

Sistema FIEB



PELO FUTURO DA INOVAÇÃO

CENTRO UNIVERSITÁRIO SENAI CIMATEC
Programa de Pós-Graduação em
Modelagem Computacional e Tecnologia Industrial

RAÍ FAUSTINO MIRANDA SANTOS

**COMPARATIVOS DE MOTORES DE FÍSICA PARA JOGOS 3D OPEN SOURCE
PARA INTEGRAÇÃO COM GUARASCRIPT**

SALVADOR

2021

RAÍ FAUSTINO MIRANDA SANTOS

**COMPARATIVOS DE MOTORES DE FÍSICA PARA JOGOS 3D OPEN SOURCE
PARA INTEGRAÇÃO COM GUARASCRIPT**

Dissertação apresentada ao Programa de Pós-Graduação em Modelagem Computacional e Modelagem Industrial do Centro Universitário SENAI CIMATEC, como requisito parcial para a obtenção do título de Mestre em Modelagem Computacional e Tecnologia Industrial.

Orientador: Prof. Dr. Roberto Luiz de Sousa Monteiro

SALVADOR

2021

FICHA CATALOGRÁFICA

Ficha catalográfica elaborada pela Biblioteca do Centro Universitário SENAI CIMATEC

S237c Santos, Raí Faustino Miranda

Comparativos de motores de física para jogos 3D open source para integração com GuaraScript / Raí Faustino Miranda Santos. – Salvador, 2021.

57 f. : il. color.

Orientador: Prof. Dr. Roberto Luiz Sousa Monteiro.

Dissertação (Mestrado em Modelagem Computacional e Tecnologia Industrial) – Programa de Pós-Graduação, Centro Universitário SENAI CIMATEC, Salvador, 2021.

Inclui referências.

1. Motor de física. 2. Open-Source. 3. Simulação 3D. I. Centro Universitário SENAI CIMATEC. II. Monteiro, Roberto Luiz Sousa. III. Título.

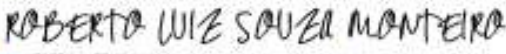
CDD 005.13

CENTRO UNIVERSITÁRIO SENAI CIMATEC

Mestrado Acadêmico em Modelagem Computacional e Tecnologia Industrial

A Banca Examinadora, constituída pelos professores abaixo listados, aprova a Defesa de Mestrado, intitulada “**COMPARATIVOS DE MOTORES DE FÍSICA PARA JOGOS 3D OPEN SOURCE PARA INTEGRAÇÃO COM GUARASCRIPT**” apresentada no dia 30 de agosto de 2021, como parte dos requisitos necessários para a obtenção do Título de Mestre em Modelagem Computacional e Tecnologia Industrial.

Orientador:


DocuSigned by:

AF98264FC084A8
Prof. Dr. Roberto Luiz Souza Monteiro
SENAI CIMATEC

Membro Interno:

DocuSigned by:

78651980D01F66D
Prof. Dr. Renelson Ribeiro Sampaio
SENAI CIMATEC

Membro Interno:

DocuSigned by:

55E77348B4
Prof. Dr. Marcelo Albano Moret Simões Gonçalves
SENAI CIMATEC

Membro Externo:

DocuSigned by:

46C3CF2DAF0E49F
Prof. Dr. José Roberto de Araújo Fontoura
UNEB

Dedico este trabalho aos meus pais por investirem e acreditarem em mim.

AGRADECIMENTOS

Agradeço aos meus pais pela educação e amor, à família e amigos pelo apoio, aos colegas de Mestrado pelo companheirismo e ao prof. Roberto Monteiro por toda a orientação.

RESUMO

COMPARATIVOS DE MOTORES DE FÍSICA PARA JOGOS 3D OPEN SOURCE PARA INTEGRAÇÃO COM GUARASCRIPT

Este trabalho tem o objetivo de avaliar a possibilidade do desenvolvimento de um motor de física para jogos 3D (game physics engine) utilizando integração com a linguagem de programação GuaraScript, desenvolvida pelo professor Roberto Monteiro. Para isso, foi feita uma modelagem computacional de fenômenos da física clássica, assim como uma pesquisa do estado-da-arte dos *game engines open-source* disponíveis no mercado, acompanhado de um comparativo para definir qual o melhor candidato para integração com a linguagem *GuaraScript*. O motor de física Newton Dynamics foi escolhido como o mais apto para a integração, e foi diagramado um programa de teste em formato UML. Foi concluído que, com as ferramentas supracitadas, é possível e viável programar um motor de física 3D open-source confiável para utilização em diversas aplicações na indústria e em ambientes educacionais.

Palavras-chave: Motor de Física, Open-Source, Simulação 3D.

ABSTRACT

COMPARISONS OF PHYSICS ENGINES FOR OPEN SOURCE GAMES FOR INTEGRATION WITH GUARASCRIPT

This work aims to evaluate the possibility of developing a physics engine for 3D games (game physics engine) using integration with the GuaraScript programming language, developed by professor Roberto Monteiro. For this, a computational modeling of classical physics phenomena was carried out, as well as a survey of the state-of-the-art of open-source game engines available on the market, accompanied by a comparison to define the best candidate for integration with the GuaraScript language. The Newton Dynamics physics engine was chosen as the best fit for integration, and a test program in UML format was diagrammed. It was concluded that, with the aforementioned tools, it is possible and feasible to program a reliable open-source 3D physics engine for use in various applications in industry and educational environments.

Key words: Physics Engine, Open-Source, 3D Simulation

LISTA DE FIGURAS

FIGURA 1 -	Desenvolvimento em interface Gamedeveloper	14
FIGURA 2 -	Desenvolvimento em interface Unity	15
FIGURA 3 -	Desenvolvimento em interface OGRE 3D	16
FIGURA 4 -	Desenvolvimento em interface Unreal Engine	17
FIGURA 5 -	Desenvolvimento em interface Marmalade	18
FIGURA 6 -	Desenvolvimento em interface Shiva Game Engine	19
FIGURA 7 -	Logo da game engine Havok	19
FIGURA 8 -	Desenvolvimento em interface Blender Game Engine	20
FIGURA 9 -	Still de jogo desenvolvido em Bullet	21
FIGURA 10 -	Still de jogo desenvolvido em Newton Game Dynamics	22
FIGURA 11 -	Janela debug do Bullet Physics Example Browser	24
FIGURA 12 -	Sketch do personagem com os eixos 3d	25
FIGURA 13 -	Diagrama de caso de uso	27
FIGURA 14 -	Diagrama de classes	28
FIGURA 15 -	Diagrama de objetos	28
FIGURA 16 -	Diagrama de implantação	29
FIGURA 17 -	Diagrama de componentes	29

LISTA DE TABELAS

TABELA 1 - Comparativo dos motores físicos	23
--	----

SUMÁRIO

RESUMO	IV
ABSTRACT	V
LISTA DE ILUSTRAÇÕES	6
1 INTRODUÇÃO	10
1.1 CONTEXTUALIZAÇÃO	10
1.2 JUSTIFICATIVA	10
1.3 HIPÓTESE	10
1.4 OBJETIVO.....	11
1.5 METODOLOGIA.....	11
2 REVISÃO DE LITERATURA	12
2.1 MOTORES DE FÍSICA: DEFINIÇÃO E HISTÓRICO.	12
2.2 MOTORES DE FÍSICA POPULARES.....	14
2.2.1 GEMEMAKER	14
2.2.2 UNITY.....	15
2.2.3 OGRE 3D	16
2.2.4 UNREAL ENGINE	17
2.2.5 MARMALADE.....	18
2.2.6 SHIVA GAME ENGINE	19
2.2.7 HAVOK.....	19
2.2.8 BLENDER GAME ENGINE	20
2.2.9 BULLET	21
2.2.10 NEWTON GAME DYNAMICS.....	22
3 MATERIAIS E MÉTODOS	23
3.1 COMPARATIVO	23
3.2 EXEMPLO DE SIMULAÇÃO	24
3.3 MODELO DA SIMULAÇÃO	25
3.4 DIAGRAMAS UML	27
3.4.1 DIAGRAMA DE CASO DE USO	27
3.4.2 DIAGRAMA DE CLASSES.....	28
3.4.3 DIAGRAMA DE OBJETOS.....	28
3.4.4 DIAGRAMA DE IMPLANTAÇÃO.....	29
3.4.5 DIAGRAMA DE COMPONENTES	29

4	DISCUSSÃO	30
5	CONCLUSÕES	31
	REFERÊNCIAS.....	32

1 INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO

As simulações realistas em ambientes 3D cumprem um papel importante na nossa sociedade. Elas são frequentemente associadas a videogames, mas também se fazem presentes em ambientes virtuais educacionais, e também em simuladores medicinais voltados para a capacitação de profissionais (MONTEIRO¹, 2006).. Portanto, a implementação deste tipo de simulação, além de entreter, também é parte fundamental para a formação de muitos profissionais ao redor do mundo. Estas simulações dependem, além da parte gráfica, de um motor de física realista, para simular fenômenos físicos no ambiente virtual de maneira fidedigna (MARESCAUX, 1998).

1.2 JUSTIFICATIVA

O desenvolvimento de aplicações científicas e jogos digitais envolvem, com frequência, representações fidedignas no mundo real. Essas representações requerem cálculos e algoritmos complexos, difíceis de serem implementados nas linguagens de programação mais utilizadas para construção desse tipo de aplicação. A linguagem GuaraScript foi desenvolvida por Roberto Luiz Sousa Monteiro com a intenção de facilitar a programação de plataformas virtuais para profissionais acadêmicos que, porventura, não tenham muita experiência em trabalhar com código. A maior parte dos motores físicos disponíveis estão programados em C e C++, linguagens que não são necessariamente intuitivas para leigos na área. A idéia deste trabalho é identificar qual dos motores físicos disponíveis no mercado podem ser integrados com mais facilidade com a linguagem de programação em questão, democratizando assim o acesso à simulação 3D (MONTEIRO, 2005)..

1.3 HIPÓTESE

Uma engine de física open source pode ser integrada a linguagem GuaraScript e permitir um desenvolvimento mais fácil e rápido de jogos e simulações 3D.

1.4OBJETIVO

Esta pesquisa tem por objetivo modelar o desenvolvimento de um motor de física para jogos 3D (game physics engine) utilizando a linguagem de programação GuaraScript, permitindo a representação computacional de fenômenos da física clássica, tais como aceleração, queda livre, colisões elásticas etc.

Especificamente:

1. Serão definidos quais os fenômenos físicos relevantes para simulação em jogos 3D;
2. Será estudado o histórico de game engines para melhor compreensão da evolução e do estado-da-arte;
3. Serão comparados os engines 3D disponíveis para definição do mais indicado;
4. Será implementado um protótipo de um fenômeno físico no engine escolhido.

1.5METODOLOGIA

Existem diversas game engines no mercado, portanto há muitas possibilidades de pontos de partida para atender o escopo deste trabalho. Porém, existem 3 pontos chave para possibilitar a integração com a linguagem GuaraScript, e entre eles temos:

- Capacidade de trabalhar com gráficos em 3D;
- Ser open-source, isto é, gratuita e de código aberto;
- Ser programada em C ou possuir API em C para integração com GuaraScript.

Com essas diretrizes sendo utilizadas como ponto de partida, foi iniciada a pesquisa, e após selecionar a melhor engine dentro do contexto, foi feita a diagramação UML de uma aplicação teste.

2 REVISÃO DE LITERATURA

2.1 MOTORES DE FÍSICA: DEFINIÇÃO E HISTÓRICO.

Os chamados motores de jogo (game engines) são um conjunto de bibliotecas que simplificam o desenvolvimento de jogos ou de qualquer outro tipo de aplicação com gráficos em tempo real. Os game engines podem ser divididos em duas principais categorias - um motor gráfico para renderizar gráficos 2D ou 3D; e um motor de física para simular eventos físicos reais dentro do ambiente virtual, utilizando sistemas como detecção de colisão (BOYER, 2007). Motores gráficos processam os dados abstraídos de alto nível necessários para renderizar os gráficos do jogo, e geram dados de baixo nível que o hardware possa compreender. Ex: OGRE, Crystal Space, OpenSceneGraph. Já motores de física simulam ações reais que respeitam as leis da física, através de variáveis como gravidade, massa, atrito, força e flexibilidade. Ex: Havok, Bullet, ODE.

Ao utilizar um game engine, o processo de desenvolvimento de jogos é otimizado de maneira considerável, já que é possível utilizar a mesma biblioteca como base para criar vários jogos, ou mesmo facilitar a adaptação do jogo para diferentes plataformas. A maioria dos engines é distribuída em forma de API, mas alguns são distribuídos em um conjunto de ferramentas chamado de middleware (BAKKEN, 2001). Isso facilita mais ainda o processo, pois utilizar ferramentas como IDEs e scripts pré-programados possibilita deixar o jogo pronto para distribuição; além de dispensar a compra de outras ferramentas, reduzindo os custos e aumentando a competitividade dentro da indústria.

Antes dos game engines, os jogos ,eram escritos como um código único, sem separação de áreas como gráfica e física. Era necessário manter o código o mais simples possível, para fazer uso otimizado do hardware, por causa de suas limitações da década de 80 (SCHERER, BATISTA, DE CANTALICE MENDES, 2020). Devido a isso, não havia muita reutilização de código entre um jogo e outro, então não existia um padrão de biblioteca que pudesse cuidar da performance gráfica ou física. Outro fator foi o desenvolvimento rápido do hardware ao longo dos anos, que criava uma

demanda por novos game designs que pudessem aproveitar os recursos extras que a nova tecnologia trazia.

A nomenclatura “game engine” surgiu em meados da década de 90, associada ao desenvolvimento de jogos 3D do estilo FPS (first person shooter, ou jogos de tiro em primeira pessoa). Ao invés de construir o jogo do zero, outros desenvolvedores licenciaram os núcleos dos jogos, usando como base para seus próprios game engines. Jogos como *Doom* e *Quake*, que eram muito populares nesta época, foram alguns dos que utilizaram desse tipo de engine. Com a consolidação desse tipo de tecnologia, os game engines passaram a ser utilizados em outras áreas além do desenvolvimento de jogos, como demonstrações, visualizações arquiteturais, simulações de treinamento (como de pilotagem de aeronaves e manuseio de armas), ferramentas de modelagem e, simulações físicas para a criação de animações e cenas de filmes realistas. (GAZETTE, 2011)

APIs como DirectX e OpenGL surgiram e impulsionaram a evolução das tecnologias usadas nos jogos, ajudando no desenvolvimento do mercado. O DirectX foi lançado em 30 de setembro de 1995, permitindo todas as versões do Microsoft Windows (a partir do Windows 95) a incorporar multimídia de alto desempenho. Apesar do OpenGL ter surgido antes (janeiro de 1992), o DirectX teve (e continua tendo) mais aceitação na área de desenvolvimento de jogos.

Motores de jogo modernos são aplicações extremamente complexas, cuja contínua evolução tem criado uma separação forte entre rendering, scripting, arte e level design. Atualmente é muito comum que um time de desenvolvimento de games tenha artistas e programadores trabalhando lado a lado. Como a maioria dos jogos 3D busca explorar ao máximo o poder de processamento do *hardware*, se tornou um mau negócio utilizar diretamente linguagens de programação de alto nível como C#, Java e Python, pois elas acarretam em perda de desempenho. Por outro lado, estas mesmas linguagens oferecem um ganho de produtividade muito bem-vindo para os desenvolvedores de motor de jogo, que as utilizam para compilar suas bibliotecas.

Os preços de motores de jogos variam bastante. Existem engines gratuitas open-source, porém existem também aquelas cujo preço pode variar de dezenas até dezenas de milhares de dólares. Cada caso vai indicar qual o melhor engine a ser utilizado, baseado nas necessidades técnicas e no orçamento disponível. A indústria dos games expandiu exponencialmente com o passar dos anos, conseguindo um lucro de US\$ 9,5 bilhões, em 2007, ultrapassando a indústria cinematográfica. A receita bruta em 2007 alcançou US\$ 18,8 bilhões, e dez anos depois, em 2017, alcançou US\$ 36 bilhões. (BUSINESS WIRE, 2018)

Por muito tempo as companhias se dedicaram a fazer suas próprias engines. Porém, com o tempo e o amadurecimento da indústria e o aumento da exigência dos consumidores, o custo de fazer engines cresceu muito, criando um nicho no mercado - as companhias especializadas na construção de engines. Para uma companhia desenvolvedora de jogos, hoje em dia tornou-se muito oneroso e demorado fazer sua própria engine para depois construir o jogo, então as companhias especializadas surgem para suprir essa demanda.

2.2 MOTORES DE FÍSICA POPULARES

2.2.1 GEMEMAKER

FIGURA 1 - Desenvolvimento em interface Gamedeveloper



Fonte: Steam, 2020

GameMaker foi lançado pela primeira vez com o nome “Animo” em 1991, e foi adequado para criar Jogos 2D. Embora o mecanismo de jogo seja adequado para

jogos 2D, ele permite que o usuário adicione gráficos 3D e física. O GameMaker é fácil de usar para usuários novatos, já que não requer programação conhecimento. No entanto, a funcionalidade da câmera 3D em comparação com gráficos 3D é limitada (GAMEMAKER, 2020).

2.2.2 UNITY

FIGURA 2 - Desenvolvimento em interface Unity

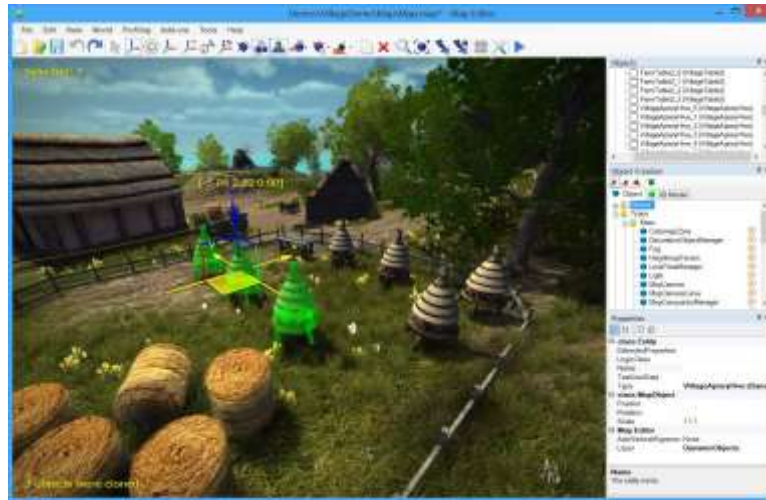


Fonte:Unity, 2020

A Unity oferece aos usuários a capacidade de criar jogos em 2D e 3D, utilizando o MonoDevelop para a criação dos scripts. Ela suporta APIs de Direct3D (Windows, Xbox 360); OpenGL (MacOS, Linux); OpenGL ES (Android iOS) e WebGL (Internet). Nos jogos 2D, a Unity permite a importação de sprites e um avançado renderizador de mundo 2D. Para jogos 3D, a Unity permite a especificação de compressão de textura, mipmaps e configurações de resolução para cada plataforma suportada pelo mecanismo de jogo, e fornece suporte para mapeamento de relevo, mapeamento de reflexão, mapeamento de paralaxe, oclusão de ambiente de espaço de tela (SSAO) sombras usando mapas de sombras, efeitos de pós-processamento de renderização para textura e tela inteira (UNITY, 2020).

2.2.3 OGRE 3D

FIGURA 3 - Desenvolvimento em interface OGRE 3D



Fonte: OGRE 3D, 2020

OGRE (acrônimo para "Object-oriented Graphics Rendering Engine") é um motor gráfico 3D orientado a objetos. A principal linguagem usada por ele é C++, porém existem versões de teste para Python, Java e Microsoft .NET. Esta biblioteca possui um sistema de scripts, que permite declarar materiais, pós processadores, sistema de partículas e shaders. Deste modo, a compilação do código fonte não é necessária, bastando apenas alterar o script correspondente. Sendo altamente orientada a objetos e com a portabilidade em primeiro plano, a mudança entre sistemas de renderização (OpenGL/DirectX), sistemas operacionais (Windows/Linux/Mac OS) e gerenciadores de cenário ocorre de forma automatizada, graças a uma arquitetura de layers que permite que essas trocas se tornem transparentes (OGRE, 2020).

2.2.4 UNREAL ENGINE

FIGURA 4 - Desenvolvimento em interface Unreal Engine



Fonte: softwareengineer.lk (Medium), 2020

Unreal Engine é um motor de jogo desenvolvido pela Epic Games, usado pela primeira vez em 1998 no jogo de tiro em primeira pessoa Unreal, ele tem sido a base de muitos jogos desde então. Embora usado inicialmente para jogos de tiro em primeira pessoa, ele tem sido usado com sucesso em uma grande variedade de gêneros de jogos. Seu núcleo é escrito em C++, possibilitando a portabilidade (UNREAL ENGINE, 2020).

2.2.5 MARMALADE

FIGURA 5 - Desenvolvimento em interface Marmalade



Fonte: Windows Developer Blog (Windows Blog), 2016

O Marmalade SDK é um kit de desenvolvimento de software multiplataforma e um mecanismo de jogo da Marmalade Technologies Limited (anteriormente conhecido como Ideaworks3D Limited) que contém arquivos de biblioteca, amostras, documentação e ferramentas necessárias para desenvolver, testar e implantar aplicativos para dispositivos móveis. O conceito subjacente do Marmalade SDK é escrever uma vez, executar em qualquer lugar para que uma única base de código possa ser compilada e executada em todas as plataformas suportadas, em vez de precisar ser escrita em diferentes linguagens de programação usando uma API diferente para cada plataforma. Isso é obtido fornecendo uma API baseada em C / C++ que atua como uma camada de abstração para a API principal de cada plataforma (MARMALADE, 2020).

2.2.6 SHIVA GAME ENGINE

FIGURA 6 - Desenvolvimento em interface Shiva Game Engine



Fonte:Unity, 2020

O ShiVa3D é um mecanismo de jogos 3D com um editor gráfico projetado para criar aplicativos e videogames para PCs desktop, web, consoles de jogos e dispositivos móveis. Os jogos feitos com o ShiVa podem ser exportados para mais de 20 plataformas de destino, com novos destinos de exportação sendo adicionados regularmente. Inúmeras aplicações foram criadas usando ShiVa, incluindo o remake de Prince of Persia 2 para Mobiles e Babel Rising publicado pela Ubisoft. Com o ShiVa 2.0, a próxima versão do Editor está atualmente em forte desenvolvimento. Os usuários do ShiVa com licenças anteriores a 1º de janeiro de 2012 são convidados a baixar as versões beta, testar minuciosamente e fornecer feedback (SHIVA, 2020).

2.2.7 HAVOK

FIGURA 7 - Logo da game engine Havok



Fonte:Wikipedia, 2020

Havok, também conhecido como Havok Physics, é um motor de física desenvolvido pela companhia Havok. Ele é desenhado para jogos eletrônicos, permitindo interação entre objetos em tempo real e dando qualidades físicas aos objetos. O Havok foi comprado pela Intel em 2007. Em 2008, Havok foi premiado no 59º Annual Technology & Engineering Emmy Awards por avançar o desenvolvimento de motores de física em entretenimento eletrônico (HAVOK, 2020).

2.2.8 BLENDER GAME ENGINE

FIGURA 8 - Desenvolvimento em interface Blender Game Engine



Fonte:CG Cookie (canal Youtube), 2012

Blender Game Engine, também conhecido como BGE, Game Blender ou Ketsji, é o motor de jogo do Blender, uma aplicação de código aberto popular. Ele foi desenvolvido para criação de aplicações interativas em 3D, tais como, jogos, apresentações, planejamentos arquitetônicos e outros. Está disponível sob a GNU GPL, versão 2 ou posterior (BLENDER, 2020).

2.2.9 BULLET

FIGURA 9 - Still de jogo desenvolvido em Bullet



Fonte:PyBullet, 2016

O Bullet é um mecanismo de física que simula a detecção de colisões, dinâmica corporal macia e rígida. Foi usado em videogames e também para efeitos visuais em filmes. Erwin Coumans, seu principal autor, ganhou um prêmio da Academia Científica e Técnica por seu trabalho em Bullet. Ele trabalhou para a P&D da Sony Computer Entertainment nos EUA de 2003 a 2010, para a AMD até 2014, e agora trabalha para o Google. A biblioteca de física Bullet é um software livre e de código aberto sujeito aos termos da licença zlib. O código fonte está hospedado no GitHub; antes de 2014, ele estava hospedado no Google Code (BULLET, 2020).

2.2.10 NEWTON GAME DYNAMICS

FIGURA 10 - Still de jogo desenvolvido em Newton Game Dynamics



Fonte:NewtonDynamics (canal Youtube), 2017

Newton Dynamics é uma biblioteca de simulação de física semelhante à vida de plataforma cruzada. Ele pode ser facilmente integrado a mecanismos de jogos e outras aplicações e fornece desempenho de primeira classe e estabilidade de simulação. O desenvolvimento contínuo e uma licença permissiva fazem da Newton Dynamics a melhor escolha para todos os tipos de projetos, desde projetos científicos até mecanismos de jogos. A Newton Dynamics implementa um solucionador determinístico, que não se baseia nos métodos tradicionais do LCP ou iterativo, mas possui a estabilidade e a velocidade de ambos, respectivamente. Esse recurso faz do Newton Dynamics uma ferramenta não apenas para jogos, mas também para qualquer simulação de física em tempo real. (NEWTON, 2020)

3 MATERIAIS E MÉTODOS

3.1 COMPARATIVO

Para atender o escopo deste trabalho de integrar um motor de física à linguagem GuaraScript, a engine escolhida precisaria atender aos seguintes requisitos:

1. Possuir capacidade de trabalhar com gráficos em 3D;
2. Ser open-source, isto é, ser gratuita e de código aberto;
3. Possuir API em C para integração com GuaraScript.

Segue uma tabela comparativa das game engines contempladas na pesquisa, baseada nos requisitos acima:

TABELA 1 - Comparativo dos motores físicos

	Linguagem nativa	3D	Open-Source	API em C/C++
GameMaker	C++	limitada	X	X
Unity	C++	✓	X	X
OGRE 3D	C++	✓	X	X
Unreal Engine	C++	✓	X	X
Marmalade	C++	✓	X	-
Shiva Game Engine	C++	✓	X	-
Havok	C/C++	✓	X	X
Blender Game Engine	C++	✓	✓	-
Bullet	C/C++	✓	✓	✓
Newton Dynamics	C	✓	✓	✓

O primeiro requisito – ser uma engine capaz de lidar com gráficos 3D – contempla a maioria das engines pesquisadas, com exceção da GameMaker, que possui limitações. Logo, ela foi descartada imediatamente.

O segundo requisito – ser uma engine open-source –elimina a maioria das opções restantes. Unity, OGRE 3D, Unreal Engine, Marmalade e Shiva Game Engine são proprietárias, isto é, têm a licença paga. Logo, também foram eliminadas.

O terceiro requisito – possuir API em C – deixa apenas duas engines restante, a Bullet e a Newton Dynamics. A Bullet possibilitaria a integração através da API, mas a Newton Dynamics pode ser compilada em .dll diretamente para ser utilizada em qualquer aplicação em C, como dito no site oficial:

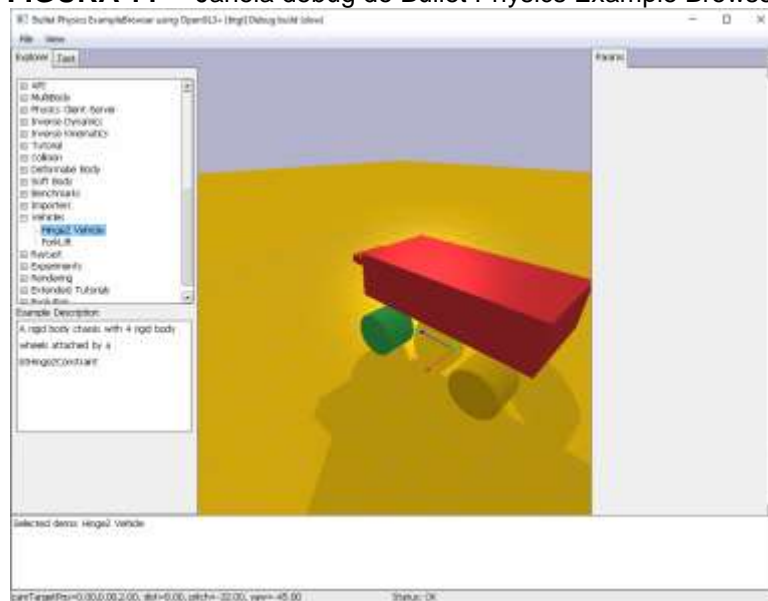
“Qualquer linguagem de programação que possa entender e utilizar a interface C simples (chamadas de funções simples, ponteiros, estruturas e retornos de chamada) pode ser usada, como Delphi e Lazarus / freepascal. Newton pode ser compilado como .dll ou .so para praticamente qualquer plataforma” (NEWTON DYNAMICS, 2020)

Portanto, a Newton Dynamics é a mais indicada para o modelo desse trabalho

3.2 EXEMPLO DE SIMULAÇÃO

Para exemplificar a aparência de uma simulação em ambiente 3D, utilizaremos uma aplicação da biblioteca de introdução da linguagem Bullet 3, acompanhada do seu código fonte (no anexo deste trabalho).

FIGURA 11 - Janela debug do Bullet Physics Example Browser



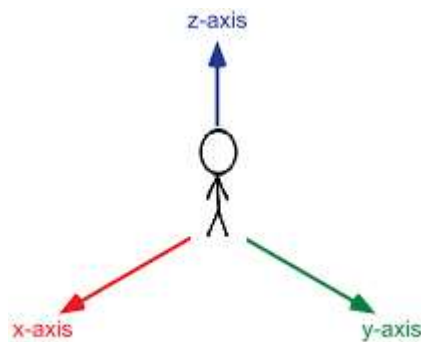
Fonte: Autor

3.3 MODELO DA SIMULAÇÃO

Apresentamos abaixo um modelo simples com funções e atributos que descrevem a movimentação de um personagem no espaço 3D, com duas funções básicas – andar e pular. A ação de andar será atribuída à função “move()” e a ação de pular, à função “jump()”. Elas podem acontecer separadas ou simultaneamente.

A função move() ocasionará mudanças nos eixos x e y, enquanto a função jump() altera a posição do personagem no eixo z.

FIGURA 12 - Sketch do personagem com os eixos 3d



Fonte: Autor

Para detectar movimento, teremos variáveis referentes ao direcionamento do personagem nos eixos x e y, denominadas axisX e axisY, com valores int que variam de -1 a 1. O valor 0 indica que o personagem está estacionário em relação ao eixo referido, enquanto valores diferentes de zero indicam movimento para um lado ou para o outro. O que determina o valor destas variáveis são os comandos direcionais, definidos pelo jogador.

Para controlar o arco de pulo, teremos três constantes – massa, gravidade e força do pulo do personagem. Utilizando a fórmula $F = m.a$ (Força = massa x aceleração), o sistema utiliza a função gravity() para calcular a velocidade atual e sua direção, determinando em quais momentos o personagem estará subindo ou descendo no pulo. Esta função obedecerá a fórmula de movimento uniformemente acelerado, pois assim que o personagem deixa o

contato com o chão, e estará sujeito à ação constante da aceleração de gravidade até retornar ao chão (posição inicial). (MATIAS, 2010)

$$S = S_0 + V \cdot t + a \cdot \frac{t^2}{2}$$

S = Posição atual

S₀ = Posição inicial

V = Velocidade

a = aceleração

t = tempo

A aceleração da gravidade, obedecendo a constante universal da física, consiste num valor negativo de -9,8m/s², armazenado na constante **GForce**. Então, teremos a seguinte fórmula para a função **gravity()**, que retorna o valor de **axisZ**:

$$axisZ = 0 + V \cdot t + (9,8) \cdot \frac{t^2}{2}$$

Para determinar a aceleração inicial para cima, que vai superar a força da gravidade (para que o personagem saia do chão), precisamos saber a massa do personagem e também a força do pulo, valores que serão armazenados em **mass** e **JForce**.

Os valores podem ser configurados arbitrariamente, contanto que a massa seja maior que 0 e **JForce** seja maior em módulo do que **GForce**, ou seja, **JForce** > 9,8.

Com isso, temos as seguintes funções, variáveis e constantes:

- Funções: **move()**, **jump()**, **gravity()**;
- variáveis: **axisX**, **axisY**, **axisZ**;
- constantes: **GForce**, **JForce**, **Mass**.

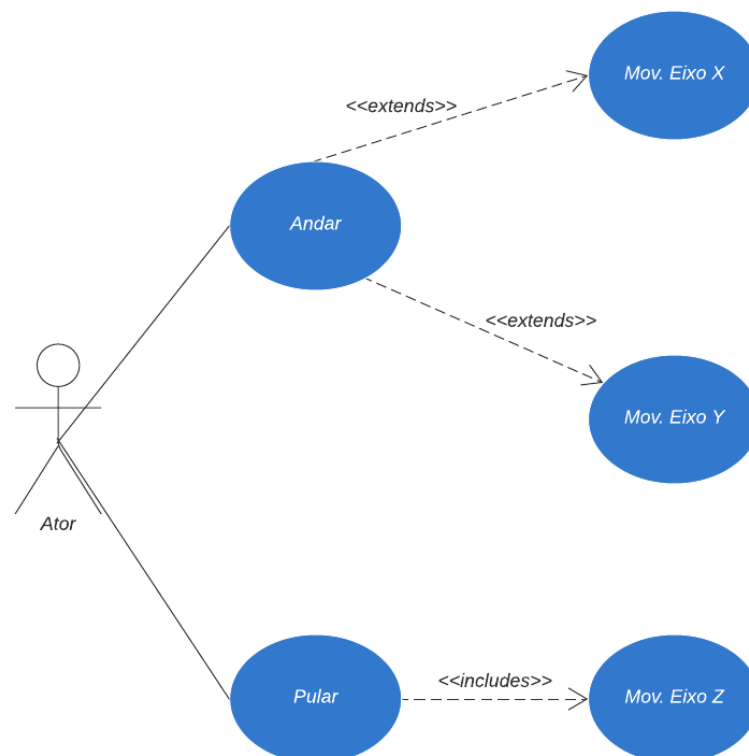
3.4 DIAGRAMAS UML

3.4.1 DIAGRAMA DE CASO DE USO

O diagrama de caso de uso é utilizado dentro do sistema UML para levantar os requisitos de um sistema, descrevendo sua funcionalidade proposta (LARMAN, 2000). Dentro do nosso modelo de simulação de exploração do personagem no mundo 3D, precisamos contemplar as ações de andar e pular.

Na ação de andar, o movimento pode ou não causar alterações no eixo X e eixo Y (pode ser um, o outro, ou os dois simultaneamente), logo utilizaremos o relacionamento *extends*. Já no movimento de pular, ele sempre causará alteração no eixo Z, então o relacionamento será *includes*, devido à obrigatoriedade associada.

FIGURA 13 - Diagrama de caso de uso



Fonte: Autor

3.4.2 DIAGRAMA DE CLASSES

O diagrama de classes define os requisitos de classe para o sistema, representando as relações e a estrutura das mesmas, servindo de modelo para os objetos (LARMAN, 2000). No nosso modelo, variáveis e constantes associadas ao posicionamento e características do personagem (descritas no capítulo 3.1) serão armazenadas na classe Personagem. A constante **GForce** e a função **gravity()** serão parte da classe **Ambiente**.

FIGURA 14 - Diagrama de classes



Fonte: Autor

3.4.3 DIAGRAMA DE OBJETOS

O diagrama de objetos tem uma notação similar ao diagrama de classes, com a diferença de que ele mostra objetos que de fatos são instanciados (LARMAN, 2000). Para fins de ilustração, vamos estabelecer que a simulação ocorre no Ambiente do planeta Terra, com dois personagens sendo controlados, João e Maria.

FIGURA 15 - Diagrama de objetos

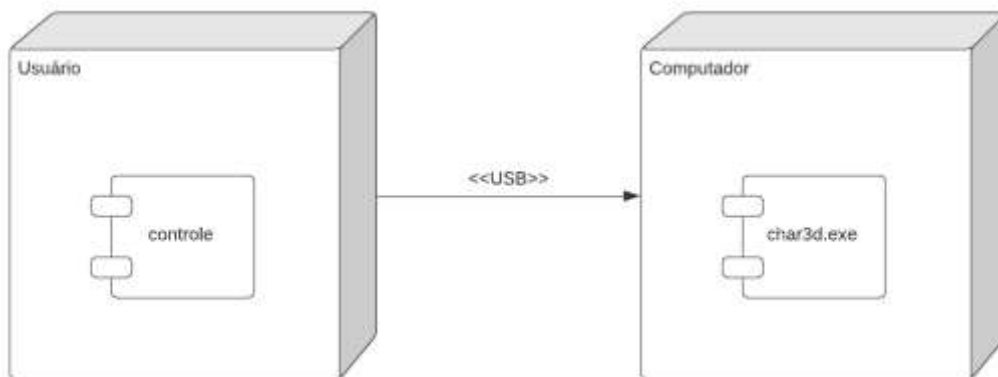


Fonte: Autor

3.4.4 DIAGRAMA DE IMPLANTAÇÃO

O diagrama de implantação descreve os componentes de hardware e software, assim como sua interação com os outros elementos de suporte ao processamento (LARMAN, 2000).

FIGURA 16 - Diagrama de implantação



Fonte: Autor

3.4.5 DIAGRAMA DE COMPONENTES

O diagrama de componentes determina a organização das classes, a fim de modelar os dados do código fonte, destacar a função de cada módulo para facilitar sua reutilização; e auxiliar no processo de engenharia reversa (LARMAN, 2000).

FIGURA 17 - Diagrama de componentes



Fonte: Autor

4 DISCUSSÃO

A importância das simulações 3D no cenário moderno é indiscutível, assim como a necessidade da sua disponibilidade de forma gratuita para profissionais em momentos de aprendizado, sejam eles estudantes de programação com intenção de ingressar na indústria de desenvolvimento, sejam cidadãos comuns ou profissionais de outras áreas que utilizarão o poder realista das simulações para seu devido treinamento (simulações de cirurgias na área médica, simulação automotiva para aulas de direção, entre outros).

Após ser feito um levantamento de possíveis game engines para integração com o GuaraScript, a decisão final ficou entre as engines Bullet e Newton Dynamics – já que ambas possuem capacidade de trabalho com gráficos em 3D, são open-source, e também permitem integrar com a linguagem C, o que é pré-requisito para a harmonia com o GuaraScript.

O modelo de testes se comporta de forma satisfatória em ambas as engines, tendo em vista a existência de jogos consagrados utilizando as plataformas, como *Trials HD* (Bullet) e *Amnesia: Dark Descent* (Newton Game Dynamics). O desempate vem na possibilidade da Newton Dynamics ser compilada em .dll diretamente para ser utilizada em qualquer plataforma, desta forma agilizando a integração com o GuaraScript, que é de suma importância para este trabalho.

Após essa decisão, o trabalho contempla uma diagramação em UML para exemplificar uma aplicação simples em 3D, de um personagem navegando um mundo cuja única restrição de movimento é a força da gravidade. Esse foi o escopo prático abordado nesse trabalho, e pode ser utilizado como ponto de partida para futuras intervenções.

5 CONCLUSÕES

Este trabalho começou com o levantamento da hipótese de uma game engine 3D confiável e realista totalmente open-source, que funcionasse de forma integrada com a linguagem GuaraScript, criada pelo professor e orientador deste trabalho, Roberto Monteiro. O desenvolvimento de uma ferramenta gratuita dessa categoria ajudaria a democratizar o acesso ao desenvolvimento de aplicações em 3D, como videogames, plataformas educacionais, entre outros.

Após o surgimento da ideia, foi delineado o método ideal para que ela se tornasse realidade, que é utilizar uma linguagem já disponível no mercado como referência. Esta engine deveria ter, obviamente, capacidade pra trabalhar com gráficos 3D, além de ser totalmente open-source, e ter suporte à linguagem C (seja nativa ou por API), o que possibilitaria a integração com o GuaraScript.

Após filtrar as game engines mais populares do mercado, restaram dois engines que poderiam se encaixar na proposta, Bullet e Newton Game Dynamics. O modelo de testes se comportou de forma satisfatória em ambas as engines, e o critério de desempate a favor da Newton Game Dynamics foi a possibilidade de compilação da mesma em .dll, desta forma agilizando a integração com o GuaraScript.

Para finalizar, foram feitos diagramas UML para desenvolvimento de uma aplicação básica de navegação de um personagem num ambiente 3D, que pode ser utilizado como ponto de partida para desenvolvimento de jogos e/ou ambientes virtuais interativos. A partir disso, futuros trabalhos podem desenvolver aplicações 3D interativas, com simulações físicas realistas, de forma gratuita, confiável e eficiente.

REFERÊNCIAS

- SCHELL, Jesse. **The Art of Game Design: A book of lenses**. CRC press, 2008.
- MONTEIRO¹, Bruno S. et al. **Anatoml 3D: Um atlas digital baseado em realidade virtual para ensino de medicina**. 2006.
- MARESCAUX, Jacques et al. **Virtual reality applied to hepatic surgery simulation: the next revolution**. *Annals of surgery*, v. 228, n. 5, p. 627, 1998.
- DESPAIN, Wendy. **100 principles of game design**. New Riders, 2013.
- BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **UML: guia do usuário**. Elsevier Brasil, 2006.
- LARMAN, Craig. **Utilizando UML e padrões**. Bookman Editora, 2000.
- MATIAS, Roque; FRATEZZI, André. **Física Geral Para o Ensino Médio - Volume Único**. Harbra, 2010
- BAKKEN, David. **Middleware**. *Encyclopedia of Distributed Computing*, v. 11, 2001.
- SCHERER, Daniel; BATISTA, Daniele Ventura; DE CANTALICE MENDES, Aline. **Análise da Evolução de Engines de Jogos**. In: Anais do V Congresso sobre Tecnologias na Educação. SBC, 2020. p. 425-434.
- MONTEIRO, Roberto Luiz Souza. **GuaraScript: Projeto e Implementação de uma Linguagem de Programação “Open Source” Orientada a Computação Científica**. 2005.
- Souza Monteiro – Roberto Luiz Souza Monteiro, Ph.D.** Disponível em: <<http://www.guarascript.org/wordpress/>>. Acesso em: 20 de jun. de 2020.
- Motor de jogo**. Disponível em: <https://pt.wikipedia.org/wiki/Motor_de_jogo>. Acesso em: 20 de jun. de 2020.
- BOYER, Brandon. Serious Game Engine Shootout**. Gamasutra, 2007. Disponível em:<https://www.gamasutra.com/view/news/12772/SGS_Feature_Serious_Game_Engine_Shootout.php> Acesso em: 20 de jun. de 2020
- US Video Game Industry Revenue Reaches \$36 Billion in 2017**. Business Wire, 2018. Disponível em: <<https://www.businesswire.com/news/home/20180118006511/en/US-Video-Game-Industry-Revenue-Reaches-36-Billion-in-2017>> Acesso em: 20 de jun. de 2020

Video games starting to get serious. GAZETTE, 2011. Disponível em: <http://ww2.gazette.net/stories/083107/businew11739_32356.shtml> Acesso em: 20 de jun. de 2020

Gamemaker | Yoyo Games. Disponível em: <<https://www.yoyogames.com/gamemaker>>. Acesso em: 20 de jun. de 2020.

Plataforma de desenvolvimento em tempo real do Unity. Disponível em: <<https://unity.com/pt>>. Acesso em: 20 de jun. de 2020.

Unity. Disponível em: <<https://pt.wikipedia.org/wiki/Unity>>. Acesso em: 20 de jun. de 2020.

OGRE - Open Source 3D Graphics Engine | Home of a marvelous rendering engine. Disponível em: <<https://www.ogre3d.org/>>. Acesso em: 20 de jun. de 2020.

OGRE. Disponível em: <<https://pt.wikipedia.org/wiki/OGRE>>. Acesso em: 20 de jun. de 2020.

The most powerful real-time 3D creation platform - Unreal Engine. Disponível em: <<https://www.unrealengine.com/en-US/>>. Acesso em: 20 de jun. de 2020.

Unreal Engine. Disponível em: <https://pt.wikipedia.org/wiki/Unreal_Engine>. Acesso em: 20 de jun. de 2020.

Computer Science for modern software engineers | by Shalitha Suranga | softwareengineer.lk | Medium. Disponível em: <<https://medium.com/softwareengineer-lk/computer-science-for-modern-software-engineers-614c09e8345e>>

Marmalade (software). Disponível em: <[https://en.wikipedia.org/wiki/Marmalade_\(software\)](https://en.wikipedia.org/wiki/Marmalade_(software))>. Acesso em: 20 de jun. de 2020.

Marmalade makes it easy to create games that run cross-platform - Windows Developer Blog. Disponível em: <<https://blogs.windows.com/windowsdeveloper/2016/04/12/marmalade-makes-it-easy-to-create-games-that-run-cross-platform/>>

ShiVa Engine – Cross-platform Game Engine and IDE. Disponível em: <<https://shiva-engine.com/>>. Acesso em: 20 de jun. de 2020.

Havok – Technology for games. Disponível em: <<https://www.havok.com/>>. Acesso em: 20 de jun. de 2020.

Havok. Disponível em: <<https://pt.wikipedia.org/wiki/Havok>>. Acesso em: 20 de jun. de 2020.

Blender Game Engine. Disponível em: <https://pt.wikipedia.org/wiki/Blender_Game_Engine>. Acesso em: 20 de jun. de 2020.

Blender Game Engine: Simple Character – YouTube. Disponível em: <<https://www.youtube.com/watch?v=DBdpIDGXEPU>>

Bullet Real-Time Physics Simulation | Home of Bullet and PyBullet: physics simulation for games, visual effects, robotics and reinforcement learning. Disponível em: < <https://pybullet.org/wordpress/>>. Acesso em: 20 de jun. de 2020.

Bullet (software). Disponível em: < [https://en.wikipedia.org/wiki/Bullet_\(software\)](https://en.wikipedia.org/wiki/Bullet_(software))>. Acesso em: 20 de jun. de 2020.

Newton Dynamics • About Newton. Disponível em: < <http://newtondynamics.com/forum/newton.php> >. Acesso em: 20 de jun. de 2020.

Newton Dynamics 3.14 articulated basic vehicle – YouTube. Disponível em: <<https://www.youtube.com/watch?v=00LJPT7-RjA>>

Integration – Newton Wiki Disponível em: <<https://newtondynamics.com/wiki/index.php/Integration>>. Acesso em: 20 de jun. de 2020.

Newton Game Dynamics. Disponível em: < https://en.wikipedia.org/wiki/Newton_Game_Dynamics >. Acesso em: 20 de jun. de 2020.

GitHub - chriscamacho/bulletCapi: C API for Bullet Physics. Disponível em: < <https://github.com/chriscamacho/bulletCapi>>. Acesso em: 20 de jun. de 2020.

APÊNDICE A – Código fonte da aplicação Hinge2Vehicle.cpp do Bullet Physics Example Browser

```
#include "Hinge2Vehicle.h"

#include "btBulletDynamicsCommon.h"
#include "BulletCollision/CollisionShapes/btHeightfieldTerrainShape.h"

#include "BulletDynamics/MLCPSolvers/btDantzigSolver.h"
#include "BulletDynamics/MLCPSolvers/btSolveProjectedGaussSeidel.h"
#include "BulletDynamics/MLCPSolvers/btMLCPSolver.h"

class btVehicleTuning;

class btCollisionShape;

#include "BulletDynamics/ConstraintSolver/btHingeConstraint.h"
#include "BulletDynamics/ConstraintSolver/btSliderConstraint.h"

#include "../CommonInterfaces/CommonExampleInterface.h"
#include "LinearMath/btAlignedObjectArray.h"
#include "btBulletCollisionCommon.h"
#include "../CommonInterfaces/CommonGUIHelperInterface.h"
#include "../CommonInterfaces/CommonRenderInterface.h"
#include "../CommonInterfaces/CommonWindowInterface.h"
#include "../CommonInterfaces/CommonGraphicsAppInterface.h"

#include "../CommonInterfaces/CommonRigidBodyBase.h"

class Hinge2Vehicle : public CommonRigidBodyBase
{
public:
    /* extra stuff*/
    btVector3 m_cameraPosition;

    btRigidBody* m_carChassis;
    btRigidBody* localCreateRigidBody(btScalar mass, const btTransform& worldTransform,
    btCollisionShape* colShape);
};
```

```
GUIHelperInterface* m_guiHelper;
int m_wheelInstances[4];

bool m_useDefaultCamera;
//-----

class btTriangleIndexVertexArray* m_indexVertexArrays;

btVector3* m_vertices;

btCollisionShape* m_wheelShape;

float m_cameraHeight;

float m_minCameraDistance;
float m_maxCameraDistance;

Hinge2Vehicle(struct GUIHelperInterface* helper);

virtual ~Hinge2Vehicle();

virtual void stepSimulation(float deltaTime);

virtual void resetForklift();

virtual void clientResetScene();

virtual void displayCallback();

virtual void specialKeyboard(int key, int x, int y);

virtual void specialKeyboardUp(int key, int x, int y);

virtual bool keyboardCallback(int key, int state);

virtual void renderScene();

virtual void physicsDebugDraw(int debugFlags);
```

```

void initPhysics();
void exitPhysics();

virtual void resetCamera()
{
float dist = 8;
float pitch = -32;
float yaw = -45;
float targetPos[3] = {0,0,2};
m_guiHelper->resetCamera(dist, yaw, pitch, targetPos[0], targetPos[1], targetPos[2]);
}

/*static DemoApplication* Create()
{
Hinge2Vehicle* demo = new Hinge2Vehicle();
demo->myinit();
demo->initPhysics();
return demo;
}
*/
};

static btScalar maxMotorImpulse = 4000.f;

#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

#ifndef M_PI_2
#define M_PI_2 1.57079632679489661923
#endif

#ifndef M_PI_4
#define M_PI_4 0.785398163397448309616
#endif

//static int rightIndex = 0;

```



```

//static int upIndex = 1;
//static int forwardIndex = 2;
static btVector3 wheelDirectionCS(0, -1, 0);
static btVector3 wheelAxleCS(-1, 0, 0);

static bool useMCLPSolver = false; //true;

#include <stdio.h> //printf debugging

#include "Hinge2Vehicle.h"

//static const int maxProxies = 32766;
//static const int maxOverlap = 65535;

static float gEngineForce = 0.f;

static float defaultBreakingForce = 10.f;
static float gBreakingForce = 100.f;

static float maxEngineForce = 1000.f; //this should be engine/velocity dependent
//static floatmaxBreakingForce = 100.f;

static float gVehicleSteering = 0.f;
static float steeringIncrement = 0.04f;
static float steeringClamp = 0.3f;
static float wheelRadius = 0.5f;
static float wheelWidth = 0.4f;
//static floatwheelFriction = 1000;//BT_LARGE_FLOAT;
//static floatsuspensionStiffness = 20.f;
//static floatsuspensionDamping = 2.3f;
//static floatsuspensionCompression = 4.4f;
//static floatrollInfluence = 0.1f;//1.0f;

//static btScalar suspensionRestLength(0.6);

#define CUBE_HALF_EXTENTS 1

////////////////////////////////////

```

```

Hinge2Vehicle::Hinge2Vehicle(struct GUIHelperInterface* helper)
    : CommonRigidBodyBase(helper),
      m_carChassis(0),
      m_guiHelper(helper),
      m_indexVertexArrays(0),
      m_vertices(0),
      m_cameraHeight(4.f),
      m_minCameraDistance(3.f),
      m_maxCameraDistance(10.f)
{
    helper->setUpAxis(1);

    m_wheelShape = 0;
    m_cameraPosition = btVector3(30, 30, 30);
    m_useDefaultCamera = false;
}

void Hinge2Vehicle::exitPhysics()
{
    //cleanup in the reverse order of creation/initialization

    //remove the rigidbodies from the dynamics world and delete them
    int i;
    for (i = m_dynamicsWorld->getNumCollisionObjects() - 1; i >= 0; i--)
    {
        btCollisionObject* obj = m_dynamicsWorld->getCollisionObjectArray()[i];
        btRigidBody* body = btRigidBody::upcast(obj);
        if (body && body->getMotionState())
        {
            while (body->getNumConstraintRefs())
            {
                btTypedConstraint* constraint = body->getConstraintRef(0);
                m_dynamicsWorld->removeConstraint(constraint);
                delete constraint;
            }
            delete body->getMotionState();
            m_dynamicsWorld->removeRigidBody(body);
        }
        else

```

```

    {
    m_dynamicsWorld->removeCollisionObject(obj);
    }
    delete obj;
    }

    //delete collision shapes
    for (int j = 0; j < m_collisionShapes.size(); j++)
    {
    btCollisionShape* shape = m_collisionShapes[j];
    delete shape;
    }
    m_collisionShapes.clear();

    delete m_indexVertexArrays;
    delete m_vertices;

    //delete dynamics world
    delete m_dynamicsWorld;
    m_dynamicsWorld = 0;

    delete m_wheelShape;
    m_wheelShape = 0;

    //delete solver
    delete m_solver;
    m_solver = 0;

    //delete broadphase
    delete m_broadphase;
    m_broadphase = 0;

    //delete dispatcher
    delete m_dispatcher;
    m_dispatcher = 0;

    delete m_collisionConfiguration;
    m_collisionConfiguration = 0;
}

```

```

Hinge2Vehicle::~Hinge2Vehicle()
{
    //exitPhysics();
}

void Hinge2Vehicle::initPhysics()
{
    m_guiHelper->setUpAxis(1);

    btCollisionShape* groundShape = new btBoxShape(btVector3(50, 3, 50));
    m_collisionShapes.push_back(groundShape);
    m_collisionConfiguration = new btDefaultCollisionConfiguration();
    m_dispatcher = new btCollisionDispatcher(m_collisionConfiguration);
    btVector3 worldMin(-1000, -1000, -1000);
    btVector3 worldMax(1000, 1000, 1000);
    m_broadphase = new btAxisSweep3(worldMin, worldMax);
    if (useMCLPSolver)
    {
        btDantzigSolver* mlcp = new btDantzigSolver();
        //btSolveProjectedGaussSeidel* mlcp = new btSolveProjectedGaussSeidel;
        btMLCPSolver* sol = new btMLCPSolver(mlcp);
        m_solver = sol;
    }
    else
    {
        m_solver = new btSequentialImpulseConstraintSolver();
    }
    m_dynamicsWorld = new btDiscreteDynamicsWorld(m_dispatcher, m_broadphase, m_solver,
m_collisionConfiguration);
    if (useMCLPSolver)
    {
        m_dynamicsWorld->getSolverInfo().m_minimumSolverBatchSize = 1; //for direct solver it is
better to have a small A matrix
    }
    else
    {
        m_dynamicsWorld->getSolverInfo().m_minimumSolverBatchSize = 128; //for direct solver, it
is better to solve multiple objects together, small batches have high overhead

```

```

}
m_dynamicsWorld->getSolverInfo().m_numIterations = 100;
m_guiHelper->createPhysicsDebugDrawer(m_dynamicsWorld);

//m_dynamicsWorld->setGravity(btVector3(0,0,0));
btTransform tr;
tr.setIdentity();
tr.setOrigin(btVector3(0, -3, 0));

//either use heightfield or triangle mesh

//create ground object
localCreateRigidBody(0, tr, groundShape);

btCollisionShape* chassisShape = new btBoxShape(btVector3(1.f, 0.5f, 2.f));
m_collisionShapes.push_back(chassisShape);

btCompoundShape* compound = new btCompoundShape();
m_collisionShapes.push_back(compound);
btTransform localTrans;
localTrans.setIdentity();
//localTrans effectively shifts the center of mass with respect to the chassis
localTrans.setOrigin(btVector3(0, 1, 0));

compound->addChildShape(localTrans, chassisShape);

{
btCollisionShape* suppShape = new btBoxShape(btVector3(0.5f, 0.1f, 0.5f));
btTransform suppLocalTrans;
suppLocalTrans.setIdentity();
//localTrans effectively shifts the center of mass with respect to the chassis
suppLocalTrans.setOrigin(btVector3(0, 1.0, 2.5));
compound->addChildShape(suppLocalTrans, suppShape);
}

const btScalar FALLHEIGHT = 5;
tr.setOrigin(btVector3(0, FALLHEIGHT, 0));

const btScalar chassisMass = 2.0f;

```

```

const btScalar wheelMass = 1.0f;
m_carChassis = localCreateRigidBody(chassisMass, tr, compound); //chassisShape);
//m_carChassis->setDamping(0.2,0.2);

//m_wheelShape = new
btCylinderShapeX(btVector3(wheelWidth,wheelRadius,wheelRadius));
m_wheelShape = new btCylinderShapeX(btVector3(wheelWidth, wheelRadius,
wheelRadius));

btVector3 wheelPos[4] = {
btVector3(btScalar(-1.), btScalar(FALLHEIGHT-0.25), btScalar(1.25)),
btVector3(btScalar(1.), btScalar(FALLHEIGHT-0.25), btScalar(1.25)),
btVector3(btScalar(1.), btScalar(FALLHEIGHT-0.25), btScalar(-1.25)),
btVector3(btScalar(-1.), btScalar(FALLHEIGHT-0.25), btScalar(-1.25))};

for (int i = 0; i < 4; i++)
{
// create a Hinge2 joint
// create two rigid bodies
// static bodyA (parent) on top:

btRigidBody* pBodyA = this->m_carChassis;
pBodyA->setActivationState(DISABLE_DEACTIVATION);
// dynamic bodyB (child) below it :
btTransform tr;
tr.setIdentity();
tr.setOrigin(wheelPos[i]);

btRigidBody* pBodyB = createRigidBody(wheelMass, tr, m_wheelShape);
pBodyB->setFriction(1110);
pBodyB->setActivationState(DISABLE_DEACTIVATION);
// add some data to build constraint frames
btVector3 parentAxis(0.f, 1.f, 0.f);
btVector3 childAxis(1.f, 0.f, 0.f);
btVector3 anchor = tr.getOrigin();
btHinge2Constraint* pHinge2 = new btHinge2Constraint(*pBodyA, *pBodyB, anchor,
parentAxis, childAxis);

//m_guiHelper->get2dCanvasInterface();

```

```

//pHinge2->setLowerLimit(-SIMD_HALF_PI * 0.5f);
//pHinge2->setUpperLimit(SIMD_HALF_PI * 0.5f);

// add constraint to world
m_dynamicsWorld->addConstraint(pHinge2, true);

// Drive engine.
pHinge2->enableMotor(3, true);
pHinge2->setMaxMotorForce(3, 1000);
pHinge2->setTargetVelocity(3, 0);

// Steering engine.
pHinge2->enableMotor(5, true);
pHinge2->setMaxMotorForce(5, 1000);
pHinge2->setTargetVelocity(5, 0);

pHinge2->setParam( BT_CONSTRAINT_CFM, 0.15f, 2 );
pHinge2->setParam( BT_CONSTRAINT_ERP, 0.35f, 2 );

pHinge2->setDamping( 2, 2.0 );
pHinge2->setStiffness( 2, 40.0 );

pHinge2->setDbgDrawSize(btScalar(5.f));
}

resetForklift();

m_guiHelper->autogenerateGraphicsObjects(m_dynamicsWorld);
}

void Hinge2Vehicle::physicsDebugDraw(int debugFlags)
{
    if (m_dynamicsWorld && m_dynamicsWorld->getDebugDrawer())
    {
        m_dynamicsWorld->getDebugDrawer()->setDebugMode(debugFlags);
        m_dynamicsWorld->debugDrawWorld();
    }
}

```

```

//to be implemented by the demo
void Hinge2Vehicle::renderScene()
{
    m_guiHelper->syncPhysicsToGraphics(m_dynamicsWorld);

    m_guiHelper->render(m_dynamicsWorld);

    btVector3 wheelColor(1, 0, 0);

    btVector3 worldBoundsMin, worldBoundsMax;
    getDynamicsWorld()->getBroadphase()->getBroadphaseAabb(worldBoundsMin,
worldBoundsMax);
}

void Hinge2Vehicle::stepSimulation(float deltaTime)
{
    float dt = deltaTime;

    if (m_dynamicsWorld)
    {
        //during idle mode, just run 1 simulation step maximum
        int maxSimSubSteps = 2;

        int numSimSteps;
        numSimSteps = m_dynamicsWorld->stepSimulation(dt, maxSimSubSteps);

        if (m_dynamicsWorld->getConstraintSolver()->getSolverType() == BT_MLCP_SOLVER)
        {
            btMLCPSolver* sol = (btMLCPSolver*)m_dynamicsWorld->getConstraintSolver();
            int numFallbacks = sol->getNumFallbacks();
            if (numFallbacks)
            {
                static int totalFailures = 0;
                totalFailures += numFallbacks;
                printf("MLCP solver failed %d times, falling back to btSequentialImpulseSolver (SI)\n",
totalFailures);
            }
            sol->setNumFallbacks(0);
        }
    }
}

```



```

    }

    //#define VERBOSE_FEEDBACK
    #ifdef VERBOSE_FEEDBACK
        if (!numSimSteps)
            printf("Interpolated transforms\n");
        else
        {
            if (numSimSteps > maxSimSubSteps)
            {
                //detect dropping frames
                printf("Dropped (%i) simulation steps out of %i\n", numSimSteps - maxSimSubSteps,
numSimSteps);
            }
            else
            {
                printf("Simulated (%i) steps\n", numSimSteps);
            }
        }
    #endif //VERBOSE_FEEDBACK
    }

void Hinge2Vehicle::displayCallback(void)
{
    //glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    //renderme();

    //optional but useful: debug drawing
    if (m_dynamicsWorld)
        m_dynamicsWorld->debugDrawWorld();

    //glFlush();
    //glutSwapBuffers();
}

void Hinge2Vehicle::clientResetScene()
{

```

```

        exitPhysics();
        initPhysics();
    }

void Hinge2Vehicle::resetForklift()
{
    gVehicleSteering = 0.f;
    gBreakingForce = defaultBreakingForce;
    gEngineForce = 0.f;

    m_carChassis->setCenterOfMassTransform(btTransform::getIdentity());
    m_carChassis->setLinearVelocity(btVector3(0, 0, 0));
    m_carChassis->setAngularVelocity(btVector3(0, 0, 0));
    m_dynamicsWorld->getBroadphase()->getOverlappingPairCache()-
>cleanProxyFromPairs(m_carChassis->getBroadphaseHandle(),          getDynamicsWorld()-
>getDispatcher());
}

bool Hinge2Vehicle::keyboardCallback(int key, int state)
{
    bool handled = false;
    bool isShiftPressed = m_guiHelper->getAppInterface()->m_window-
>isModifierKeyPressed(B3G_SHIFT);

    if (state)
    {
        if (isShiftPressed)
        {
        }
        else
        {
            switch (key)
            {
            case B3G_LEFT_ARROW:
            {
                handled = true;
                gVehicleSteering += steeringIncrement;
                if (gVehicleSteering > steeringClamp)
                    gVehicleSteering = steeringClamp;
            }
            }
        }
    }
}

```

```

break;
}
case B3G_RIGHT_ARROW:
{
handled = true;
gVehicleSteering -= steeringIncrement;
if (gVehicleSteering < -steeringClamp)
gVehicleSteering = -steeringClamp;

break;
}
case B3G_UP_ARROW:
{
handled = true;
gEngineForce = maxEngineForce;
gBreakingForce = 0.f;
break;
}
case B3G_DOWN_ARROW:
{
handled = true;
gEngineForce = -maxEngineForce;
gBreakingForce = 0.f;
break;
}

case B3G_F7:
{
handled = true;
btDiscreteDynamicsWorld* world = (btDiscreteDynamicsWorld*)m_dynamicsWorld;
world->setLatencyMotionStateInterpolation(!world->getLatencyMotionStateInterpolation());
printf("world      latencyMotionStateInterpolation      =      %d\n",      world-
>getLatencyMotionStateInterpolation());
break;
}
case B3G_F6:
{
handled = true;

```

```

//switch solver (needs demo restart)
useMCLPSolver = !useMCLPSolver;
printf("switching to useMLCPSolver = %d\n", useMCLPSolver);

delete m_solver;
if (useMCLPSolver)
{
btDantzigSolver* mlcp = new btDantzigSolver();
//btSolveProjectedGaussSeidel* mlcp = new btSolveProjectedGaussSeidel;
btMLCPSolver* sol = new btMLCPSolver(mlcp);
m_solver = sol;
}
else
{
m_solver = new btSequentialImpulseConstraintSolver();
}

m_dynamicsWorld->setConstraintSolver(m_solver);

//exitPhysics();
//initPhysics();
break;
}

case B3G_F5:
handled = true;
m_useDefaultCamera = !m_useDefaultCamera;
break;
default:
break;
}
}
}
else
{
}
return handled;
}
}

```

```

void Hinge2Vehicle::specialKeyboardUp(int key, int x, int y)
{
}

```

```

void Hinge2Vehicle::specialKeyboard(int key, int x, int y)
{
}

```

```

btRigidBody* Hinge2Vehicle::localCreateRigidBody(btScalar mass, const btTransform& startTransform,
btCollisionShape* shape)

```

```

{
    btAssert(!shape || shape->getShapeType() != INVALID_SHAPE_PROXYTYPE);

    //rigidbody is dynamic if and only if mass is non zero, otherwise static
    bool isDynamic = (mass != 0.f);

    btVector3 localInertia(0, 0, 0);
    if (isDynamic)
        shape->calculateLocalInertia(mass, localInertia);

    //using motionstate is recommended, it provides interpolation capabilities, and only
synchronizes 'active' objects

#define USE_MOTIONSTATE 1
#ifdef USE_MOTIONSTATE
    btDefaultMotionState* myMotionState = new btDefaultMotionState(startTransform);

    btRigidBody::btRigidBodyConstructionInfo cInfo(mass, myMotionState, shape, localInertia);

    btRigidBody* body = new btRigidBody(cInfo);
    //body->setContactProcessingThreshold(m_defaultContactProcessingThreshold);

#else
    btRigidBody* body = new btRigidBody(mass, 0, shape, localInertia);
    body->setWorldTransform(startTransform);
#endif //

    m_dynamicsWorld->addRigidBody(body);
    return body;
}

```

```
}
```

```
CommonExampleInterface* Hinge2VehicleCreateFunc(struct CommonExampleOptions& options)
```

```
{
```

```
    return new Hinge2Vehicle(options.m_guiHelper);
```

```
}
```