e exibição das entidades 3D e cenários;
- Máquina Virtual: inicializa a máquina virtual, onde será executado o código Lua, e de onde é feita a comunicação entre Lua e C++;
- Input: reconhece e manipula os dispositivos de entrada (teclado, mouse e joysticks);
- Física: controla a simulação física e detecção de colisões;
- GUI: cria e controla janelas e elementos de interface gráfica;
- Áudio: controla o áudio da aplicação.

Cada módulo pode fazer uso de uma ou mais bibliotecas de terceiros, criando uma interface comum para que os módulos possam funcionar em conjunto.

A separação em módulos permite que uma biblioteca possa ser substituída ou modificada, sem que seja necessário modificar os outros módulos. Bastar manter a mesma interface e a integração entre os módulos se mantém.

3.2 Exportando Classes e Objetos C++ para a Linguagem Lua

Um dos objetivos deste trabalho é simplificar ao máximo a definição de uma cena. Assim como a linguagem Lua é um linguagem de script de sintaxe simples e execução eficiente, nossa idéia é utilizá-la para a descrição de uma cena e os comportamentos nela presentes. Desta maneira, evita-se que o desenvolvedor de um jogo ou simulação tenha que descrever sua aplicação em C++. Para isso, a linguagem Lua foi estendida pela inserção de objetos e serviços oferecidos pelos diferentes módulos do aplicativo VRScript. Assim, como o código Lua é interpretado, a aplicação não precisa ser recompilada a cada modificação. Isto permite que o código seja alterado e o resultado seja visualizado instantaneamente, fato que torna eficiente a prototipação de soluções a novos requisitos.

Através da biblioteca *ToLua++* [21] exportamos para a linguagem Lua classes e métodos escritos em C++. O código em Lua é interpretado pela máquina virtual Lua, que faz as chamadas às funções correspondentes em C++. A biblioteca *ToLua++* é responsável por inserir e ler valores da pilha de execução conforme necessário.

4. Trabalho Anterior

Este trabalho é a continuação do que foi iniciado no primeiro semestre de 2007, na disciplina Projeto Orientado, texto que está disponível em [1]. Naquela primeira parte foi feita a revisão da literatura e desenvolvimento de alguns módulos:

- O módulo de renderização já fazia a manipulação de entidades (carregar objetos, posicioná-los, transformá-los, animá-los), câmeras, luzes, terrenos, overlays (camada "sobre" a tela onde podem ser escritos textos e desenhadas figuras e gráficos).
- O módulo de input, que tratava de mouse e teclado (mas ainda não tratava joysticks).
- A máquina virtual.
- O módulo de física foi iniciado, mas ainda não era utilizável.

A figura 4.1 mostra o diagrama das principais classes e módulos da versão desenvolvida até então. Ela mostra a classe central (`Scene`) os módulos desenvolvidos.

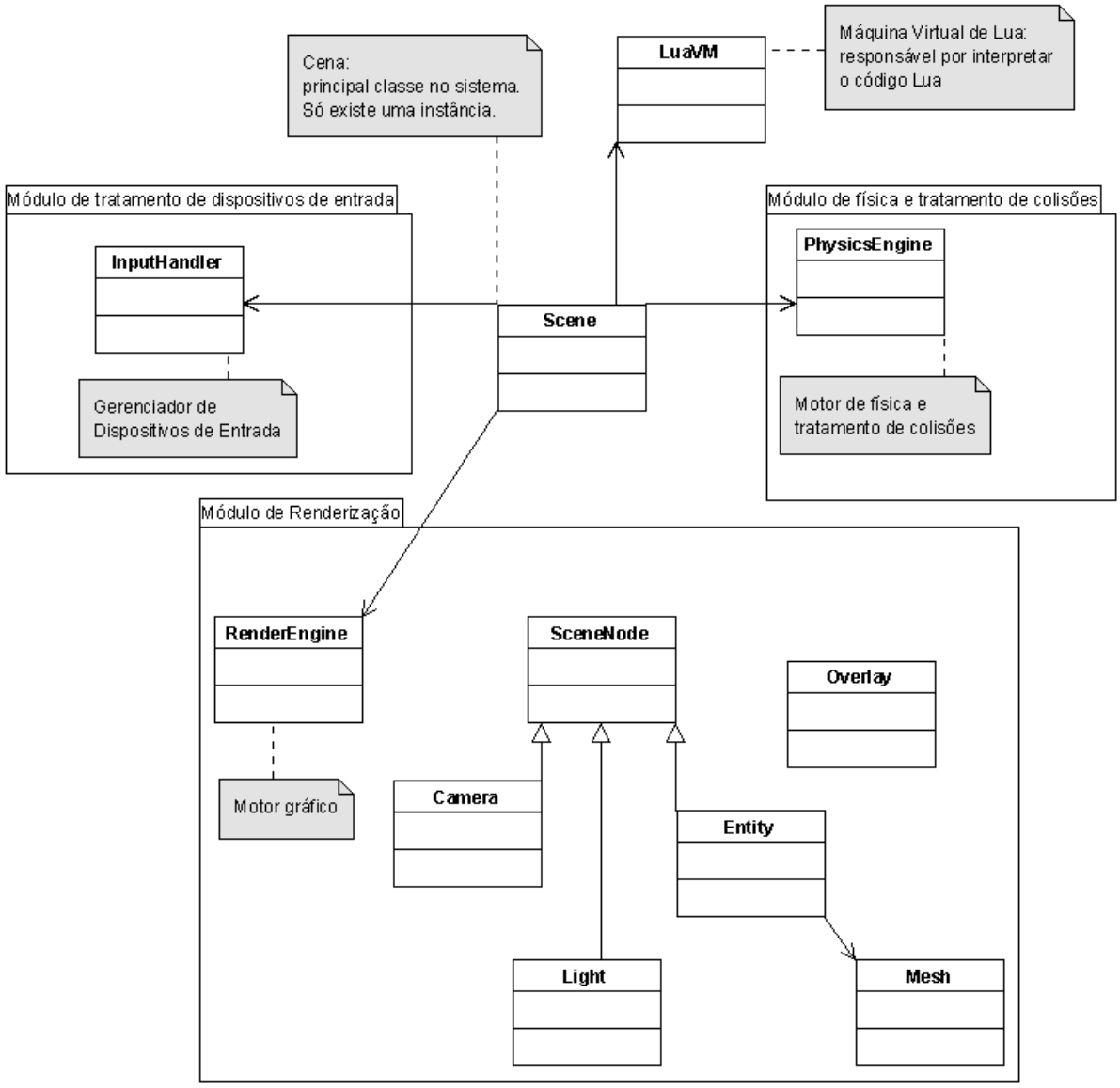

Figura 4.1: Diagrama de classes do aplicativo desenvolvido no trabalho anterior

O exemplo 4.1 mostra o código Lua de uma simulação desenvolvida, e a figura 4.2 mostra a imagem resultante.

```
------------------------------------------------------------
-- INICIALIZAÇÃO
------------------------------------------------------------
-- cria uma janela de dimensões 640x480
createWindow(640,480,false,false,DIRECT3D)

-- cria uma câmera e define sua posição
cam = Camera("maincam")
cam:setPosition(0,100,0)

-- carrega o modelo de um terreno
terrain = Entity("terrain", "terrain.mesh")

------------------------------------------------------------
-- LAÇO PRINCIPAL
------------------------------------------------------------
function updateWorld()
      -- verifica se as teclas W ou S estão pressionadas,
      -- movendo a câmera para frente (W) ou para trás (S)
      if keyDown(KEY_W) then
            cam:translate(0,0,-10)
      elseif keyDown(KEY_S) then
            cam:translate(0,0,10)
      end

      -- o laço principal retorna true para indicar
      -- que a seimulação deve prosseguir
      return true
end
```

Exemplo 4.1 - Código de uma simulação

Figura 4.2: Aplicativo em execução

5. Descrição do Trabalho

5.1. Integração com o Framework Qt

Com o intuito de oferecer uma maior interatividade para o usuário de aplicativos VRScript, é interessante dar ao desenvolvedor de uma aplicação VRScript a possibilidade de inserir elementos de interface gráfica com o usuário (*GUI*) em suas simulações. Desta maneira, o usuário desta aplicação poderia, através de caixas de diálogo, menus e outros controles, fornecer ou receber informações sobre objetos presentes na cena tridimensional.

Para alcançar esse objetivo, decidimos por utilizar a biblioteca GUI do Qt, e integrá-la ao nosso aplicativo. O primeiro passo foi passar a janela de renderização, que é criada pela Ogre 3D, para o controle do Qt. Fizemos isso através de uma opção da Ogre 3D na hora de criação da janela de renderização, que permite fornecer uma referência para outra janela como "pai" da janela de renderização. Criamos uma classe chamada `OgreView`, que é uma subclasse de `QWidget`. `QWidget` faz parte da biblioteca Qt e é a classe base para todos os componentes GUI. A janela de renderização da Ogre 3D fica contida no componente `OgreView`, que posteriormente é adicionado a uma janela principal (criada pela Qt).

Mas por que não fazer a janela de renderização contida diretamente na janela principal criada pelo Qt, ao invés de usar o componente OgreView? Porque desta forma, com a janela de renderização sendo um componente da janela principal, é possível redimensionar a área de renderização e posicionar outros componentes na janela principal, por exemplo, podemos criar um painel com controles em uma coluna do lado direito da janela principal, com a área de renderização ocupando o restante da janela.

O próximo passo foi exportar para a linguagem Lua funções quer permitiam a manipulação de componentes GUI. De início exportamos apenas algumas funções para teste, como uma função para exibir uma caixa de mensagem, mostrar uma tela de carregamento (*splash screen*), e exibir controles na tela do tipo executar/pausar e um *slider* para alterar a velocidade de execução da simulação. A figura 5.1.1 mostra um exemplo de tela de uma simulação com componentes GUI.

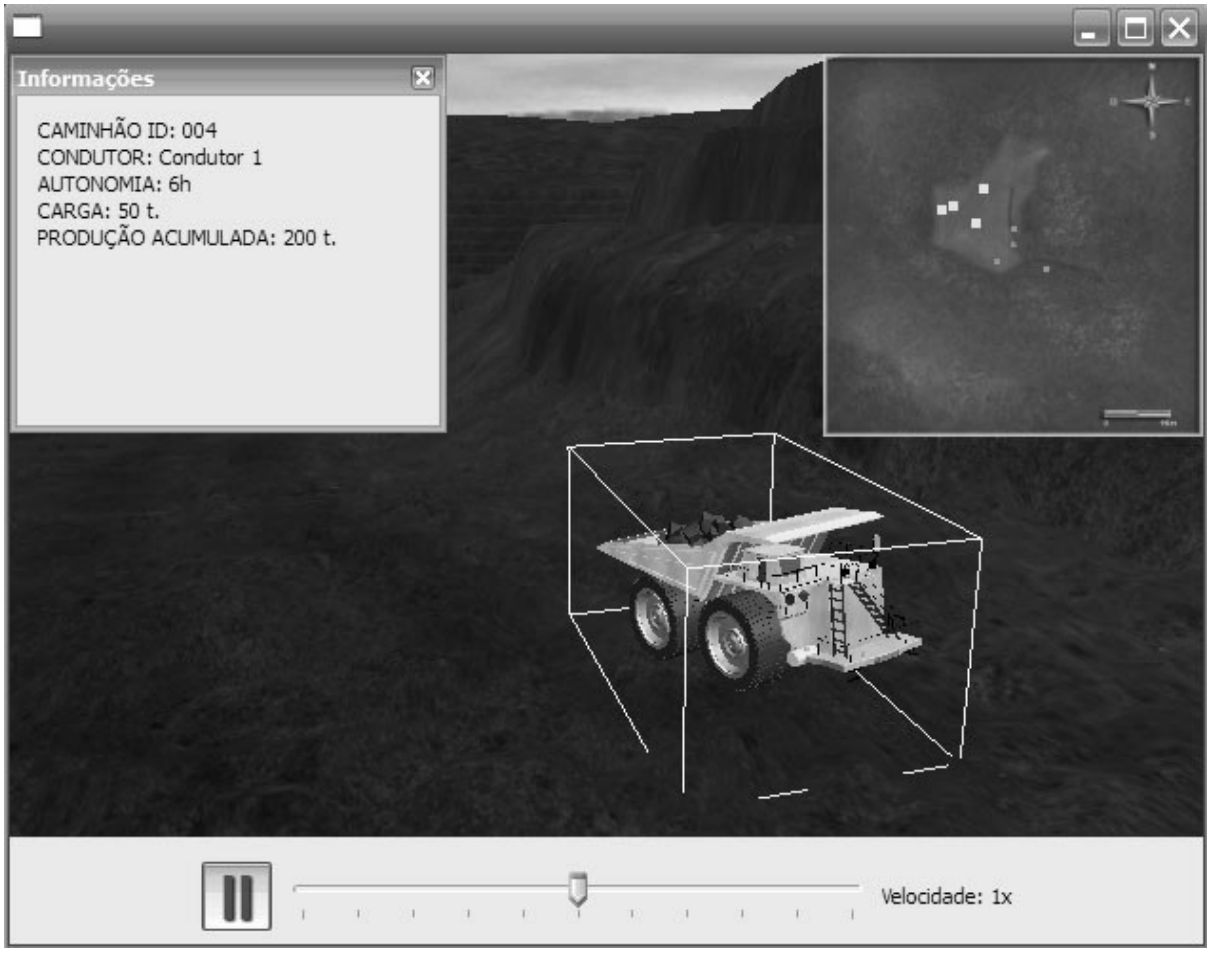

Figura 5.1.1 - Tela de uma simulação com elementos GUI

5.2. Reconhecimento de Joysticks

O aplicativo desenvolvido até o momento dava suporte para teclado e mouse, mas ainda faltava suporte a joysticks. Para isso, a classe `Joystick` foi desenvolvida utilizando-se a biblioteca OIS.

O funcionamento para joysticks é semelhante ao de teclado e mouse. Criamos classes *listeners*, que recebem da OIS informações sobre eventos nos dispositivos de entrada assim que eles ocorrem (quando um botão é pressionado, quando e liberado, movimentos de mouse, alavancas de joysticks, etc.). Guardamos estas informações e fornecemos à aplicação métodos para que possam ser verificados os estados dos dispositivos, por exemplo, o estado de um botão, a posição do cursor do mouse, de uma alavanca em um joystick.

A implementação do tratamento de joysticks apresentou algumas dificuldades, primeiramente pelo fato de diferentes joysticks possuírem diferentes mecanismos que precisam ser tratados. Alguns joysticks, por exemplo, possuem um ou mais direcionais analógicos, alguns possuem botões deslizantes, e cada modelo de joystick possui diferentes quantidades de botões. Outra dificuldade foi dar suporte a mais de um joystick ativo ao mesmo tempo. Ao passo que, normalmente se tem apenas um teclado e um mouse ativos em determinado momento, pode haver mais de um joystick.

Para resolver este problema, exportamos para Lua a classe `Joystick`, e exportamos funções para pegar referências aos joysticks instalados através de seus nomes ou números (exemplo 5.2.1).

```
-- pega referência ao primeiro joystick instalado
meuJoystick = getJoystick(1)

-- verifica se o botão 1 está pressionado
if (meuJoystick:joyDown(1)) then
    print("botão 1 pressionado")
end
```

Exemplo 5.2.1 - Utilização de joysticks

Fizemos testes com sucesso utilizando um joystick do tipo "volante" e outro do tipo "manche", ambos sendo usados simultaneamente.

5.3. Sistema de Rotas

Outra funcionalidade que julgamos interessante disponibilizar ao desenvolvedor de aplicativos VRSctript era um sistema de rotas, onde ele poderia definir rotas por meio de um conjunto de pontos, e fazer com que objetos da cena percorressem estas rotas. Por exemplo, fazer veículos percorrerem uma auto-estrada. O desenvolvedor poderia carregar um conjunto de pontos exportado de uma ferramenta de modelagem 3D, ou ferramenta CAD (Computer Aidded Design), ou ferramenta GIS (Geographic Information System) e definir objetos para percorrerem estas seqüências de pontos.

Criamos uma nova classe, a `MobileEntity`, que é uma `Entity` (entidade) mas com características extras como funções para receber pontos e percorrer uma rota.

Em uma primeira tentativa de resolver o problema, criamos um algoritmo onde uma `MobileEntity` recebia os pontos para formar uma rota. Estes pontos ficavam em uma pilha, e quando mandávamos a entidade executar o movimento ela ia consumindo estes pontos e percorrendo-os, um a um. Quando a entidade chegava a um ponto, ela se direcionava para o ponto seguinte e movimentava-se em sua direção.

Esta primeira solução, no entanto, mostrou algumas deficiências. A rota formada pelos pontos não era uma linha suave, mas sim com quinas entre um segmento e outro. Quando a entidade percorria essas rotas, ocorria uma mudança brusca de direção quando ela chegava a cada ponto. Como conseqüência também do fato das rotas serem formadas por linhas retas, a mudança de orientação do objeto era brusca. A um dado instante o objeto está alinhado ao segmento que ele está percorrendo, e no instante seguinte ele está alinhado ao próximo segmento. Uma solução para suavizar a rota seria adicionar pontos cada vez mais próximos uns dos outros, mas isso iria aumentar consideravelmente o consumo de processamento e memória, além de dificultar para o usuário a criação e manutenção destas rotas. A figura 5.3.1 ilustra o problema.

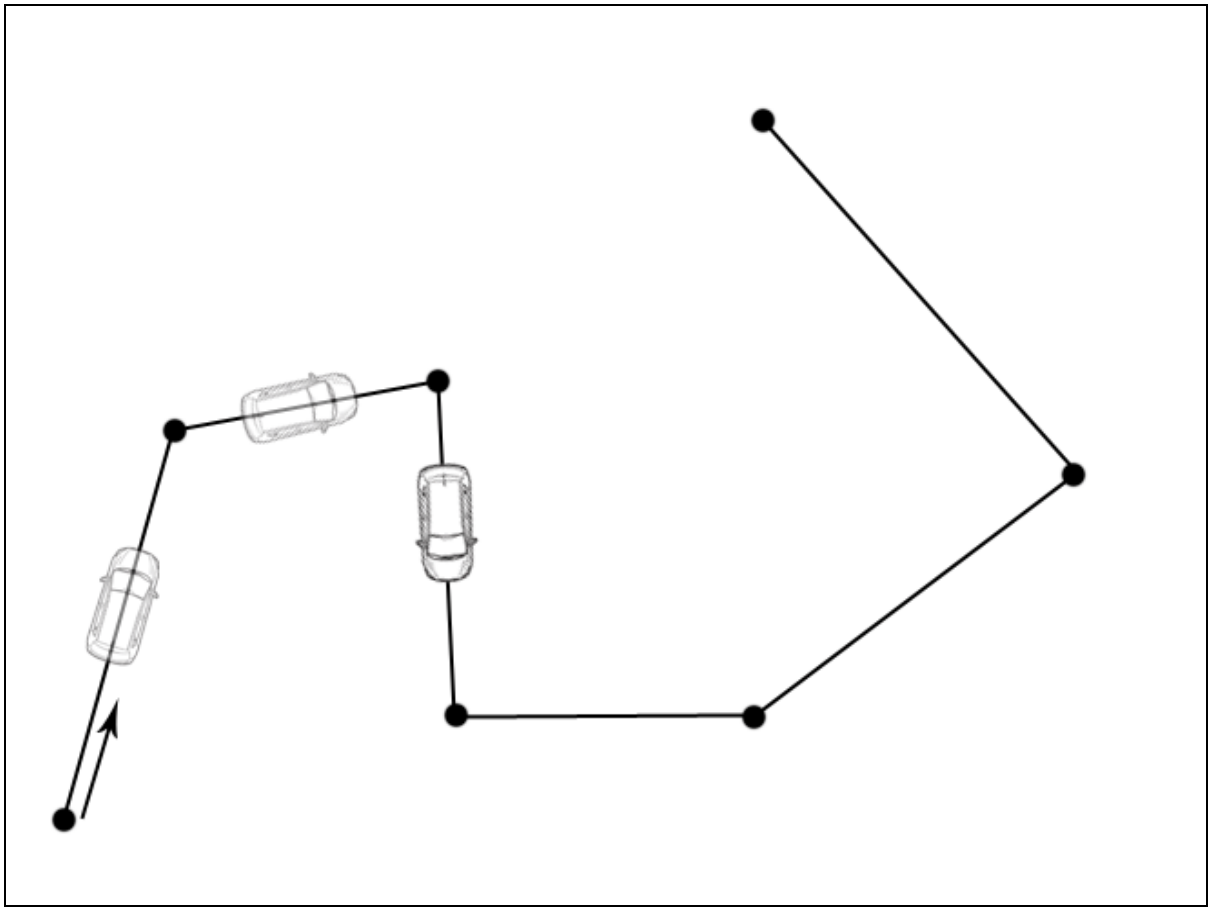

Figura 5.3.1 - Primeira tentativa de solucionar o problema de rotas

Passamos então a estudar como resolver estas deficiências. Pesquisamos sobre *splines*, que são linhas curvas criadas a partir de um conjunto de pontos. Felizmente o motor Ogre3D possui algumas classes para se trabalhar com *splines*, inclusive a classe `SimpleSpline`, que cria uma spline dado um conjuntos de pontos. A figura 5.3.2 mostra o mesmo conjunto de pontos da figura 5.3.1, mas agora com a rota criada utilizando splines.

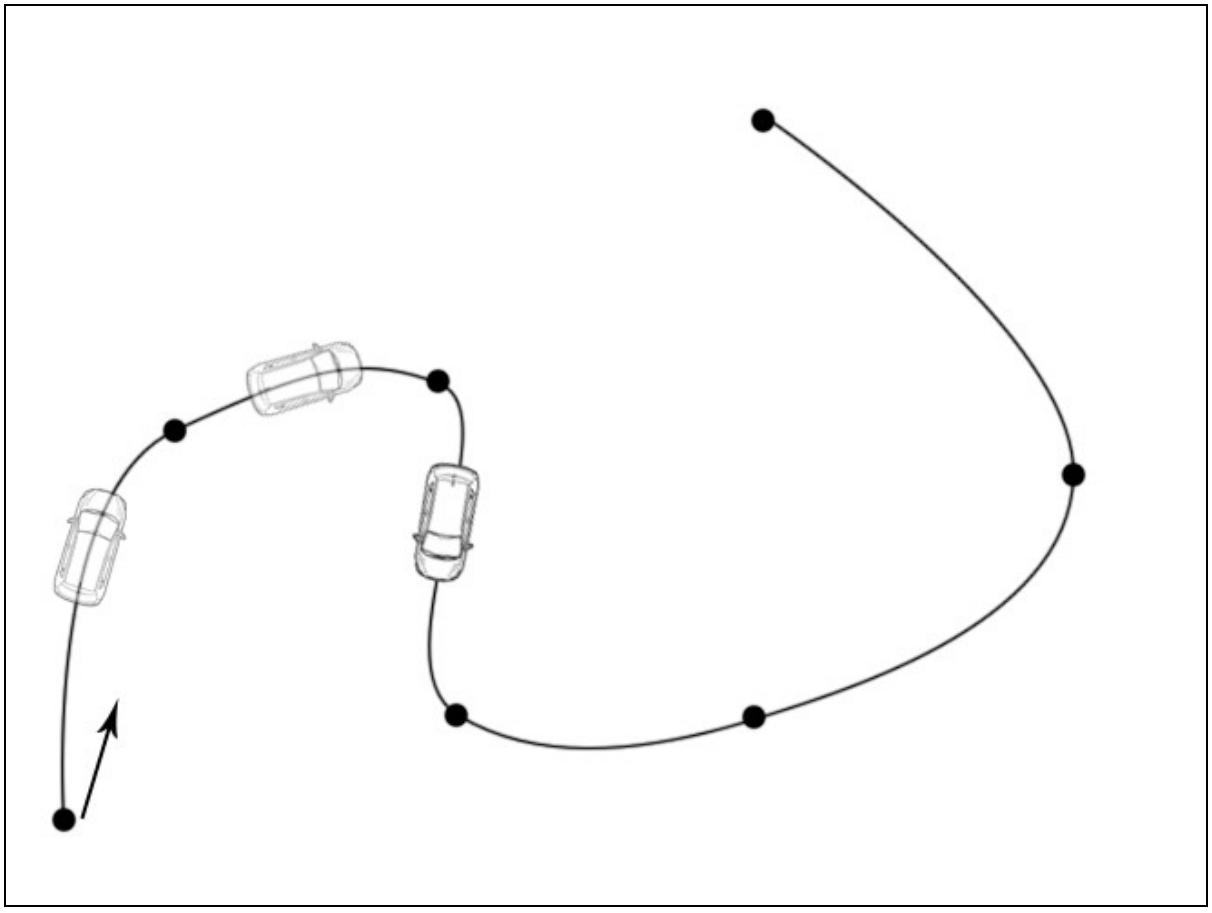

Figura 5.3.2 - Nova rota criada utilizando splines

A partir daí o nosso sistema de rotas foi reescrito. Além de adicionar o conceito de *splines*, mudamos a estrutura do sistema a fim de torná-lo mais flexível. Criamos a classe `Route`, que representa uma rota. É a classe `Route` que recebe os pontos, e internamente cria uma *spline*. Uma rota pode ser percorrida por mais de uma entidade, e uma entidade pode mudar de rota a qualquer momento. A classe `Route` possui ainda um método que permite que a rota seja mostrada durante a simulação, para que o usuário possa visualizar com clareza por onde a rota passa. A classe `MobileEntity`, além do método `walkRoute`, que diz para a entidade iniciar o caminhamento por determinada rota, possui também as funções `pause`, `resume` e `stop`, que respectivamente pausam, retomam e param o caminhamento da entidade.

A classe `Router` (roteador) faz o papel de um gerente de rotas. Ele gerencia um conjunto de rotas, onde o desenvolvedor pode definir regras que indicam como um objeto pode passar de uma rota para outra dentro daquele conjunto, ou para uma rota em outro conjunto, gerenciado por outro `Router`.

O exemplo 5.3.1 mostra a utilização do sistema de rotas.

```
-- cria uma rota
rota = Route()
-- adiciona pontos à rota
rota:addPoint(0,0,0)
rota:addPoint(100,0,50)
...
-- faz a rota ser desenhada na cena
rota:showRoute(true)
-- cria uma entidade móvel
carro = MobileEntity("meu_carro", "carro.mesh")
-- define a velocidade de caminhamento
carro: setWalkSpeed(50)
-- inicia o caminhamento na rota
carro:walkRoute(rota)
```

Exemplo 5.3.1 - Utilização do sistema de rotas

Uma vez que o método `walkRoute` é chamado, a entidade passa a percorrer a rota automaticamente, até chegar ao final, ou até que o usuário chame um dos métodos `pause` ou `stop`. A figura 5.3.3 mostra uma simulação com uma rota sendo desenhada e um veículo a percorrendo.

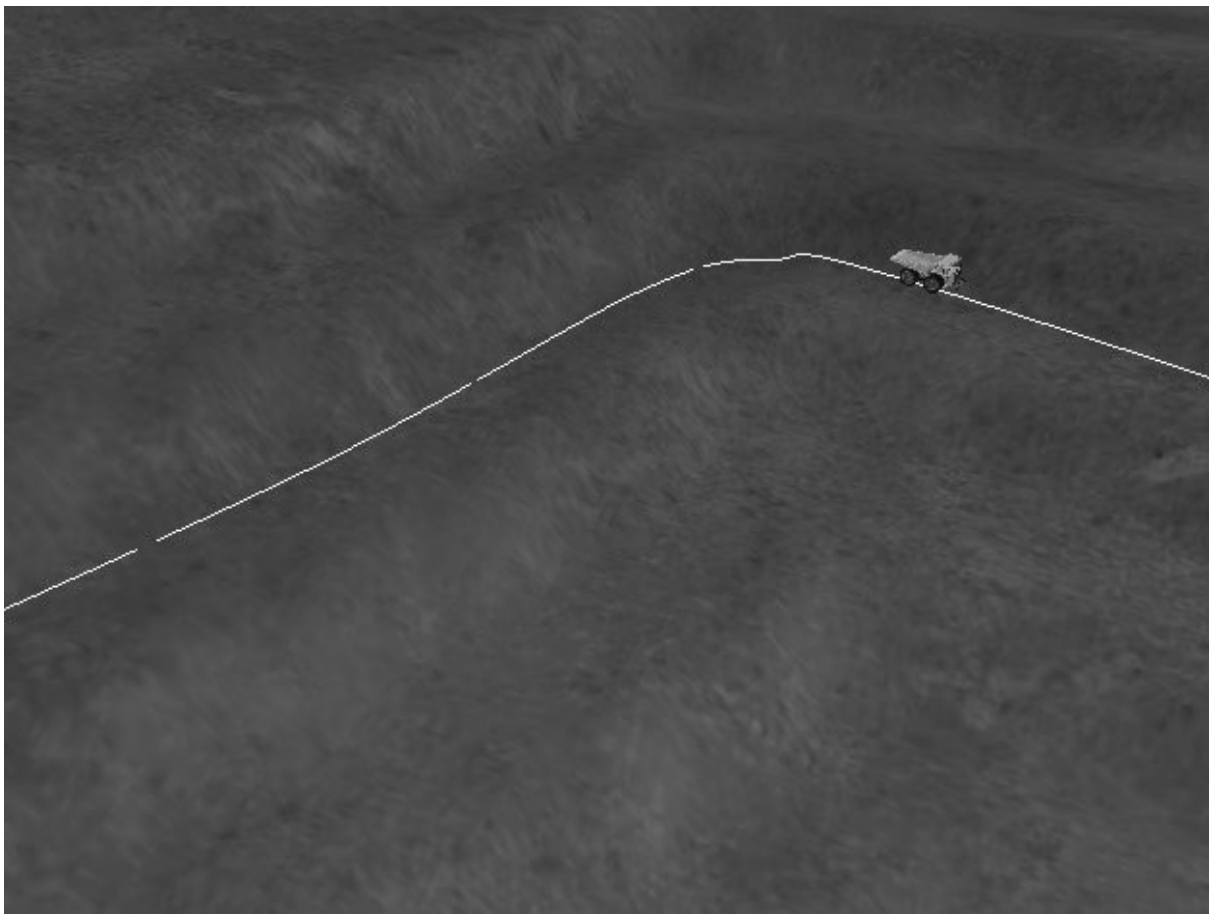

Figura 5.3.3 - Simulação com uma entidade percorrendo uma rota

5.4. Sistema de Física e Colisão

Durante a primeira parte do desenvolvimento do aplicativo, iniciamos a implementação do módulo de física e tratamento de colisão utilizando a biblioteca *Newton Game Dynamics*. Até então escrevemos algumas classes e fizemos testes, mas não havíamos exportado funções deste módulo para Lua.

Neste trabalho o desenvolvimento do módulo avançou a um ponto onde pode ser utilizado eficientemente. Foram exportadas para a linguagem Lua funções para inicializar a simulação física e definir a gravidade padrão. Foram implementados também métodos para tornar uma entidade parte da simulação física, e torná-la suscetível à aplicação de forças, da gravidade, e da colisão com outras entidades. O exemplo 5.4.1 mostra a utilização do sistema de física.

```
-- inicializa a simulação fisica
initPhysics(true)
-- define a gravidade
setGravity(9.8)

-- cria um objeto
objeto = Entity("meu_obj", "objeto.mesh")
-- torna o objeto ativo na simulação física
objeto:setPhysicsActive(true)
-- define a massa do objeto
objeto:setMass(10)

-- a partir daqui o objeto já sofre a ação da gravidade, de forças
-- aplicadas a ele, e é capaz de colidir com outros objetos

-- aplicação de uma força para cima no objeto
objeto:addForce(0, 20, 0)
```

Exemplo 5.4.1 - Utilização do sistema de física

Este módulo ainda não oferece todas as funcionalidades presentes na biblioteca *Newton*, como definição de materiais, colisão com formas arbitrárias (por exemplo, um terreno), criação de juntas. No entanto, já conseguimos criar uma interface coerente com o resto da aplicação, o que torna mais fácil a adição de novas funcionalidades.

5.5. Tratamento de Áudio

Na primeira parte do trabalho não havíamos ainda desenvolvido o tratamento de áudio e, portanto o aplicativo não oferecia nenhum suporte a sons. Nesta etapa utilizamos a API *OpenAL* e criamos o módulo de tratamento de áudio.

A principal classe do módulo é a `AudioEngine`, que é uma classe *singleton*, responsável por inicializar a *OpenAL* e preparar o hardware de áudio, e atualizar as posições do emissores e ouvintes.

A classe `SoundEmitter` representa uma fonte emissora de som. Ela é ligada a um nodo de cena (lembrando que o nodo de cena guarda informações de posição e orientação) e fornece funções para tocar e parar o som (*play*, *pause*), definir volume e repetição (*loop*). A classe

`AudioEngine` atualiza as posições dos emissores de acordo com a posição do respectivo nodo de cena.

A classe `Sound` é responsável por carregar um arquivo de som (por exemplo, WAV) e guardar estas informações em um buffer do *OpenAL*. Um objeto `Sound` pode ser designado a um ou mais emissores.

O exemplo 5.5.1 mostra algumas das funções do módulo de áudio que foram exportadas para a linguagem Lua.

```
-- define a câmera como o ouvinte
setListener(cam)
-- carrega um arquivo de som (o motor de um carro)
sndMotor  = Sound("engine.wav")
-- define um emissor (passando uma entidade, no caso um carro)
emissor = SoundEmitter(carro)
-- define o som que o emissor irá emitir
emissor:setSound(sndMotor)
-- ativa repetição
emissor:setLoop(true)
-- emite o som (que irá se repetir até que seja chamado um stop ou pause)
emissor:play()
```

Exemplo 5.5.1 - Utilização do módulo de áudio

6. Resultados

Nesta segunda etapa do trabalho foram desenvolvidas novas funcionalidades para os módulos existentes, e novos módulos foram integrados ao conjunto. Dentre as adições aos módulos existentes destacamos:

- No módulo de renderização, foram criadas funções para aplicar força e torque às entidades, bem como definir sua massa e estrutura de colisão. Funções estas que se comunicam com o módulo de física.
- O módulo de física e tratamento de colisões foi evoluído e está em um ponto onde pode ser utilizado eficientemente. Forças aplicadas às entidades são calculadas, levando em consideração suas massas e formas de colisão, e o resultado é atualizado na tela.
- Foi criado o sistema de rotas, utilizando *splines*, permitindo várias rotas, e várias entidades móveis.
- Foi implementado com sucesso o tratamento de joysticks.
- O módulo GUI foi adicionado, e a integração com elementos de interface gráfica da biblioteca Qt foi feita com sucesso.
- O módulo de tratamento de áudio foi criado.

A figura 6.1 apresenta o diagrama de classes atualizado, mostrando os novos módulos GUI, Áudio e Física, além de novas classes adicionadas nos módulos existentes.

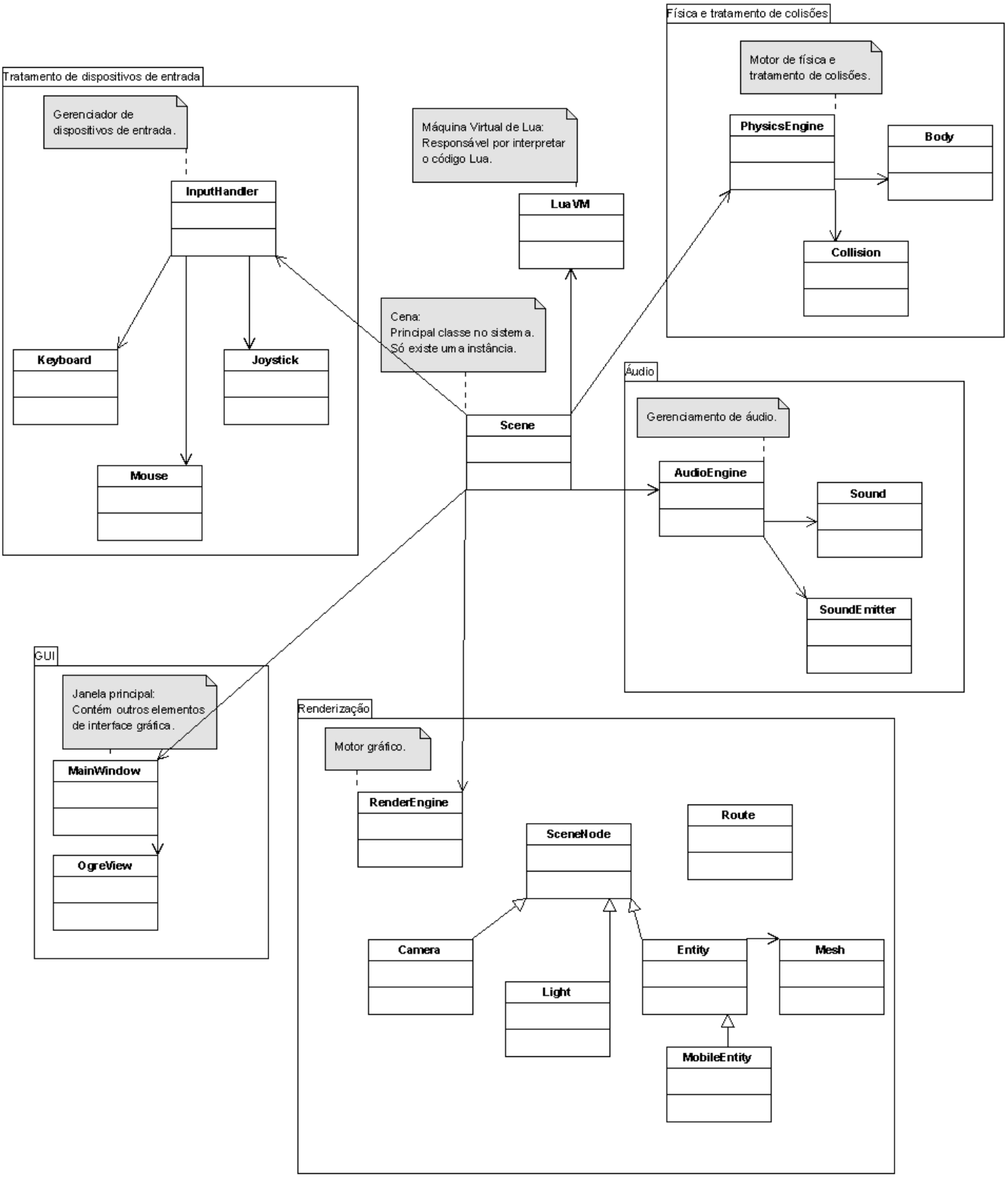

Figura 6.1 - Diagrama de classes do aplicativo desenvolvido

O exemplo 6.1 mostra o código Lua com algumas das novas funcionalidades desenvolvidas, e a figura 6.2 mostra a cena resultante, destacando algumas rotas que podem ser visualizadas na cena

```
-- inicialização
showSplashScreen(true)
setSplashText("Inicializando...")
createWindow(640,480,false,true,DIRECT3D)

-- câmera
cam = Camera("cam")
-- céu
setSkybox("islands.jpg", 10)
-- terreno
setTerrain("terreno.cfg")

-- entidade móvel
carro = MobileEntity("carro", "4x4b.mesh")
carro:setWalkSpeed(30)

-- rota
rota = Route()
rota:addPoint(0,50,0)
rota:addPoint(100,60,20)
rota:addPoint(150,80,80)
rota:addPoint(80,80,120)
rota:finalize()
rota:showRoute(true)
carro:walkRoute(rota)

-- áudio
setListener(cam)
musica = Sound("../media/music.wav")
pivo = SceneNode("pivo")
player = SoundEmitter(pivo)
player:setSound(musica)
player:setLoop(true)
player:setVolume(0.0)
player:play()

showSplashScreen(false)

setInfoText("Alô Mundo!")
showInfoWindow(true)

-- loop principal
function updateWorld()
      if keyDown(KEY_UP) then
            cam:translate(-velMov*dt,0,0)
      elseif keyDown(KEY_DOWN) then
            cam:translate(velMov*dt,0,0)
      end

      if keyHit(KEY_P) > 0 then
            player:pause()
      end
end
```

Exemplo 6.1 - Código Lua de uma simulação

Figura 6.2 - Aplicativo em execução

7. Conclusão

A visualização tridimensional e a interatividade em tempo-real com modelos 3D de grandes empreendimentos de diversas áreas da engenharia (civil, ambiental, de produção, mecânica, metalúrgica, de minas) facilitam o diálogo e a discussão de complexos projetos por parte dos membros da equipe de desenvolvimento e dos investidores. Tanto o entendimento da obra, quanto o diagnóstico de problemas são facilitados e agilizados. Desta maneira, tempo e custos são reduzidos.

A integração em um único aplicativo das características atribuídas a um sistema de realidade virtual (SRV), um sistema de informação geográfica (SIG) e a um sistema de modelagem e simulação (SMS), constitui um grande avanço nas ferramentas computacionais para o apoio à tomada de decisão e análise de impactos utilizados pelas indústrias e órgãos de gestão governamentais (prefeituras, governos estaduais, etc.).

Através da facilidade e da estensibilidade da linguagem *Lua*, podemos oferecer as funcionalidades de um motor gráfico como o *OGRE 3D*, aliado a ferramentas como uma biblioteca de simulação de física, e de tratamento de eventos, de uma forma intuitiva, com uma baixa curva de aprendizagem.

8. Referências Bibliográficas

[1] Monografia: Interatividade em Tempo-Real com Simulações Tridimensionais e Georeferenciadas. Disponível em: <http://www.douglasmatoso.com/docs/monografia-po.pdf>. Acesso em: 05 Janeiro 2008

[2] Carneiro, Tiago G.S. (2006). *Nested-CA: a foundation for multiscale modeling of land use and land change*. Tese de doutorado em Computação Aplicada apresentada ao Instituto Nacional de Pesquisas Espaciais, São José dos Campos, INPE.

[3] *TerraLAB: Laboratório Associado INPE/UFOP para Modelagem e Simulação de Sistemas Terrestres*. Disponível em: <http://www.terralab.ufop.br/>. Acesso em: 05 Janeiro 2008

[4] *VRContext - Walkinside*. Disponível em: <http://www.walkinside.com>. Acesso em: 05 Janeiro 2008

[5] *Act-3D – Quest3D*. Disponível em: <http://www.quest3d.com>. Acesso em: 05 Janeiro 2008

[6] *VRToolbox - VRWorx*. Disponível em: <http://www.vrtoolbox.com/vrworx26.html>. Acesso em: 05 Janeiro 2008

[7] Ierusalimschy, R. *Programming in LUA*, segunda edição, 2006.

[8] *The Programming Language Lua*. Disponível em: <http://www.lua.org/>. Acesso em: 05 Janeiro 2008

[9] *ExtremeTech: Game Engine Anatomy 101*. Disponível em: <http://www.extremetech.com/article2/0,3973,594,00.asp>. Acesso em: 05 Janeiro 2008

[10] Eberly, D. H. *3D Game Engine Architecture: Engineering Real-Time Applications with Wild Magic*, 2004

[11] *DevMaster.net: 3D Engines Database*. Disponível em: <http://www.devmaster.net/engines/>. Acesso em: 05 Janeiro 2008

[12] *OGRE 3D: Open-Source Graphics Engine*. Disponível em: <http://www.ogre3d.org/>. Acesso em: 05 Janeiro 2008

[13] Junker, G. *Pro OGRE 3D Programming*, 2005

[14] *Irrlicht Engine: A Free Open-Source 3D Engine*. Disponível em: <http://irrlicht.sourceforge.net/>. Acesso em: 05 Janeiro 2008

[15] *Crystal Space 3D*. Disponível em: <http://www.crystalspace3d.org/>. Acesso em: 05 Janeiro 2008

[16] *OIS: Object-Oriented Input System*. Disponível em: <http://www.wreckedgames.com/wiki/index.php/WreckedLibs:OIS>. Acesso em: 05 Janeiro 2008

[17] *Qt - Trolltech*. Disponível em: <http://trolltech.com/products/qt>. Acesso em: 05 Janeiro 2008

[18] *Newton Game Dynamics*. Disponível em: <http://www.newtondynamics.com>. Acesso em: 05 Janeiro 2008

[19] *OpenAL*. Disponível em: <http://www.openal.org/>. Acesso em: 05 Janeiro 2008

[20] *Efeito Doppler - Wikipedia*. Disponível em: <http://pt.wikipedia.org/wiki/Efeito_Doppler>. Acesso em: 05 Janeiro 2008

[21] *ToLUA++: Binding C/C++ Code to Lua*. Disponível em: <http://www.codenix.com/~tolua/>. Acesso em: 05 Janeiro 2008