CENTRO PAULA SOUZA

FACULDADE DE TECNOLOGIA DE AMERICANA
Curso Superior de Jogos Digitais

Victor Alcântara Ferreira

CRIAÇÃO DE UMA *ENGINE* 3D E UM PROTÓTIPO DE *GAME*

Americana, SP
2015

CENTRO PAULA SOUZA

FACULDADE DE TECNOLOGIA DE AMERICANA
Curso Superior de Jogos Digitais

Victor Alcântara Ferreira

CRIAÇÃO DE UMA *ENGINE* 3D E UM PROTÓTIPO DE *GAME*

Trabalho Monográfico, desenvolvido em cumprimento à exigência curricular do Curso Superior de Tecnologia em Jogos Digitais da Fatec-Americana, sob orientação do Prof. Kléber Andrade

Área: Jogos Digitais

Americana, SP
2015

FICHA CATALOGRÁFICA – Biblioteca Fatec Americana - CEETEPS
Dados Internacionais de Catalogação-na-fonte

Ferreira, Victor Alcantâra
F444c Criação de uma engine 3D e um protótipo de game. / Victor Alcântara Ferreira. – Americana: 2015.
65f.

Monografia (Graduação em Tecnologia em Jogos Digitais). - - Faculdade de Tecnologia de Americana – Centro Estadual de Educação Tecnológica Paula Souza.
Orientador: Prof. Me. Kleber de Oliveira Andrade

1. Jogos digitais I. Andrade, Kleber de Oliveira II. Centro Estadual de Educação Tecnológica Paula Souza – Faculdade de Tecnologia de Americana.

CDU: 681.6

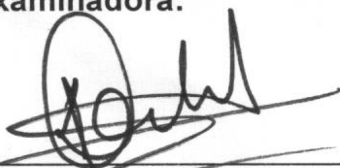
Victor Alcântara Ferreira

CRIAÇÃO DE UMA *ENGINE* 3D E UM PROTÓTIPO DE *GAME*

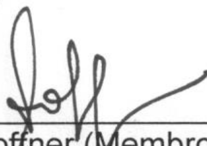
Trabalho de graduação apresentado como exigência parcial para obtenção do título de Tecnólogo em Jogos Digitais pelo CEETEPS/Faculdade de Tecnologia – FATEC/ Americana.
Área de concentração: Jogos Digitais

Americana,10 de dezembro de 2015.

Banca Examinadora:



Kleber de Oliveira Andrade (Presidente)
Mestre
Fatec Americana



Renato Kraide Soffner (Membro)
Doutor
Fatec Americana

Luiz Carlos Caetano (Membro)
Especialista
Fatec Americana

RESUMO

Este trabalho busca verificar se na situação atual do mercado *engines* de jogos ainda é possível uma pequena equipe ou apenas uma pessoa criar sua própria sem bibliotecas de terceiros (com exceção da API gráfica e do sistema operacional) e descrever as principais diferenças desta *engine* com as comerciais.

A fim de observar os tópicos propostos, uma *engine* foi criada e um protótipo de jogo desenvolvido, com isso concluiu-se que uma *engine* 3D feita “do zero” possui espaço no mercado, seja criando seus jogos (limitados, como os para dispositivos móveis) ou sendo licenciada para desenvolvedores necessitando de um sistema compacto, de fácil entendimento e manutenção, alta performance, e com soluções para problemas específicos

Palavras Chave: engine de jogo; computação gráfica; programação de jogos

ABSTRACT

This paper attempts to verify if in the current game engine market still is possible a small team or single developer creating their own game engine without third-party libraries (with exception of graphics and operating system API) and describe the diferences among them.

To observe those differences, an engine has been created and a game prototype designed, coming to the conclusion that a 3D engine “from scratch” has space on the Market by creating its own games (although limited, ideal for mobile devices) or by licensing to developers who need a compact system, easy of understanding and maintaining, high performance and offering solutions to specific problems.

Keywords: game engine; computer graphics; game programming

SUMÁRIO

INTRODUÇÃO 11

GAME ENGINE (VISÃO GERAL DO QUE EXISTE) 16

1.1 INTERFACES 18
1.2 INTERFACE COM HARDWARE, APIS E SISTEMA OPERACIONAL 18
1.3 GERENCIAMENTO DE RECURSOS 20
1.4 SISTEMA DE ENTIDADE 21
1.5 SISTEMA DE EVENTO 24
1.6 INPUT 25
1.7 PARTICIONAMENTO DO ESPAÇO 26
1.8 COLISÃO 28
1.9 RENDERIZAÇÃO 29
1.10 ANIMAÇÃO 30
1.11 GERENCIAMENTO DE MEMÓRIA 32
1.12 INTELIGÊNCIA ARTIFICIAL 33

DESENVOLVIMENTO DA ENGINE 36

1.13 VISÃO GERAL 36
1.14 FERRAMENTAS 38
1.15 DETALHES TÉCNICOS DA ENGINE 39
1.16 MACROS, FUNÇÕES, REDEFINIÇÃO DE TIPOS E RESPOSTA A ERROS .. 40
1.17 ALGORÍTMOS E ESTRUTURAS DE DADOS 43
1.18 GERENCIAMENTO DE MEMÓRIA 45
1.19 MATEMÁTICA 46
1.20 GERENCIADOR DE MODELOS (MESHES) 47
1.21 RENDERIZADOR, SHADERS E EFEITOS 49
1.22 GERENCIADOR DE CENA 50
1.23 ENTIDADES, COMPONENTES E SEUS GERENCIADORES 52
1.24 GERENCIADOR DE EVENTOS 54
1.25 SISTEMA DE RECURSO 55
1.26 CENTRALIZAÇÃO DOS MÓDULOS NA CLASSE CENGINE 55

PROTÓTIPO DO JOGO “QUEM MEXEU NO MEU *SHADER*?” 57

CONCLUSÕES 60

REFERÊNCIAS BIBLIOGRÁFICAS 61

LISTA DE FIGURAS E DE TABELAS

Figura 1 - Esquematização de sistemas de uma *engine*.....................................17

Figura 2 - Separação de camadas de uma *engine*..19

Figura 3 - UML de um sistema de gerenciamento de modelo 21

Figura 4 - Sistema de entidade por herança e polimorfismo..............................22

Figura 5 - Sistema de entidade por componente..23

Figura 6 - Associação de input do jogador com algum comando 25

Figura 7 - Particionamento binário do espaço..27

Figura 8 - Visualização da divisão espacial de um *octree*..................................28

Figura 9 - Associação de matrizes de transformação...30

Figura 10 - Falha de alocação em memória fragmentada...................................33

Figura 11 - Estados e transições de uma FSM de um inimigo 34

Figura 12. Principais subsistemas da *engine*..37

Figura 13 - *Debugger* do Visual Studio 2015 Community Edition.......................38

Figura 14 - *Debugger* de *Shader* 39

Figura 15 - Algumas definições de tipos (*typedefs*)..40

Figura 16 - Exemplo de macros em Util.h 41

Figura 17 - Trechos de código do *header* TextureUtil.h.......................................42

Figura 18 - Mensagem de erro e classe de exceção 43

Figura 19 - Estruturas de dados sendo utilizadas na *engine*.............................44

Figura 20 - Exemplo de ponteiros esperto e fraco 46

Figura 21 - Modelo sendo executado pela *engine*...47

Figura 22 - Modelo com informação de posição e normal sendo iluminado por um "sol"...48

Figura 23 - *Shaders* "pedindo" ao gerenciador de modelo uma *mesh* em seu formato...49

Figura 24 - Renderização de cena usando múltiplos passes 51

Figura 25 - Métodos da classe CComponent 52

Figura 26 - Gerenciador de componente processando os componentes do jogo 53

Figura 27 - Código demonstrando a criação de uma entidade 54

Figura 28 - Esquema do sistema de eventos 54

Figura 29 - Código demonstrando a implementação de um *listener*..................55

Figura 30 - Definição dos *sys_*..56

Figura 31 - Interface entre engine e aplicação..56

Figura 32 - Declaração da classe do jogador..58

Figura 33 - Declaração da classe do vírus 58

Figura 34 - Protótipo do jogo 59

INTRODUÇÃO

Embora a definição de *engine* não seja única (GREGORY, 2015) seu propósito é claro: oferecer um kit de desenvolvimento de jogos para desenvolvedores, usando as mais diversas tecnologias (como animação, áudio, inteligência artificial, renderização, etc.), várias *engines* comerciais (Unity 3D, Unreal, CryEngine, etc;) apresentam características em comum:

- **Implementação encapsulada** o programador não precisa se preocupar com detalhes do funcionamento da *engine* para fazê-la funcionar.

- **Suporte para extensão** é possível criar comportamentos, ferramentas personagens novos usando ou uma linguagem criada para ser interpretada pela própria *engine* (scripts) ou acessando o seu código fonte e modificando-o.

- **Editor** com ele o cenário do jogo é desenvolvido, junto com todos os sons, eventos e personagens presentes.

- **Bibliotecas** são diversos módulos com fórmulas e algoritmos especializados em solucionar um determinado problema, como bibliotecas de matemática, som, física, etc.

Engines comerciais disponíveis ao público oferecem vastos recursos e tecnologias para criação dos mais diversos jogos. Contudo, desenvolvedores ainda optam por criar suas próprias, pelos mais diversos motivos.

Empresas de desenvolvimento certamente possuem o conhecimento técnico e mão-de-obra para isso. Desenvolvedores independentes (*indies*) porém, não possuem os recursos financeiros e humanos para desenvolver suas próprias tecnologias – criando um possível **problema**. Diante deste fato, este trabalho busca responder a seguinte **pergunta**: “É possível uma pequena equipe (ou até mesmo uma pessoa) criar uma *engine* e desenvolver um jogo dela? ”

A **hipótese** é "uma *engine* 3D criada do zero ainda possui algum espaço no mercado e é capaz de criar um jogo? "

O **objetivo geral** deste trabalho é desenvolver uma *engine* e esquematizar (prototipar) um jogo usando ela.

Tendo por objetivos específicos:

- Criar uma *engine* com suporte a colisão, renderização, leitura de script, gerenciamento de recurso e entidade;
- Como em *engines* comerciais, isolar códigos-fontes dedicados a APIs e *hardware*, expondo uma interface comum ao programador do jogo;
- Criar o jogo "Quem mexeu no meu *shader*", demonstrando os principais recursos da *engine* e provando sua capacidade de simular um mundo virtual e suas regras.
- Verificar se *engines* de autoria própria podem ser desenvolvidas por uma equipe pequena e ainda assim serem capazes de apresentar um jogo;

As **justificativas** para a criação de uma *engine* própria quanto tantas outras estão disponíveis vão desde estratégia da equipe até a possibilidade de criar um novo produto visando lucro.

Quem opta por sua própria *engine* possui total controle da tecnologia e decide quais recursos de hardware serão explorados, em quais situações e de quais formas. Por ser uma necessidade específica, é possível assumir que esta *engine* possua facilidades e soluções para um certo problema não existente nas demais. Isso abre a possibilidade de comercializar esta *engine* para aqueles com as mesmas necessidades específicas.

Vale ressaltar que o novo produto pode não ter o polimento, clareza ou recursos de uma *engine* comercial genérica, mas por atender a uma necessidade específica não é uma concorrente direta das outras no mercado.

A **metodologia** deste projeto de pesquisa é um estudo exploratório, onde uma *engine* é desenvolvida e um jogo é prototipado.

Este trabalho possui as seguintes divisões:

- Introdução. Define o problema, pergunta, hipóteses, objetivos (geral e específico), justificativa e metodologia. O capítulo também contextualiza o trabalho, mostrando uma visão geral do que existe e definindo conceitos;
- Visão geral de uma *engine* de jogos. Busca entender o que é uma *engine* de jogos e explora de forma geral os principais sistemas, recursos e técnicas em *engines* comerciais;
- Desenvolvimento da *engine*. Dedicado a explicar de forma detalhada o funcionamento da *engine* e justificar as decisões tomadas resultadas em sua arquitetura.
- Desenvolvimento do protótipo de jogo: “Quem mexeu no meu *shader*?”. Explora o procedimento de criação do jogo e suas características, além de concluir e comentar o caso da hipótese que o trabalho conseguiu provar (se conseguiu).

No início da história do desenvolvimento de jogos, as equipes eram extremamente especializadas e necessitavam possuir grande conhecimento técnico para poderem criar o código-fonte de seus projetos, tornando este mercado extremamente restrito e de acesso limitado ao público geral. (KUSHNER, 2003).

O avanço das tecnologias de consoles e PCs permitiu maiores mundos, recursos e efeitos nos jogos. Muitos desses complexos de programar. Este fato aliado à facilidade oferecida pela internet de obter novos softwares e o sonho de pessoas que cresceram jogando criar seus próprios mundos virtuais – porém sem possuírem todo o conhecimento técnico para criar um jogo - abriram a possiblidade de um produto onde as principais (e mais complexas) tecnologias sejam apresentadas de forma amigável (interface gráfica, ao invés de código), sem a necessidade imediata de ter grande conhecimento de seu funcionamento.

Suprindo esta carência, empresas começaram a disponibilizar suas *engines* e ferramentas de desenvolvimento (*Software Development Kit*, SDK) para o público entusiasta criar seus próprios jogos. As ferramentas possuem as mais diversas complexidades e flexibilidade, além de modelos de negócio, como grátis, mensalidades, compra de licença, etc. (GREGORY, 2015).

Para desenvolver um game, equipes (ou até mesmo indivíduos) podem escolher diversos pontos de início, seja criar seu próprio motor (*engine*), usar um já existente, usar *frameworks*, etc.

Aqueles que optam por fazer sua própria *engine* precisam ter o conhecimento técnico para programá-la, além de ter de lidar com os diversos defeitos e mal funcionamento causados pelo código (*bugs*). Como já existem várias *engines* no mercado, talvez uma das existentes possa suprir todas as necessidades da equipe desenvolvendo sua *engine*. No entanto, *engines* comerciais podem possuir recursos demasiados e terem uma curva de aprendizagem maior do que uma tecnologia proprietária e mais específica. (MCSHAFFRY; GRAHAM, 2013).

Quem escolhe uma *engine* pronta (Como Unreal Engine da Epic, GameMaker da YoYoGames) tem a sua disposição ferramentas para a criação de jogos aptos para iniciar o desenvolvimento, sem precisar se preocupar em aprender todos os detalhes necessários de sua programação. Além disso, *engines* comerciais oferecem comunidades ativas, garantia maior de ausência de bugs (o produto é testado por milhares de desenvolvedores ao mesmo tempo).

Frameworks por sua vez, como o XNA da Microsoft e o projeto Monogame, são voltados para programadores e disponibilizam vários algoritmos, sistemas, *debuggers* (ferramentas para encontrar defeitos), para desenvolvedores criarem suas *engines* sem precisar preocupar-se com detalhes dos hardwares, sistemas operacionais, drivers onde a *engine* rodará (MONOGAME, 2015).

Por fim, APIs (*Application programming interface*), como DirectX da Microsoft ou o OpenGL, são limitadas em oferecer ao programador uma interface entre seu software e os diversos drivers e hardwares disponíveis para realizar determinada tarefa.

Engines de games existem nas mais diversas linguagens, diversos preços e licenças (apenas para uso interno de um estúdio, grátis, grátis até certo lucro, etc.). No entanto, no início da era do vídeo game, *engines* eram extremamente específicas, sendo basicamente uma construída para cada jogo (GREGORY, 2015).

Este trabalho busca entender se uma *engine* criada como as antigas, sem bibliotecas de suporte, sem *engines third-party*, limitadas ao jogo que visam produzir, ainda são capazes de criar ao menos um protótipo de jogo.

A linguagem de escolha é a C++, comum ao desenvolvimento de jogos tanto para PC como para consoles, além de ser a família de linguagens sempre presente no desenvolvimento de jogos, seja com clássicos antigos como Doom (C clássico) (KUSHNER, 2003) ou um dos melhores jogos já criados (METACRITIC, 2015), Half-life 2 (C++) (VALVE, 2004) ou os incontáveis jogos para mobile feitos na *engine* Unity usando scripts em C# (UNITY 3D, 2015).

Para comandar a placa de vídeo, é utilizada a API gráfica DirectX 11 da Microsoft. Ela oferece o poder de controlar os estados e operações da placa gráfica sem precisar conhecer os detalhes de seu driver, sendo assim permitindo ao jogo executar em uma gama de configurações de hardware, software e hardware. (GREGORY, 2015).

O DirectX força o uso da HLSL (*High Level Shading Language*), uma linguagem com sintaxe semelhante ao C executada na placa de vídeo, resultando em programas capazes de simular luz, sombras, reflexos, efeitos atmosféricos, efeitos de câmera ou até mesmo fazer cálculos de física (GAME INSTITUTE, 2015).

Os sistemas operacionais suportados pela *engine* são, no momento da escrita, Windows 7, Windows 8, Windows 8.1 e Windows 10. No entanto, visto a retrocompatibilidade sempre almejada a cada versão do Windows, não é absurdo afirmar que versões posteriores à 10 ainda darão suporte à engine mesmo com o desenvolvimento desta acabado (SPOLSKY, 2004).

Por fim, o trabalho de compilar e *debugar* e auxiliar na organização dos fontes do projeto cabe ao Visual Studio 2015 Community Edition (MICROSOFT, 2015).

GAME ENGINE (VISÃO GERAL DO QUE EXISTE)

Atualmente, as *engines* não são apenas um serviço fechado, feitas para empresas e equipes extremamente especializadas desenvolverem seus produtos (games). Elas próprias são o produto e por isso possuem uma interface amigável, documentos detalhados e comunidades ativas, prontas para ajudar desenvolvedores novatos e experientes em suas jornadas para criar seus jogos (EPIC, 2015) (CRYTEK, 2015) (UNITY 3D, 2015).

Se no passado *engines* licenciáveis possuíam preços exorbitantes para o cidadão comum e empresas pequenas (MCSHAFFRY; GRAHAM, 2013), hoje os mais diversos planos e licenças existem para qualquer pessoa poder iniciar no mundo de desenvolvimento de games. Planos como pagar a partir do momento que o jogo atingir certo lucro (Unreal Engine e CryEngine) ou ser totalmente grátis, no entanto com recursos limitados (Unity 3D) ou possui diversas versões para cada tipo de equipe (GameMaker).

Na parte do processo de desenvolvimento, a programação torna-se de alto nível, ou seja, o programador consegue através de poucas linhas e sem precisar conhecer o funcionamento de baixo nível de sua *engine*, design e algoritmos, criar suas rotinas (GREGORY, 2015). Para algumas *engines*, o programador não consegue modificar seu código, apenas criar scripts para ela. Outras, como a IdTech (Doom) e a Ogre 3D permitem total controle por parte do programador (MORDETH, 200-).

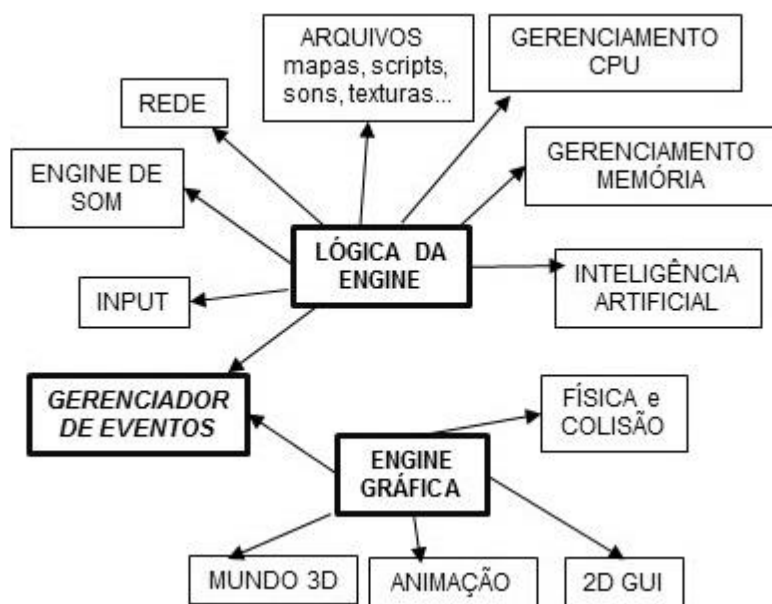
Quanto à parte artística, *engines* atuais facilitam artistas aplicarem aos seus cenários efeitos, cores e texturas. Mostrando-lhes em tempo real como suas ideias ficam no mundo do jogo, além de permitir visualizar/combinar animações, remixar sons, etc (EPIC, 2015).

Por fim, a comunidade criada oferece ideias, tutoriais, ajuda para aqueles tentando desenvolver em determinada *engine* (UNITY 3D, 2015). A própria comunidade pode começar a criar produtos como livros (AMRESH; OKITA, 2010) ou cursos online (GAMES INDIE, 2015) com o foco em ensinar a criação de jogos.

(GREGORY, 2015) embora a definição de *engine* não seja precisa, é possível afirmar que qualquer jogo lançado comercialmente possui sua *engine* ou partes de uma. *Engine* pode ser vista como o conjunto de bibliotecas (gráfico, IA, física, som, animação) e ferramentas utilizadas para o desenvolvimento de um jogo. Cada qual podendo ter um maior foco em programação (IdTech) ou em oferecer ferramentas para o desenvolvedor (GameMaker) ou ambas como a Source (VALVE, 2004).

Devido ao grande número de *engines*, focos (FPS, mundo aberto, RPG, filme interativo, etc.) e a algumas nunca ter o código fonte revelado, é no mínimo impreciso abstrair todos os conceitos, rotinas e sistemas para descrevê-la. Mas toda *engine* deve possuir sistemas básicos para garantir seu funcionamento. A figura 1 mostra alguns sistemas de uma *engine*, a qual parte pertencem (lógica ou gráfica) e o meio de comunicação entre eles.

Figura 1 - Esquematização de sistemas de uma *engine*



Fonte: http://www.felixgers.de/teaching/game/game-modules-talk/GameEngineAndModules.html (200-, tradução do autor)

1.1 INTERFACES

Interface refere a algum meio de isolar um sistema de outro. A necessidade e importância desta técnica é facilitar operações de mais alto nível serem executadas sem conhecer ou modificar o código em pontos não relevantes a tarefa em questão. Interfaces garantem o encapsulamento, pois reúne o código necessário para realizar uma tarefa e caso alguma mudança seja feita, espera apenas mudar os pontos de chamada da interface para a *engine* (ou seja, uma estrutura para outra), ao invés de alterar em todas as chamadas que o jogo faz à *engine* (ELBERY, 2007).

Em programação orientada a objetos, interface está relacionada a criação de uma classe abstrata e sua implementação é feita através da implementação dessas interfaces utilizando o polimorfismo. No entanto games, por serem aplicações de alta performance, podem evitar a sobrecarga de processamento e utilizar meios não orientados a objetos, como utilizar um *namespace*[1] e funções para centralizar a interface, criar a declaração de uma classe e diversas implementações controladas por macros (Doom 3 utiliza esta técnica para encapsular a biblioteca de contagem de tempo, por exemplo) (ID SOFTWARE, 2012).

1.2 INTERFACE COM HARDWARE, APIS E SISTEMA OPERACIONAL

Segundo Elbery (2007) e Gregory (2015), para realizar operações como:

- Criar ou carregar um *save*
- Tirar uma *screenshot*
- Encontrar algum arquivo
- Renderizar uma cena
- Carregar um mapa

[1] Isola o código através de um nome. Por exemplo, chamadas para a biblioteca de matemática devem iniciar com *Math* por causa de seu *namespace*

É preciso utilizar bibliotecas de terceiros, normalmente do fabricante, pois apenas eles sabem dos detalhes da implementação de seus hardwares e softwares.

Ao invés de chamar diretamente pelo código dessas bibliotecas, as interfaces são utilizadas para encapsular a comunicação externa da *engine*, oferecendo as vantagens e recursos vistos no tópico de interfaces.

Uma outra vantagem de utilizar interfaces é a capacidade de mudar as bibliotecas para diferentes sistema, por exemplo jogos com suporte para renderização por DirectX, OpenGL e software podem ser usados e por parte dos desenvolvedores não ser preciso mudar as chamadas para as APIs pois elas são feitas de forma indireta através da interface, como demostrado por McShaffry e Graham (2013).

A figura 2 resume a ideia de uma camada de interface separando o hardware da aplicação. As interfaces estão no segundo bloco de cima para baixo, separando sistema operacional e hardware (baixo nível) da aplicação (alto nível):

Figura 2 - Separação de camadas de uma *engine*

Fonte: http://www.heptomino.com/vge/ (201-, tradução do autor)

1.3 GERENCIAMENTO DE RECURSOS

Para muitos jogos e sistemas não basta a *engine* ser capaz de carregar texturas, sons, mapas, entre outros arquivos. É preciso gerenciar quando o arquivo é carregado, por quanto tempo dura em memória e durante seu tempo de vida em RAM de qual forma deve ser armazenado para extrair a maior desempenho (MCSHAFFRY; GRAHAM, 2013).

Esses cuidados são precisos pois há sistemas limitados como consoles e dispositivos mobile que simplesmente não são capazes de carregar todo o conteúdo do jogo de uma vez. Outro fator limitante é o tempo de carga, pois mesmo a memória tendo o espaço necessário o tempo de *loading* pode ser grande.

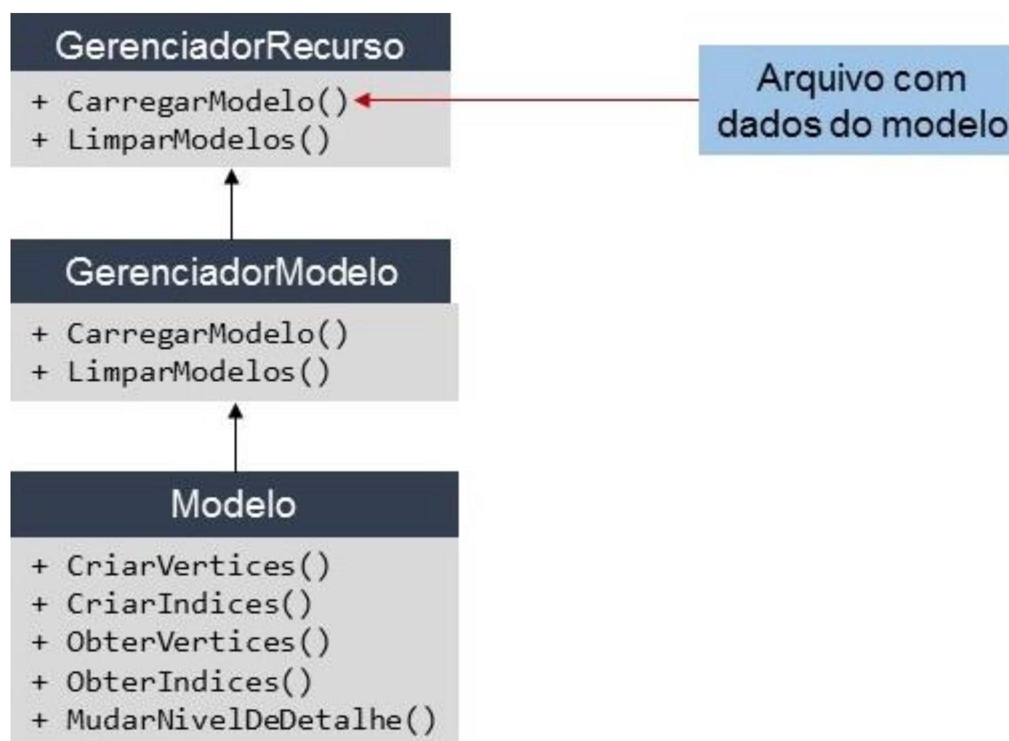
A necessidade de cada jogo cria uma forma de gerenciar recursos. Por exemplo, jogos de mundo aberto podem manter um ciclo de alocação-deslocação de recurso baseando-se em FIFO (*first in, first out*), no qual, usando como exemplo texturas, a última carregada será a última a ser descartada. A lógica por trás deste algoritmo é fazer com que texturas novas (ou seja, vistas recentemente pelo jogador) permaneçam por mais tempo, tornando improvável o sistema ter de recarrega-las enquanto o player estiver na área delas (MCSHAFFRY; GRAHAM, 2014).

Outra estrutura é a lista de prioridade (*priority queue*). Nela cada recurso possui um número definindo sua importância. A pontuação pode ser utilizada para situar o recurso em áreas de rápido acesso na memória e priorizar deslocação (GREGORY, 2015) e (MCSHAFFRY; GRAHAM, 2013).

A técnica mais genérica, portanto presente em praticamente qualquer jogo, é manter a relação de todos os recursos já em memória para caso o jogo precise deles a *engine* não tentará carregar novamente o arquivo, apena passará um ponteiro ou *handle* (identificação) dele. (CLEOPESCE, 2014)

A figura 3 é o UML de um pequeno sistema de gerenciamento de recursos de um modelo (*mesh*). Onde operações básicas de criação, carregamento e manipulação de vértices e índices são feitas.

Figura 3 - UML de um sistema de gerenciamento de modelo



Fonte: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter05.html (2005, tradução do autor)

1.4 SISTEMA DE ENTIDADE

A ideia de uma entidade em um jogo é representar tudo imaginável no universo do jogo. Desde inimigos, paredes e luzes, até triggers e opções de um menu.

A entidade permite à *engine* conhecer aquilo que habita o jogo e oferece uma estrutura capaz de tornar-se qualquer coisa para os programadores poderem criar os objetos do jogo.

Por parte da *engine*, as entidades devem possuir algumas características:

- Armazenamento em estruturas de rápido acesso. Como elas sempre serão requisitadas por todos os sistemas e várias vezes durante cada loop, o acesso precisa ser instantâneo. Árvores binárias balanceadas e *hash table* são opções viáveis.

- Serialização: para salvar o progresso do jogador ou gerar arquivos de mapa durante o desenvolvimento, as entidades precisar ser colocadas em um formato que pode ser gravado e posteriormente lido e traduzido pela *engine*.

- Identificação única, evitando o problema de realizar acidentalmente determinada operação com a entidade errada. Por exemplo: ao jogador matar um inimigo este deve desaparecer; sem identificação única talvez outro inimigo desapareça ou até mesmo um objeto não relacionado, como a arma do jogador.

Quanto aos programadores, segundo Gregory (2015) e Gaul (2013), basta a entidade ser polimórfica para tornar-se qualquer objeto, como uma arma, inimigo, parede, árvore, nuvem, etc. A figura 4 esquematiza a herança e criação de várias entidades mais específicas partindo de uma classe pai genérica.

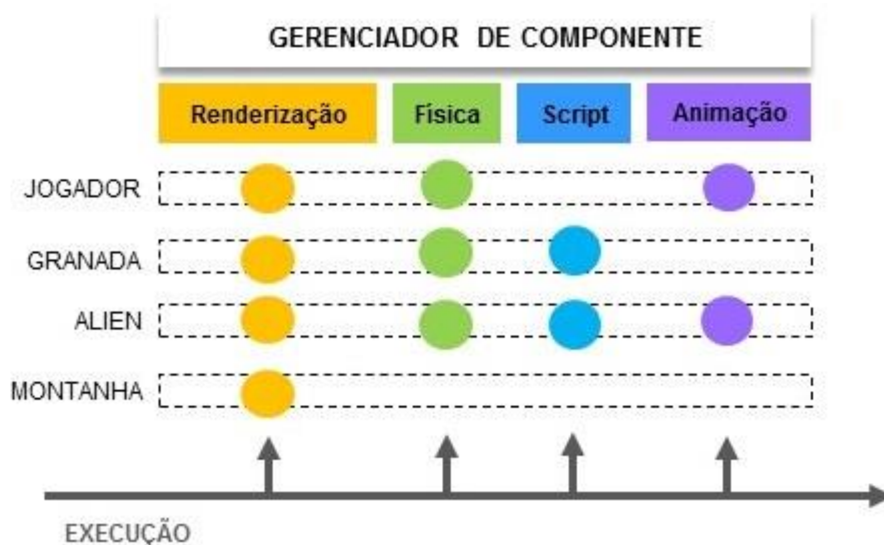
Figura 4 - Sistema de entidade por herança e polimorfismo

Fonte: http://www.radicalfishgames.com/?p=1725 (2014, tradução do autor)

Gaul (2013), McShaffry e Graham (2013) descrevem o sistema de entidades formado por componentes, utilizado em *engines* como a Unity 3D. Desta forma as diferentes operações de uma entidade ficam encapsuladas em seus componentes, por exemplo, o componente de IA quando colocado em uma entidade faz ela caminhar pelo cenário, enquanto o componente de renderização mostra a entidade. Caso algum deles seja desativado, algum comportamento deixará de existir.

A flexibilidade do sistema de componentes permite um maior controle e melhor leitura do código, além de amenizar o uso de herança. A figura 5 mostra cada entidade e seus componentes. É interessante observar que se uma entidade precisar ter um novo comportamento, como a montanha colidir com outros objetos, ela apenas precisa referenciar um componente (no caso do exemplo, de colisão).

Figura 5 - Sistema de entidade por componente



Fonte: http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/ (2007, tradução do autor)

1.5 SISTEMA DE EVENTO

O sistema de evento permite transitar mensagens entre sistemas da *engine* sem que eles saibam da existência do outro. Em linguagem técnica, sua função é reduzir acoplamento [9][34].

Para ilustrar, supondo que um jogo possua uma arma, quando seu tiro atinge uma placa de vidro ela deve rachar (mudar de textura) e emitir um som. Aqui temos, simplificando para a esquematização, os sistemas operando:

- Sistema de colisão: identifica onde (se) o tiro irá colidir e com quem
- Sistema de entidade: permite ao vidro responder ao evento de tiro
- Sistema de áudio: emite o som do vidro rachando

Todos esses sistemas são conhecidos pela entidade vidro e, portanto, podem ser acessados diretamente e funcionar.

No entanto caso algum desses sistemas mude ou seja removido (no caso, o único possível seria o de áudio) pode ser preciso ajustar todas as chamadas para esses sistemas.

Com eventos, é transitado entre os sistemas apenas um “recado” do que ocorreu e não é preciso acessar outros sistemas diretamente por código, diminuindo dependência e acoplamento.

Seu funcionamento pode ser divido em três partes:

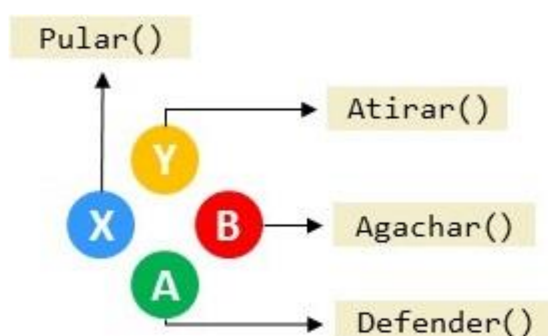
- Mensagem (ou evento): uma estrutura contendo um identificador do evento ocorrido, no caso, um identificador para o evento de a bala atingir o vidro.
- Gerenciador de mensagens: a interface possibilitando o envio de evento. No exemplo, seria o sistema detector de colisão que iria enviar uma mensagem contendo as informações da colisão ocorrida; pode-se pensar numa estrutura contendo os objetos da colisão (bala e vidro), o momento e o local da colisão.

- Receptor: a mensagem deve ser recebida pelas entidades, rotinas e sistemas, lidas e interpretadas. Para receber uma mensagem pode-se utilizar funções de call-back ou classes polimórficas com métodos para ler eventos. Estes call-backs ou classes são registradas ao gerenciador de eventos e executa-as quando um evento é acionado. No exemplo, a entidade “vidro” deve receber a mensagem de colisão entre ele e a bala e rachar (seja dando a responsabilidade para outro sistema ou chamando pelo seu próprio `Rachar()`).

1.6 INPUT

O principal problema resolvido pelo sistema de input é a separação entre os mais diversos dispositivos de entrada que o jogador possa utilizar e as ações do jogo. Mapear diretamente o pressionar de um botão do mouse a um tiro, por exemplo, cria um jogo absolutamente inflexível para configuração de controles e suporte para *gamepads*, *joysticks*, etc (NYSTROM, 201-). A figura 6 mostra a associação de um botão ou tecla do dispositivo usado pelo usuário e o comando dado ao jogo.

Figura 6 - Associação de input do jogador com algum comando



Fonte: http://gameprogrammingpatterns.com/command.html (201-, tradução do autor)

A alternativa é encapsular por meio de uma interface a comunicação entre dispositivos de entrada e sistema operacional e ter uma camada de ações entre o jogo e o sistema de input. Desta forma ações são registradas a certos botões (seja de mouse, teclado, *gamepad*, etc.), mas podem ser facilmente reassociados. Por parte do jogo, não deve verificar se uma tecla foi pressionada e sim se uma ação foi executada, desta forma o jogo torna-se indiferente aos dispositivos de entrada (GREGORY, 2015).

Outra vantagem é a mudança de plataforma. Como o jogo não faz ideia dos dispositivos do usuário, apenas de suas ações, é possível portar o jogo para mobile (*touch*), consoles (diferentes *gamepads*), notebooks (apenas teclado) sem precisar alterar o código do jogo, apenas o da parte baixa da interface que relaciona-se ao sistema operacional e ao dispositivo de entrada (NYSTROM, 201-).

1.7 PARTICIONAMENTO DO ESPAÇO

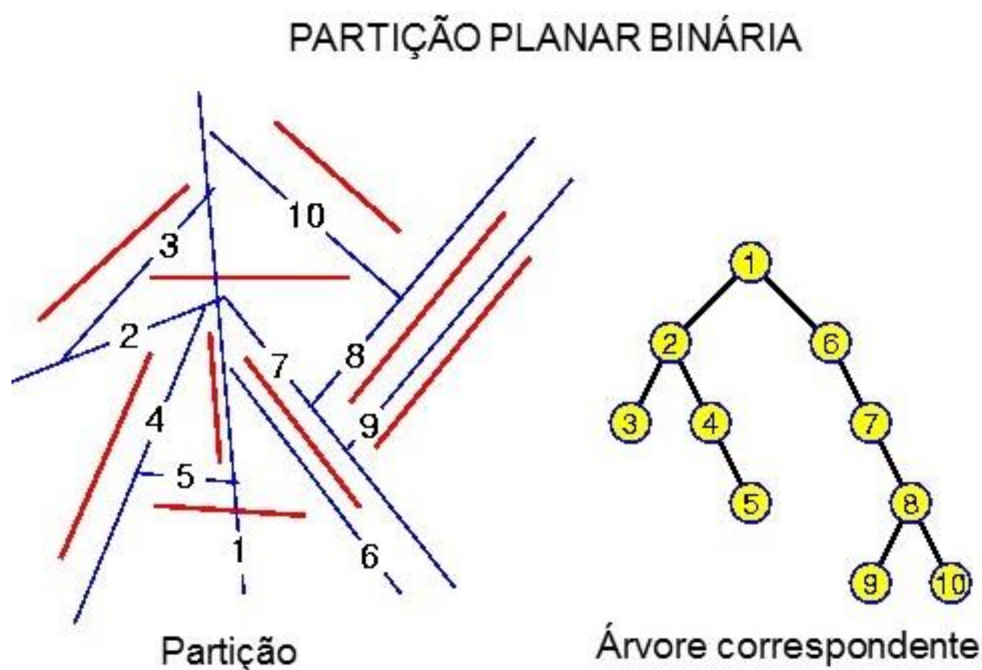
(GAME INSTITUTE, 2015), (ERICSON, 2007), (GREGORY, 2015) o mundo de um jogo é habitado por centenas, senão milhares de objetos. Realizar cálculos de colisão entre todos objetos ou verificar se todos estão no raio de visão da câmera para renderização é impraticável. E mesmo que fosse, tomaria uma grande parte do processamento, obrigando jogos a terem seu conteúdo reduzido.

Se um objeto está em um ponto, e outro está do outro lado do mapa é desperdício o simples fato de considerar a colisão entre eles. O particionamento de espaço tem por objetivo criar uma estrutura que permita buscar rapidamente entidades próximas de outra da câmera, descartando grandes porções de entidades sem qualquer chance de colisão.

Diversas estruturas de dados e algoritmos existem para particionar o espaço. Uma delas é o *Binary Space Partitioning* (BSP). Segundo Game Institute (2015), a árvore BSP é criada em tempo de compilação de um mapa (não deve ser alterada em tempo real, por razão de performance), onde uma árvore binária é criada; cada folha da árvore é um plano de divisão (parede, teto, piso, etc.) e contém do lado direito todos os objetos a frente do plano de divisão e do lado esquerdo da folha, os

atrás. Seu funcionamento é semelhante ao de uma árvore binária: da mesma forma que árvores binárias conseguem descartar grandes blocos de informação com uma simples comparação, BSPs descartam áreas do cenário (ERICSON, 2007). A estrutura de dados de uma BSP, junto com a divisão provocada no cenário pode ser vista na figura 7.

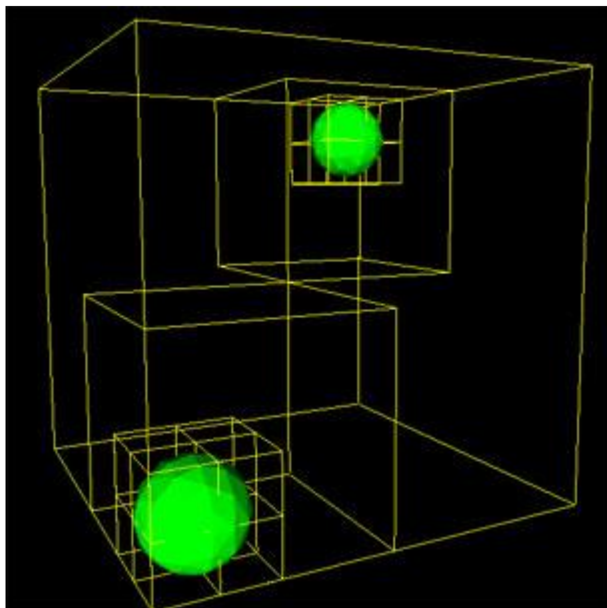
Figura 7 - Particionamento binário do espaço



Fontes: https://www.cs.berkeley.edu/~sequin/CS184/IMGS/bin_Planar_partition.gif (200-, Tradução do autor)

Em tempo real, *octrees* podem ser usadas para dinamicamente criar blocos de entidades próximas e ajustar o número de blocos de acordo com a distribuição de objetos. Como entidades ficam agrupadas em cubos, verifica-se se existe colisão entre os cubos antes de verificar a colisão entre os objetos que ali vivem (ERICSON, 2007). *Octrees* são visualmente uma série de cubos inseridos dentro de um cubo pai, podendo ter cubos filhos:

Figura 8 - Visualização da divisão espacial de um *octree*



Fonte: http://physics.ujep.cz/~mmaly/vyuka/ruzne/grafika/OpenGL/OpenGL%20-%20Open%20Graphics%20Library/OGL-Tutors/Octree_files/SpheresSub3.jpg (200-)

1.8 COLISÃO

O sistema de colisão utiliza o particionamento de espaço para evitar verificar colisão entre entidades distantes uma da outra. Para verificar a colisão são feitos testes em duas etapas: um para verificar quais objetos vão colidir (normalmente são fórmulas mais complexas levando em conta a velocidade dos objetos) e outra etapa verifica se objetos já estão colidindo (MILLINGTON, 2007).

Mesmo com o particionamento do espaço, um modelo detalhado ainda precisaria de milhares de verificação se fosse levado em conta todos os seus polígonos. Por isso um modelo muito menos detalhado é usado para colisão, enquanto o mais detalhado é usado apenas para renderização.

No caso de caixas (cubos), esferas, planos, cilindros, a verificação é ainda mais rápida, pois quando trata-se de triângulos cada um deve ser verificado

individualmente, enquanto as formas citadas possuem fórmulas que identificam intersecção sem precisar de iteração. Levando isso em conta, mesmo objetos com um modelo de colisão simplificado podem ter outro modelo de colisão utilizando cubos alinhados ao eixo ou esferas. Outra técnica é aplicar algoritmos de particonamento (como *octree*) nas próprias *meshes*. Lembrando que o fato de utilizar uma técnica não impede de usar outra, pelo contrário, elas combinam-se em diversas etapas da análise de colisão na esperança de resolver diferentes problemas de performance (GREGORY, 2015).

1.9 RENDERIZAÇÃO

Para renderizar uma cena, os vértices do mundo do jogo precisam seguir um fluxo de processos e tratamentos a fim de serem tirados de seu espaço em memória onde vivem em 3 dimensões ou mais e são transformados em 2D para os monitores respeitando a posição e ângulos da câmera por onde o jogador vê esse mundo.

No entanto, o grande obstáculo da renderização não é o processo esquematizado acima e por até contraditório que possa parecer, o desafio dos gerenciadores de cena é descobrir de forma eficiente o que não deve ser desenhado; é evitar ao máximo o número de chamadas para a placa de vídeo (*draw calls*) (GREGORY, 2015). O sistema de renderização deve preparar as *meshes*, mudar os estados dos buffers e operações de pixel para desenhar o cenário corretamente. Além desta organização, ainda é preciso organizar para ter um menor número de chamadas e mudanças de estado por parte da placa de vídeo (ELBERY, 2007).

Um modo de atingir esse objetivo é por meio de grafos de cena. Um gerenciador de cena agrupa em nós de uma árvore informações como transformação e estados. Desta forma, ao percorrer a árvore sabe-se que os filhos de um nó devem possuir os estados do pai, mas como os estados do pai já foram passados para a placa de vídeo, os filhos devem apenas ativar os seus próprios estados. Esta estrutura garante um número mínimo de mudanças de estado (MCSHAFFRY; GRAHAM, 2013). O esquema da figura 9 mostra objetos de uma árvore de cena relacionados às matrizes de transformação. É interessante notar que

uma multiplicação de matriz pode servir para mover mais de um objeto, evitando assim repetição de cálculos.

Figura 9 - Associação de matrizes de transformação

Fonte: http://archive.gamedev.net/images.gamedev.net/features/programming/scenegraph/image001.jpg (200-, tradução do autor)

1.10 ANIMAÇÃO

A ideia básica do sistema de colisão é permitir a movimentação de membros dos objetos do jogo (seja de um braço, um galho, uma flor, etc.). Animação por si só possui diversos algoritmos, como *mesh skinning*, cinemática inversa, combinação de animações (*animation blending*), animação por ossos (*bone animation*), *sprites*, etc (GRANBERG, 2009).

Jogos como Super Mario World, não possuem um sistema de animação baseando-se em transformação dos vértices dos modelos, confiando em um método muito mais rápido: *sprites*. Basicamente os movimentos são feitos em uma textura 2D, por exemplo, a animação de correr são três texturas, cada uma com o personagem em uma posição um pouco diferente da outra (LAMBERT, 2013). Por não envolver cálculos em vértices, o uso de *sprites* é extremamente rápido, mas por ser 2D, em jogos 3D, como Doom, ficam aparente (KUSHNER, 2003) e (DOOM

WIKIA, 200-). Jogos como Quake, Half-life, Sin, já não mais usam *sprites*, e partem para animações por vértice (VALVE, 200-).

Como outras técnicas, nada impede de combinar *sprites* com animação. O jogo Mass Effect 3, logo no início, mostra pessoas correndo a uma grande distância da câmera. Caso utilize a mira para dar zoom na cena, é possível notar que as pessoas fugindo são na verdade *sprites*, presume-se tal utilização para ganho de performance, mas não há confirmação por parte dos desenvolvedores (BIOWARE, 2012).

Uma técnica primitiva de animação é a *animation blending*, são dados dois modelos, um na posição inicial e outro na final. O sistema de animação utiliza interpolação entre as posições dos vértices para criar a movimentação dos personagens. Jogos como Quake usam esta técnica. Suas desvantagens são animações não muito realísticas, a necessidade de ter vários modelos aumenta o uso de memória. Mesmo assim, *blending* ainda é usado em jogos, mas não para animação de membros e sim para expressões faciais (GAME INSTITUTE, 2015) e (EPIC, 201-).

O jogo half-life foi um dos pioneiros em utilizar *bones* e *skinning* para animar modelos (VALVE SOFTWARE, 200-) Eles baseiam-se em associar os vértices aos “ossos” do modelo e a partir disso mover os vértices de acordo com a transformação dos seus ossos. Como a animação é feita por *bones* não por modelo, um mesmo esqueleto pode ser associado a diferentes personagens, desde que seus vértices sejam corretamente associados aos ossos (VEAZIE, 2014).

Skinning, por sua vez, flexibiliza o modelo, causando um efeito de pele, ou seja, quando o osso se movimenta, a “pele” ao redor estica ou retrai (UNIVERSITY OF CALIFORNIA, 200-).

Por fim, jogos como Half-life 2 aplicam um modelo físico às animações (*inverse kinematics*), fazendo personagens alinharem seus membros às superfícies em que se apoiam. Estas animações são geradas em tempo real e combinam-se com a animação atual do personagem (VALVE SOFTWARE, 200-) e (GRANBERG, 2009).

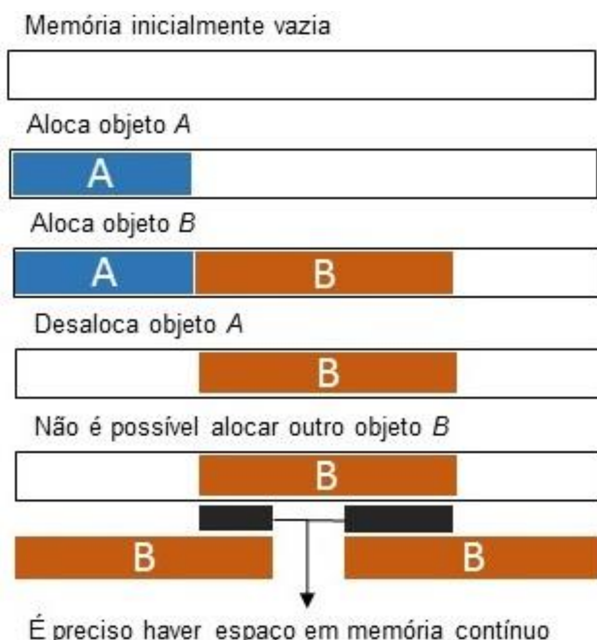
1.11 GERENCIAMENTO DE MEMÓRIA

Alocar e deslocar memória constantemente é impensável para um jogo, pois estes processos tomam tempo. Por isso *engines* utilizam seu próprio gerenciador de memória que busca evitar fragmentação, alocar e deslocar apenas quando necessário (MCSHAFFRY; GRAHAM, 2013).

Fragmentação de memória é um problema principalmente em sistemas de memória limitada. Diferente dos discos rígidos, a memória RAM não quebra objetos para serem colocados em dois ou mais blocos. Caso um objeto preciso de 5 bytes, é preciso de ter disponível um bloco contínuo de, no mínimo, 5 bytes (GREGORY, 2015).

Para evitar o problema, a *engine* reserva uma grande área de memória para ela (pool). Esta área é utilizada para guardar todos os objetos do jogo. Uma parte pode ser utilizada para guardar objetos que sempre ficarão em memória (parte estática) como o cenário. Outra parte é onde fragmentação pode ocorrer, pois nela objetos como inimigos, armas, balas, caixas, etc. são armazenados (parte dinâmica). A figura 10 explora o pool de memória tanto em situação de sucesso (há memória suficiente para alocação) como em caso de falho (não há suficiente).

Figura 10 - Falha de alocação em memória fragmentada



Fonte: http://gameprogrammingpatterns.com/images/object-pool-heap-fragment.png (201-, tradução do autor)

A parte dinâmica guarda uma identificação (*flag*) para cada instância. Esta *flag* diz se o objeto pode ser dealocado ou não. Quando um objeto é liberado para dealocação, o estado da *flag* muda, mas seu canto na memória não é limpo, para evitar perda de performance. Quando um novo objeto é instanciado, mas a área dinâmica não possui espaço anilhado para guardar o objeto, um ou mais objetos que estejam em sequência e com a *flag* de dealocação são finalmente dealocados e a nova instância assume seus lugares (NOEL, 2010).

1.12 INTELIGÊNCIA ARTIFICIAL

Millington (2006) divide a inteligência artificial de jogo em duas partes: navegação e decisão.

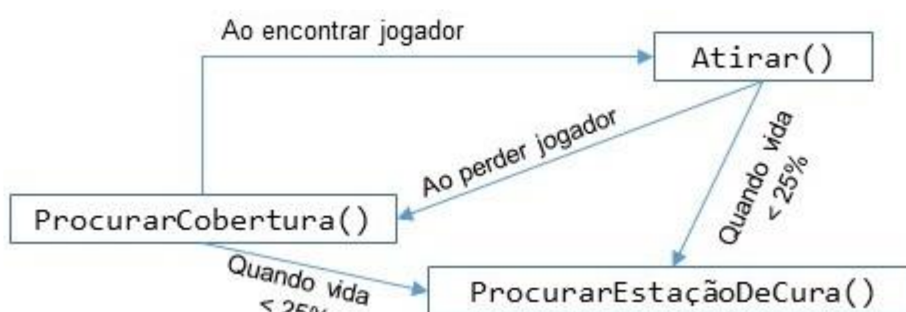
A navegação utiliza algoritmos como o A* para determinar qual caminho um personagem deve seguir para chegar ao seu objetivo. Normalmente o cenário possui várias alternativas, o trabalho do *pathfinder* é escolher de todos os caminhos possíveis o menos custoso. Custo pode referir-se a diversos elementos do cenário.

Por exemplo: um *bot* de Team Fortress precisa capturar uma bandeira. Ele possui as rotas A, B e C. A rota A possui o menor caminho, mas ela é guardada por um *sentry* inimiga. O *bot* lembra-se que na rota B ele avistou vários inimigos alguns minutos atrás. O caminho C é o mais longo, além de possuir água, o que faria a navegação levar ainda mais tempo, por outro lado é o caminho mais seguro. Qual rota o *bot* escolherá? (BUCKLAND, 2005).

A resposta depende do modo que a IA é implementada. O POD-bot, famoso bot de Counter-Strike, possui um sistema de personalidade para seus *bots* (PODBOT, 199-), com isso navegação combina-se com decisão e um *bot* mais “covarde” pode escolher o caminho C, enquanto um mais corajoso escolherá o B (ou até mesmo A). Outros jogos, o *pathfinder* indica o caminho menos custoso sem oferecer ao NPC a possibilidade de influenciar na decisão do caminho final.

Para tomar uma decisão diversos modelos são utilizados. Jogos como Half-life utilizam uma máquina de estado finito (CHAMPANDARD, 2008). A FSM (*finite state machine*) dispõe uma gama de ações para o NPC, podendo utilizar apenas um estado por vez. Como a transição de estados é pré-determinada e determinística, as ações dos NPCs podem se tornar previsíveis. Para aumentar a ilusão de inteligência e diminuir a previsibilidade, combina-se aleatoriedade ou lógica *fuzzy* para determinar as próximas ações (BUCKLAND, 2005). A figura 12 mostra os estados possíveis de uma FSM para algum inimigo de um jogo.

Figura 11 - Estados e transições de uma FSM de um inimigo



Fonte: Próprio autor

Por outro lado, programadores podem optar por deixar a própria IA determinar as ações que ela gostaria de executar. O jogo FEAR (MONOLITH, 2004) foi pioneiro em utilizar o GOAP (*Goal Oriented Action Planning*, ou planejamento de ação orientada a objetivo). Segundo Long (2007) cada ação existe como em uma FSM, mas as transições entre estados não são determinadas em tempo de programação. Uma entidade chamada de *planner* (planejador) vê as ações disponíveis como nós de grafo. Cada nó possui um custo, um efeito e conecta-se a outro através de uma pré-condição. Por exemplo, atirar tem o efeito de matar e a pré-condição de ter munição, o nó de ter munição tem a pré-condição de ter uma arma. Para determinar qual das ações executar, o *planner* utiliza um algoritmo de *pathfinding* para determinar a caminho de nós menos custoso (processo semelhante ao *pathfinding* para navegação).

DESENVOLVIMENTO DA ENGINE

1.13 VISÃO GERAL

A *engine* desenvolvida neste projeto visa oferecer a capacidade de criar um jogo FPS (*first person shooter*, ou tiro em primeira pessoa), feita puramente em C++ e HLSL, não usando bibliotecas de apoio; apenas APIs do sistema operacional e hardware.

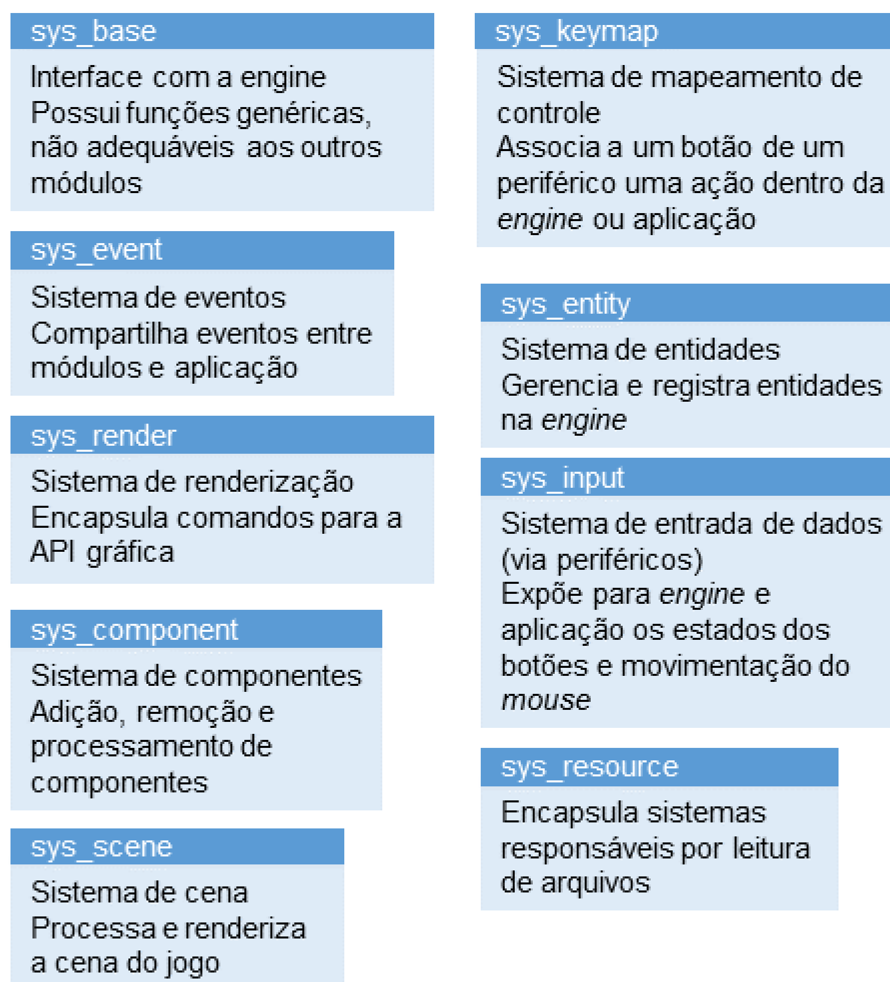
Seus principais recursos são:

- Leitura de script
- Leitura de arquivos com informações do jogo (mapas, texturas, malhas, etc.)
- Gerenciamento automático de memória/*garbage collector* (coletor de “lixo”)
- Separação entre código da aplicação (jogo) e chamadas para o sistema operacional, drivers e hardware
- Flexibilização de entidades (permite criar diferentes objetos em jogo)
- Efeitos gráficos em tempo real, opcionais e flexíveis
- Suporte para renderização 2D e 3D
- Suporte para DirectX 11 e Windows 8 e 10
- Sistemas modulares permitem expansão do código

O principal aspecto técnico do desenvolvimento da *engine* foi baseá-la em módulos, ou seja, a chamada para cada subsistema é isolada em um macro, como ilustra a figura 12. Cada módulo possui responsabilidades únicas, por exemplo, o módulo de cena adiciona, remove, gerencia objetos, fontes, efeitos, etc. em cena. A comunicação entre eles é feita através de um sistema de eventos a fim de diminuir

acoplamento e deixar os sistemas mais isolados e independentes (MCSHAFFRY; GRAHAM, 2013).

Figura 12. Principais subsistemas da *engine*



Fonte: próprio autor

No que tange as decisões de qual rumo o desenvolvimento teve de tomar, foi usada uma mentalidade imediatista e pragmática. Cada recurso presente na *engine* serve uma função no jogo. Não existiu a criação de algo que pudesse ser usado no futuro – todas as decisões basearam-se no presente. Quanto ao design e implementação do jogo foi usada a mesma filosofia do Looking Glass Studio durante a produção do clássico System Shock 2, evitado ir contra a própria tecnologia

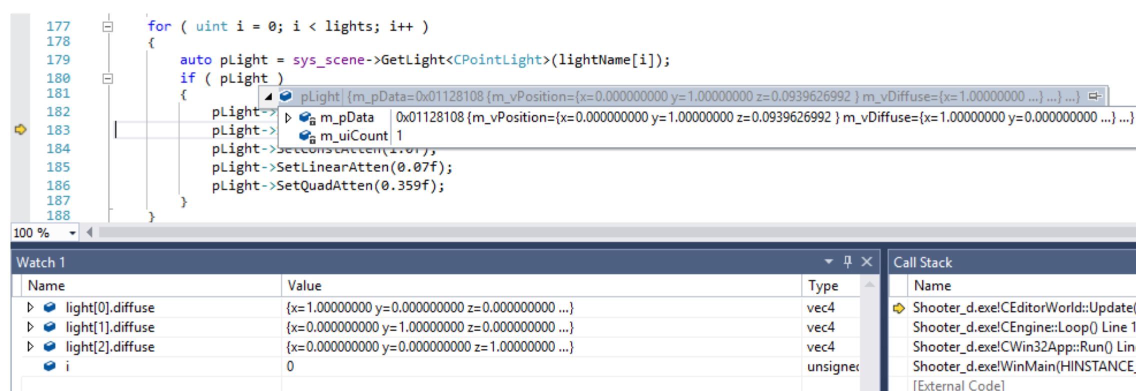
(CHEY, 1999), ou seja, não “forçar” o jogo a ter certa característica que precise de uma reestruturação ou grande mudança no código da *engine*.

1.14 FERRAMENTAS

A *engine* foi desenvolvida na linguagem C++, usando as especificações do C++11 (ESPECIFICAÇÕES C++11). A IDE foi o Microsoft Visual Studio 2015 Community Edition. O motivo por ter escolhido este software são:

- Compilador C++ compatível com C++11
- Coloração de sintaxe: palavras e símbolos são coloridos de acordo com seu significado no código fonte
- *Debugger* (figura 13): permite executar o código linha-a-linha, verificando os valores e endereços das variáveis
- *Debugger* de *Shader* (figura 14): mostra os processos, estados e dados de programas executados na placa de vídeo

Figura 13 - *Debugger* do Visual Studio 2015 Community Edition



Fonte: Próprio autor

Figura 14 - *Debugger* de *Shader*

Fonte: Próprio autor

1.15 DETALHES TÉCNICOS DA ENGINE

Este capítulo explora os principais módulos e técnicas usadas pela *engine* para descrever, abstrair e relacionar funcionalidades.

Os tópicos iniciais possuem conceitos extremamente amplos e genéricos (não focados em jogos), mas são de fundamental importância para unificar, padronizar e encapsular funções e tipos específicos de hardware e do sistema operacional. A cada tópico os assuntos abordados ficam mais próximos de conceitos aplicáveis a jogos.

1.16 MACROS, FUNÇÕES, REDEFINIÇÃO DE TIPOS E RESPOSTA A ERROS

A exploração em detalhes da *engine* começa pela redefinição de tipos. A fim de não referenciar *headers* da API do Windows em todo o código fonte e para tornar os nomes dos tipos mais amigáveis ao digitar, são usados *typedefs*, algumas redefinições podem ser vistas na figura 15.

Figura 15 - Algumas definições de tipos (*typedefs*)

```
typedef unsigned int uint;
typedef __int64 i64;
typedef unsigned __int64 ui64;
typedef unsigned short ushort;
typedef unsigned long ulong;
typedef unsigned char uchar;
typedef wchar_t wchar;

#define UINT_INVALID (~0U)

typedef std::wstring strw;
typedef std::wifstream fileReadStreamW;
typedef std::wofstream fileWriteStreamW;
typedef std::string stra;
typedef std::ifstream fileReadStreamA;
typedef std::ofstream fileWriteStreamA;

#if defined(UNICODE) || defined(_UNICODE)
    typedef fileReadStreamW fileReadStream;
    typedef fileWriteStreamW fileWriteStream;
    typedef strw str;
#else
    typedef fileReadStreamA fileReadStream;
    typedef fileWriteStreamA fileWriteStream;
    typedef stra str;
#endif

#if !defined(NULL)
    #define NULL (void*)0
#endif
```

Fonte: Próprio autor

Com os tipos redefinidos, algumas funções e constantes de ampla funcionalidade são definidas, em sua maioria, no header Util.h. Este header vai de

desencontro à arquitetura modular e de responsabilidade única, mas seu propósito é, de fato, servir razoavelmente todos os módulos.

A figura 16 possui algumas definições (macros) usadas no arquivo Util.h. Entre as funções estão: escolha aleatório de número, arredondamentos, formatação de strings, conversões, interpolação linear, verificação de igualdade, etc.

Figura 16 - Exemplo de macros em Util.h

```
#define PI 3.141592f
#define HALF_PI (PI * 0.5f)
#define TWO_PI (PI *2.0f)
#define ZERO_EPSILON 0.000001f
#define UINT_LIMIT ((uint)(~0))

#define ASSERT(exp) { static bool _skip_ = false; \
    if ( !_skip_ && AssertMessage((bool)(exp), #exp, "", __LINE__, __FILE__, _skip_) ) __asm{int 3} }
#define ASSERT2(exp, msg) { static bool _skip_ = false; if \
    ( !_skip_ && AssertMessage((bool)(exp), #exp, msg, __LINE__, __FILE__, _skip_) ) __asm{int 3} }
#define ASSERT3(msg) { static bool _skip_ = false; \
    if ( !_skip_ && AssertMessage(false, "", msg, __LINE__, __FILE__, _skip_) ) __asm{int 3} }

#define ARRAY_SIZE(x) ( sizeof(x) / sizeof(*x) )

#define THROW(msg) { ASSERT3(msg); throw CException(msg); }

#define NOCOPYABLE(className) \
    private: className(className&); className& operator = (const className&);

#ifndef min
    #define min (a, b) ( ((a) < (b)) ? (a) : (b) )
#endif

#ifndef max
    #define max (a, b) ( ((a) > (b)) ? (a) : (b) )
#endif

#define GETINSTANCEDECL(classname) public: static classname* \
    GetInstance() { static classname instance; return &instance; }
```

Fonte: Próprio autor

Outros arquivos com sufixo Util.h também estão presentes, no entanto eles possuem metas mais objetivas e de maior proximidade com uma aplicação 3D. Por exemplo, o header TextureUtil.h possui estruturas (structs, como são conhecidas no C++), enumerações (enums chamadas no C++), funções específicas de textura e, portanto, utilizadas majoritariamente pela classe de textura (CTexture) e gerenciador de textura (CTextureManager). A figura 17 apresenta trechos de código deste arquivo.

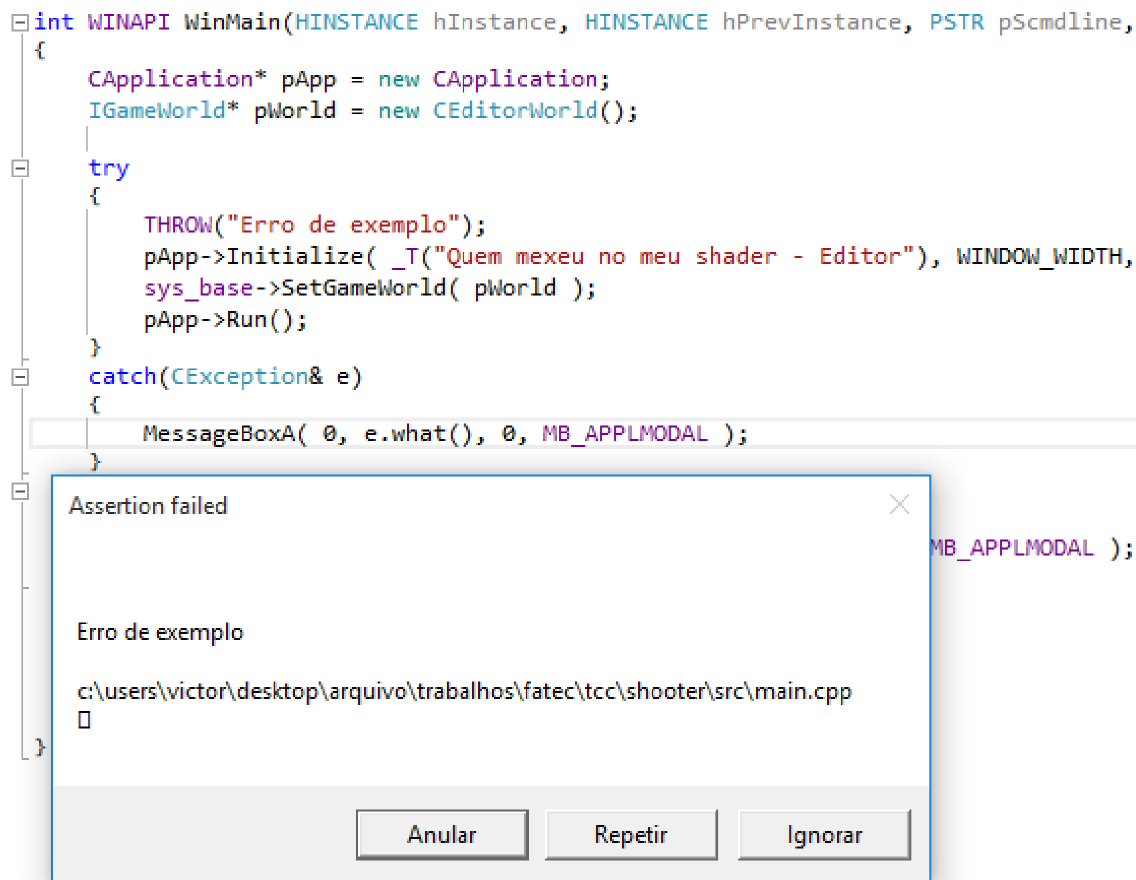
Figura 17 - Trechos de código do *header* TextureUtil.h

```
#include "Util.h"

struct ID3D11ShaderResourceView;
struct ID3D11Texture2D;

struct TextureAPIInfo
{
    ID3D11ShaderResourceView* pDX11TextureView;
    ID3D11Texture2D* pDX11Texture;
};

enum class COLORFORMAT
{
    RGB,
    RGBA,
    BGR,
    BGRA,
    UNKNOWN
};

inline int GetRedIndex(COLORFORMAT format)
{
    switch(format)
    {
        case COLORFORMAT::RGB:
        case COLORFORMAT::RGBA:
            return 0;

        case COLORFORMAT::BGR:
        case COLORFORMAT::BGRA:
            return 2;

        default:
            ASSERT3("Format not recognized or does not support red channel");
            return -1;
    }
}
```

Fonte: Próprio autor

Por fim, uma classe de exceção (exception, assim chamada em C++) é criada para caso o programa precise ser interrompido inesperadamente e apresentar uma mensagem de erro ao usuário. A figura 18 apresenta a declaração da classe e uma mensagem de erro de exemplo.

Figura 18 - Mensagem de erro e classe de exceção



Fonte: Próprio autor

1.17 ALGORÍTMOS E ESTRUTURAS DE DADOS

Com os principais tipos, algumas funções básicas e capacidade para tratar erros. O próximo passo é expandir a *engine* para ela suportar formas de manipular, ler e gravar dados rápidas e eficientes.

A estrutura de dados mais simples na *engine* é a lista. Ela possui alocação de memória dinâmica com base na necessidade da aplicação e é acessada aleatoriamente por índice. Assim como todas as outras estruturas, a lista suporta diversos tipos graças aos genéricos (*templates* em C++). A figura 19a apresenta um exemplo da *engine* usando lista.

Outra estrutura de dados é a lista duplamente encadeada (na engine é chamada de *linked list*) não há uma lista simplesmente encadeada. A principal diferença entre esta estrutura e a lista é possuir rápida inserção e remoção, pois não precisa realocar um *array* cada vez que fizer uma destas operações. Em contrapartida, não possui meios de ser acessada diretamente por índice. A figura 19b ilustra o uso da *linked list* pela *engine*.

A última estrutura é a tabela *hash* (*hash table*). Esta estrutura é utilizada para encontrar rapidamente algum objeto ou valor. Como um dicionário, um valor de rápida identificação é associado ao objeto. Deste valor é possível diminuir o número de buscas para encontrar o objeto desejado. A figura 19c ilustra o uso da *hash table* pela *engine*.

Figura 19 - Estruturas de dados sendo utilizadas na *engine*

A. LISTA

```
template <class T, class KEY>
inline void hashTable<T, KEY>::getAll(list<T>& l)
{
    if ( m_pList->count() > 0 )
    {
        l.setPreAlloc( count() );
        for ( uint i = 0; i <= m_uiLastIndex; i++ )
        {
            for ( auto pNode = (*m_pList)[i].first(); pNode; pNode = pNode->next() )
                l += pNode->get().value;
        }
    }
}
```

B. LISTA ENCADEADA

```
for ( auto pNode = m_llDebugLineTimed.first(); pNode; pNode = pNode->next() )
{
    pNode->get().time -= sys_base->GetDeltaTime();

    if ( pNode->get().time < 0.0f )
    {
        pNode->removeGetValidNextFirst( pNode );

        if ( !pNode )
            return;
    }
}
```

C. *HASHTABLE*

```
auto pNode = m_htHashStr.find(uiHash);
if ( pNode )
{
    if ( (T)pNode->get().value != sText )
    {
        THROW("CHashStr::Register() code already registered");
        return false;
    }
}
else
    pNode = m_htHashStr.add(sText, uiHash);
```

Fonte: Próprio autor

1.18 GERENCIAMENTO DE MEMÓRIA

Com as estruturas de dados prontas, uma implementação mais objetiva, porém não relacionada a um recurso em um jogo, é possível. O gerenciamento de memória é feito para evitar que ponteiros possam utilizar um endereço de memória inválido.

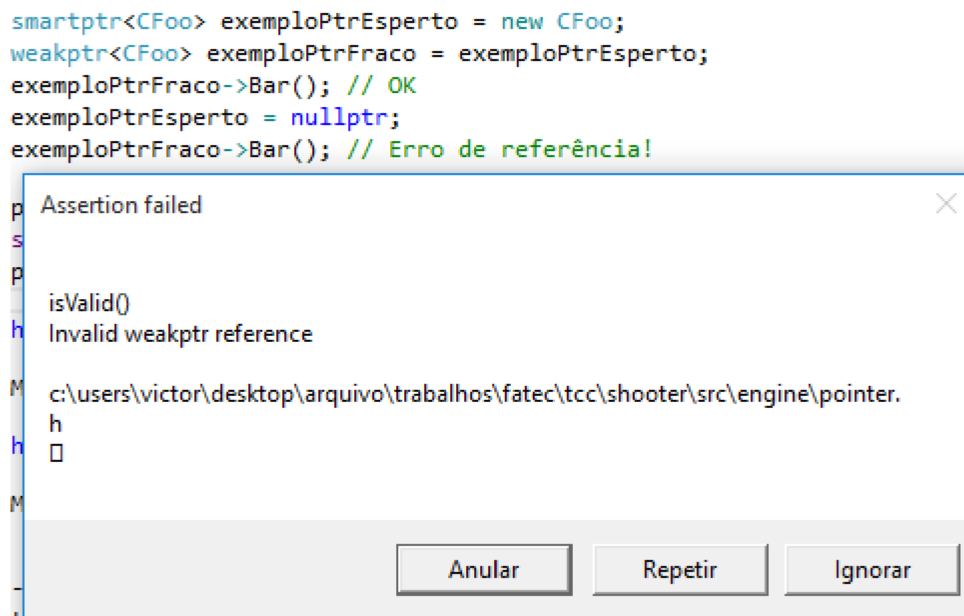
McShaffry e Graham, 2015 implementam este conceito utilizando uma classe que encapsula e esconde o ponteiro; seu nome é ponteiro esperto (*smartptr* na *engine*). Ele é acessado por uma interface e sua desalocação é controlada por um contador interno. Cada vez que a classe de ponteiro esperto é copiada, seja por construtor ou por atribuição, este contador é incrementado. Cada vez que o destrutor é chamado, o contador decrementa.

Ao chegar em zero, a classe destrói seu ponteiro.

Uma classe de apoio chamada ponteiro fraco (*weakptr*) foi criada para referenciar um ponteiro apontado por um *smartptr*, mas ao contrário desta, o *weakptr* não incrementa qualquer contador. Sua implementação esconde um dicionário de memória, aproveitando-se da rapidez das *hash tables* e *linked lists* para manter uma relação de todas as referências que os ponteiros espertos fazem. A referência é invalidada quando o objeto sendo apontado não existe mais.

A cada chamada, o ponteiro fraco verifica se sua referência ainda é válida. Caso não seja, emite um alerta para o programador. A figura 20 mostra o uso dos dois tipos de ponteiro e a validação proveniente do ponteiro fraco.

Figura 20 - Exemplo de ponteiros esperto e fraco



Fonte: Próprio autor

1.19 MATEMÁTICA

Antes de poder iniciar o desenvolvimento dos recursos da *engine* com conceitos mais concretos capazes de serem identificados durante uma jogatina. Ela precisa possuir meios de representar vértices, vetores e matrizes, além de realizar operações de translação, rotação, projeção, etc. para assim construir e manter um mundo 3D.

As primeiras classes são vetores 2D, para a interface com o usuário e texturas. 3D, para representar todo o mundo jogável e 4D, utilizado para representar cores RGBA (*red*, *green*, *blue* e *alpha*).

As classes de vetor suportam diversas funções como produto escalar e vetorial, conversão entre dimensões (transforma um vetor 4D para 3D, por exemplo), normalização e transformação por matrizes.

Já a classe de matriz não possui ramificações. Há apenas uma classe mat44 representando uma matriz linha (*row major*) com 4 linhas e colunas. As principais

operações são rotação (tanto por ângulos de Euler quanto por ângulo e eixo. Dunn e Paberry, 2011), translação, inversão, transposição e multiplicação.

Por fim, o header MathUtil.h centraliza funções de colisão, como verificar se um ponto está dentro de um cubo; cria matrizes de projeção (perspectiva e ortogononal), cria espaços ortonormais (todos os eixos formam 90º com os outros e possuem módulo unitário), etc.

1.20 GERENCIADOR DE MODELOS (MESHES)

Modelos (*meshes*) são vértices organizadas em pontos, linhas ou triângulos para formar uma figura para ser vista durante o jogo (como um personagem, uma mesa, uma parede, etc.).

A principal tarefa do gerenciador de *mesh* é ler e armazenar os modelos. Podendo disponibilizá-los quando a *engine* precisar sem perda de desempenho. Isso é feito usando a tabela *hash*, onde a chave para o rápido acesso é o nome do arquivo da *mesh*, não sendo permitido repetição de nomes. A figura 21 mostra um modelo carregado pela *engine* e os triângulos que o forma.

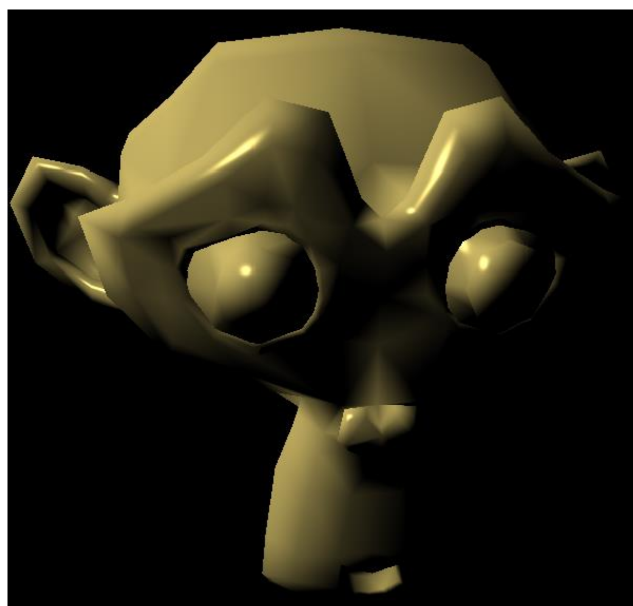
Figura 21 - Modelo sendo executado pela *engine*

Fonte: Próprio autor

Por parte da renderização, o gerenciador precisa garantir o modelo em formato de vértice específico para o *shader*. Isso é preciso pois para cada efeito gráfico desejado é preciso haver determinadas informações vindas do modelo. Os casos a seguir ilustram alguns possíveis formatos e onde quais efeitos podem produzir:

- Vértice apenas com posição. Este formato não é intuitivo, mas ele pode ser usado para criar um mapa de profundidade utilizado por sombras.
- Vértice com posição e coordenada de textura. Este formato pode ser usado para exibir bitmaps, responsáveis por criar as fontes e, por consequência, os textos em 2D (como o menu inicial, a mensagem de fim de jogo, os créditos do jogo, etc).
- Vértice com posição e normal. Pode ser usado para criar um objeto 3D que "sabe" como reagir à luz. Por exemplo, se a luz do sol atingir uma esfera com vértices neste formato, ela ficará mais iluminada na direção do sol, e os pixels ao redor começaram a escurecer quando mais distantes deste ponto chegarem. A figura 22 ilustra esta situação.

Figura 22 - Modelo com informação de posição e normal sendo iluminado por um "sol"

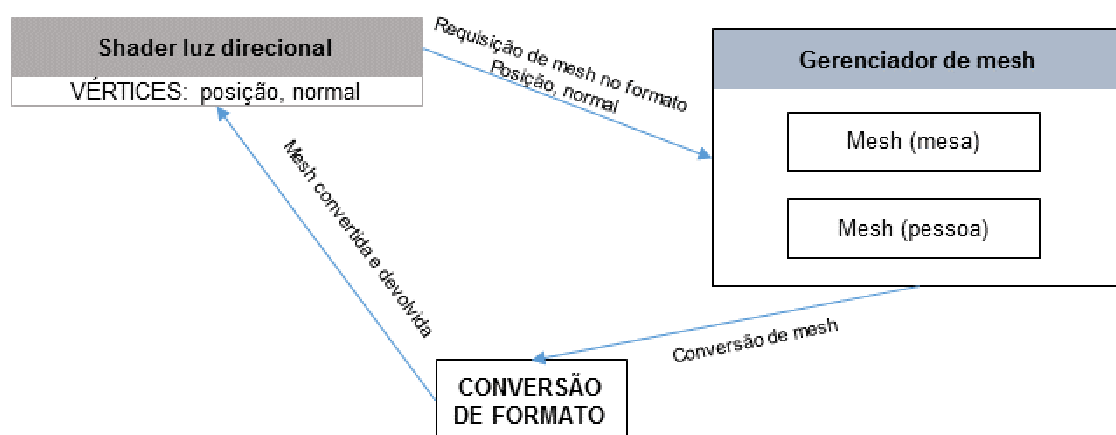


Fonte: Próprio autor

Isso é feito inteirando o gerenciador de modelo e as classes de *shader*, A figura 23 esquematiza o sistema. Quando um *shader* é criado, ele precisa receber um formato de vértice. Este formato é passado para o gerenciador de modelo no momento da renderização, para ele tentar adaptar a *mesh* ao formato do *shader*. Diversas funções de conversão são disponibilizadas para que isso ocorra.

Uma vez convertida, a *mesh* em novo formato é armazenada para não precisar processá-la novamente.

Figura 23 - *Shaders* "pedindo" ao gerenciador de modelo uma *mesh* em seu formato



Fonte: Próprio autor

1.21 RENDERIZADOR, SHADERS E EFEITOS

Para exibir um mundo 3D acelerado por hardware, é preciso uma API gráfica. A *engine* utiliza o DirectX 11, o principal motivo da escolha é integração nativa com o Visual Studio 2015 Community Edition e facilidade para "debugar".

O DirectX 11, limitado ao uso da engine, deve fornecer – e fornece – os recursos:

- Criação de um contexto de renderização para exibir o mundo 3D
- Suporte para *shader*

- Suporte para transparência e combinação de pixels
- Suporte para textura
- Suporte para oclusão e escrita em buffer de profundidade (usado para iluminação e sombra)

A fim de criar uma interface comum não-dependente de hardware ou API, todas as chamadas para as funcionalidades acima são centradas no módulo de renderização, onde devem ser chamadas pelo macro sys_render. Este módulo é usado pelo gerenciador de cena e classes de efeito para desenhar o mundo 3D.

O verdadeiro responsável por comandar a renderização dos modelos são os programas *shaders*. Programas *shaders* são escritos, no caso do DirectX 11, em HLSL uma linguagem parecida com o C, executada na placa de vídeo operando vértices e pixels (*vertex shader* e *pixel shader*, respectivamente). A *engine* controla esses programas pelas classes de efeito e classe de *shader*. A principal função da primeira é controlar os estados que o adaptador gráfico deve assumir ao executar cada instrução (ligar/desligar transparência, habilitar/desabilitar escrita no buffer de profundidade, etc.). A segunda deve compilar os programas *shader* e enviar para a placa de vídeo os dados necessários para executar o programa, como texturas e matrizes.

1.22 GERENCIADOR DE CENA

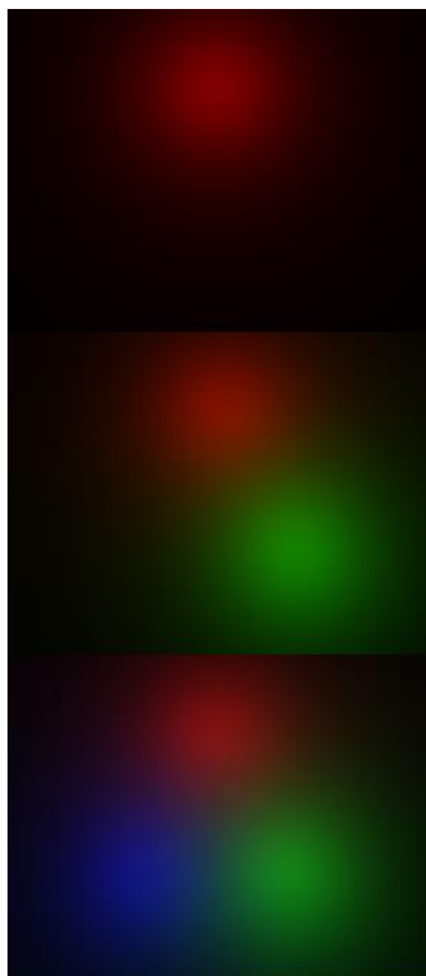
A decisão do que deve ser renderizado e qual *shader* será utilizado é tomada pelo gerenciador de cena. Este módulo possui um registro de todos as entidades com um componente de renderização e de todas as luzes no mundo virtual. Cada componente de renderização possui informações sobre seu tipo. Por exemplo, o tipo água requer um *shader* de reflexo e refração para ser executado; já um de texto, precisa de um *shader* de fonte, e assim por diante.

As entidades são registradas no gerenciador de cena através do envio de mensagem. A criação do componente de renderização faz o gerenciador de

componentes disparar uma mensagem de criação que é "ouvida" pela cena, assimilando em sua lista de componentes o novo criado.

A cena 3D é renderizada usando a mesma técnica do jogo Doom 3 (Sanglard, 2012), onde uma primeira renderização é feita apenas usando informação de profundidade, para identificar os objetos que estão na frente de outros. Após, cada luz é renderizada individualmente e o resultado é adicionado à imagem final. A figura 24 demonstra o processo passo-a-passo.

Figura 24 - Renderização de cena usando múltiplos passes



Fonte: Próprio autor

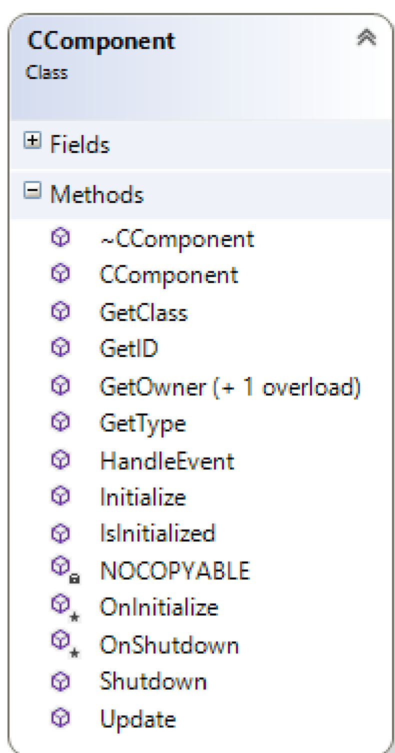
1.23 ENTIDADES, COMPONENTES E SEUS GERENCIADORES

As entidades são classes contendo componentes. Cada componente possui determinada função. Alguns renderizam, outros colidem, outros são a inteligência de um inimigo. A classe base de componente permite criar com heranças pouco extensas e os mais diversos comportamentos para o jogo.

Entidades existem apenas para unificar os vários componentes que formam um objeto em jogo. Por exemplo, o objeto "arma" é uma entidade que existe combinando os componentes: renderização, colisão, transformação, script.

A classe de componente está esquematizada na figura 25. Ela é uma base que obriga seus filhos a terem o método Update(), GetType() e GetClass(). O primeiro tem a função de processar o comportamento da classe. Os outros dois servem para ele ser diferenciada pela entidade.

Figura 25 - Métodos da classe CComponent



Fonte: Próprio autor

O gerenciador de componentes é onde todos os componentes devem ser registrados para “existirem” no mundo do jogo, pois esta classe é responsável por chamar todos os métodos do componente, como inicialização, *update*, etc. Sem ela, é como se eles ficassem “parados no tempo”. A figura 26 possui o código responsável por fazer chamadas para todos os componentes.

Figura 26 - Gerenciador de componente processando os componentes do jogo

```
void CComponentManager::Update()
{
    list<smartptr<CComponent>> lComponent;
    m_htComponent.getAll( lComponent );
    for ( uint i = 0; i < lComponent.count(); i++ )
    {
        ASSERT2( lComponent[i]->IsInitialized(), "Attempting to update uninitialized component" );
        lComponent[i]->Update();
    }
}
```

Fonte: Próprio autor

A classe de entidade possui duas informações essenciais: os componentes da entidade, responsáveis por dar funcionalidade e o tipo, utilizado para saber o que a entidade representa (se é um inimigo, uma arma, etc.). Aliado, está o método de obter e adicionar seus componentes usando genéricos. A figura 27 possui trechos de código responsáveis por criar uma entidade com componentes.

O gerenciador de entidade é quem registra, inicializa e destrói as entidades.

Figura 27 - Código demonstrando a criação de uma entidade

```
pEntity = sys_entity->CreateEntity();
pRender = pEntity->AddComponent<CComponentRender>();
float cubeSize = 40.0f;
pRender->SetMesh( createCube(cubeSize, 0.1f, cubeSize) );
m = Material();
m.specularPower = 200;
m.specular.set( 0.0f );
m.diffuse.set( 1.0f );
pRender->SetMaterial( m );
pRender->SetTexture(sys_resource->LoadTGATexture(L"../Texture/seafloor.tga"));
pTransform = pEntity->AddComponent<CComponentTransform>();
pTransform->SetPos( vec3(0.0f, -2.0f, 0.0f) );
```

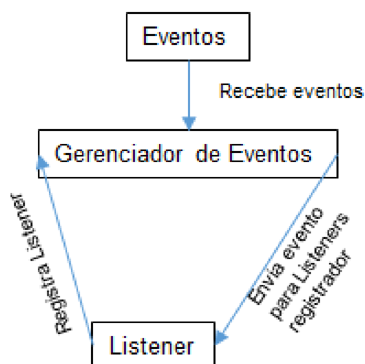
Fonte: Próprio autor

1.24 GERENCIADOR DE EVENTOS

Este gerenciador é o responsável por fazer diferentes módulos se comunicar.

A figura 28 ilustra seu funcionamento. A estrutura evento pode ser herdada para criar específicos, como colisão, movimento, etc. O evento é registrado no gerenciador de eventos. Este itera a cada quadro enviando o evento para os ouvidores (*listeners*). O *listener* é uma classe base que possui o método HandleEvent(), responsável por interpretá-lo.

Figura 28 - Esquema do sistema de eventos



Fonte: Próprio autor

Cada módulo pode possuir uma implementação de *listener* diferente. A figura 29 possui o código do módulo de cena fazendo a leitura do evento de criação de um componente de renderização para adicionar ao mundo 3D e possui o código de envio de evento feito pela entidade quando cria um novo componente.

Figura 29 - Código demonstrando a implementação de um *listener*

```
void CSceneListener::HandleEvent(Event* pEvent)
{
    EventComponent* pec;
    EventEntity* pee;

    if ( eventcmp(pec, *pEvent) )
    {
        if ( compcmptype<CComponentRender>(pec->pComponent) || compcmptype<CComponentGUI>(pec->pComponent) )
            sys_scene->OnEntityChange( pec->pComponent->GetOwner() );
    }
    else if ( eventcmp(pee, *pEvent) )
    {
        if ( pee->action == ENTITYACTION::REMOVE )
            sys_scene->RemoveFromEntityGroup( pee->pEntity );
    }
}
```

Fonte: Próprio autor

1.25 SISTEMA DE RECURSO

O sistema de recurso encapsula gerenciadores que precisam fazer leitura de arquivos, como script, fontes e texturas. Por apenas centralizar e fazer chamadas para esses gerenciadores,

1.26 CENTRALIZAÇÃO DOS MÓDULOS NA CLASSE CENGINE

Por fim, a classe CEngine é quem instancia, destrói e atualiza todos os sistemas da *engine*. É por ela (usando os sys_) que a aplicação pode acessar os módulos e interagir com eles. A figura 30 possui o código de declaração dos sys_.

Figura 30 - Definição dos *sys_*

```
#define sys_base CEngine::GetInstance()
#define sys_event sys_base->GetEventManager()
#define sys_render sys_base->GetRenderer()
#define sys_component sys_base->GetComponentManager()
#define sys_entity sys_base->GetEntityManager()
#define sys_input sys_base->GetInput()
#define sys_keyMap sys_base->GetKeyMap()
#define sys_resource sys_base->GetResourceManager()
#define sys_scene sys_base->GetScene()
#define sys_app sys_base->GetApplication()
```

Fonte: Próprio autor

A aplicação do jogo precisa registrar-se com a *engine* para fazer as chamadas para a lógica do jogo. Isso é feito com a interface IGameWorld, representada na figura 31.

Figura 31 - Interface entre engine e aplicação

```
class IGameWorld
{
    public:
        virtual bool Initialize() = 0;
        virtual void Shutdown() = 0;
        virtual void Update() = 0;
        virtual void Render() = 0;
};
```

Fonte: Próprio autor

PROTÓTIPO DO JOGO "QUEM MEXEU NO MEU *SHADER*?"

O protótipo de jogo possui a função de demonstrar a *engine*, principalmente o sistema de entidade flexível, eventos e renderização dinâmica. O cenário é uma pequena arena onde há obstáculos, o jogador e um vírus. Com o passar do tempo, um efeito gráfico é perdido (como um tipo de luz, ou um reflexo). O jogador pode recuperar este efeito atirando no vírus que fica navegando pelo cenário. Para isso foram criadas entidades totalmente diferentes:

- **Jogador** possui o interpretador de eventos do teclado e controle de câmera, além de atirar no vírus.
- **Vírus** uma IA com o comportamento de andar sem uma trajetória pré-definida.
- **Obstáculos** são compostos pelo componente de renderização e colisão, impedindo o jogador e o vírus de passarem por eles, limitando o espaço.

Embora as entidades possuam comportamentos completamente diferentes, a nível de código a *engine* foi capaz de oferecer uma interface única, porém flexível. As figuras 32 e 33 mostram a classe onde a lógica do jogador e vírus é implementada, respectivamente; a 34, o resultado final do protótipo.

Figura 32 - Declaração da classe do jogador

```
class CComponentPlayer : public CComponent
{
    COMPDECLALL(CComponentPlayer, "Player");

    public:
        CComponentPlayer();

        virtual void OnInitialize()override;
        virtual void Update()override;

        void SetMoveSpeed(float speed) { m_fPlayerMove = fabs(speed); }
        void SetCameraRotationSpeed(float speed) { m_fCamRotate = fabs(speed); }
        weakptr<CEntity> GetBullet() { return m_pBullet; }

    private:
        float    m_fPlayerMove;
        float    m_fCamRotate;
        float    m_fReloadTime;
        weakptr<CEntity> m_pBullet;
};
```

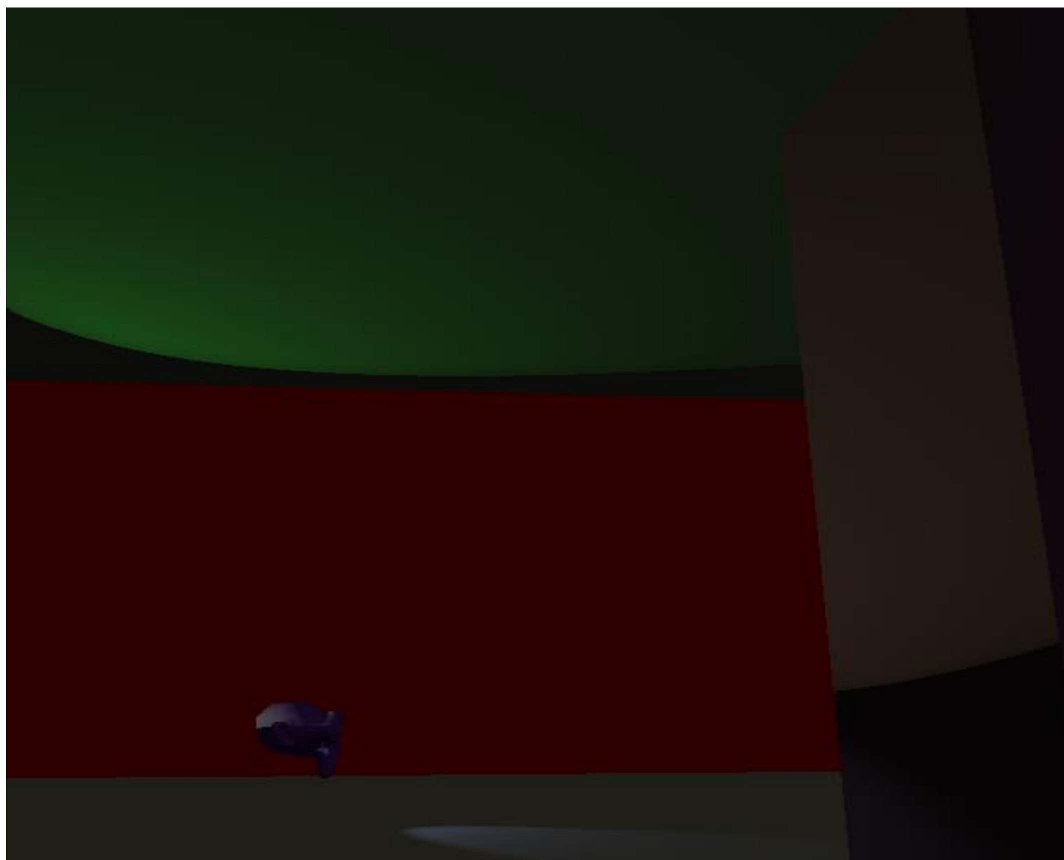
Fonte: próprio autor

Figura 33 - Declaração da classe do vírus

```
class CComponentVirusAI : public CComponent
{
    COMPDECLTYPE(CComponentVirusAI, "VirusAI");
    COMPDECLCLASS(CComponentVirusAI);

    public:
        CComponentVirusAI() {}
        virtual ~CComponentVirusAI() {}

        virtual void OnInitialize()override;
        virtual void Update()override;

    private:
        vec3 m_vPointInCircle;
        vec3 m_vVelocity;
        weakptr<CSpotLight> m_pLight;
};
```

Fonte: próprio autor

Figura 34 - Protótipo do jogo



Fonte: próprio autor

CONCLUSÕES

O protótipo do jogo “Quem mexeu no meu *shader*” esquematiza as várias interações e especificações (novos objetos) com simplicidade e fluidez, dado que a *engine* criada mostra-se capaz de suportar todos os recursos do jogo, sem ter precisado sofrer qualquer alteração em sua arquitetura, provando a hipótese levantada de ter a *engine* e o protótipo prontos.

Disso segue que uma *engine* feita por uma equipe pequena é capaz de fornecer as facilidades para a criação de um jogo quando este possui escopo limitado e a *engine* é feita com seus requisitos em mente. Mesmo sem a capacidade de concorrer com uma comercial, sua especialidade pode ser reaproveitada em outros jogos de mecânica similar.

REFERÊNCIAS BIBLIOGRÁFICAS

AMRESH, A; OKITA, A. **Unreal Game Development**. Commonwealth Road: AK Peters, 2010. 395 p.

Aprenda com Unity. Disponível em: <unity3d.com/pt/learn>. Acesso em: 05 nov. 2015.

Best Video Games of All Time - Metacritic. Disponível em: <http://www.metacritic.com/browse/games/score/metascore/all/all/filtered>. Acesso em: 24 set. 2015.

BIOWARE. **Mass Effect.** Disponível em: <masseffect.bioware.com>. Acesso em: 05 nov. 2015.

Blending Animations. Disponível em: <docs.unrealengine.com/latest/INT/Engine/Animation/AnimationBlending/index.html>. Acesso em: 05 nov. 2015.

BUCKLAND, M. **Programming Game AI by Example**. Plano: Wordware, 2005. 481 p.

Canal GamesIndie. Disponível em: <https://www.youtube.com/channel/UCEbEugtl-glHmFQFmlgNcKQ>. Acesso em: 05 nov. 2015.

CHAMPANDARD, A. J. **The AI from Half-Life's SDK in Retrospective.** Disponível em: <http://aigamedev.com/open/article/halflife-sdk/>. Acesso em: 05 nov. 2015.

CHEY, J. **Postmortem: Irrational Games' System Shock 2**. Disponível em: <http://www.gamasutra.com/view/feature/131813/postmortem_irrational_games_.php>. Acesso em: 15 nov. 2015.

CLEOPASCE, D. **A Resource Manager for Game Assets**. Disponível em: <http://www.gamedev.net/page/resources/_/technical/game-programming/a-resource-manager-for-game-assets-r3807>. Acesso em: 05 nov. 2015.

CRYTEK. **Cryengine**. Disponível em: <http://cryengine.com/get-cryengine>. Acesso em: 05 nov. 2015.

DUNN, F.; PARBERRY, I. **3D Math Primer for Graphics and Game Development**. 2. ed. Boca Raton: CRC Press, 2011. 807 p.

ELBERY, D. 3D **Game Enigine Design: a pratical approach to real-time computer graphics**. 2. ed. Boca Raton: CRC Press, 2007. 1017 (Série Morgan Kaufmann)

EPIC. **Unreal Engine Features**. Disponível em: <https://www.unrealengine.com/unreal-engine-4>. Acesso em: 05 nov. 2015.

EPIC. **What is Unreal Engine?** Disponível em <https://www.unrealengine.com>. Acesso em 05 nov. 2015.

ERICSON, C. **Real-Time Collision Detection**. San Francisco: Elsevier, 2005. 591 p. (Série Morgam Kaufmann)

GAME INSTITUTE. **Game Institute**. Disponível em: <www.gameinstitute.com>. Acesso em: 05 nov. 2015.

GAUL, R. **Component Based Engine Design**. Disponível em: <www.randygaul.net/2013/05/20/component-based-engine-design/>. Acesso em: 05 nov. 2015.

GERS, F. **Game Engine and Modules**. Disponível em <http://www.felixgers.de/teaching/game/game-modules-talk/GameEngineAndModules.html>. Acesso em 07 nov. 2015.

GRANBERG, C. **Character Animation with Direct3D**. Boston: Cenage Learning, 2009. 409 p. (Série Course Technology)

GREGORY, J. **Game Engine Architecture.** 2. ed. Boca Raton: CRC Press, 2015. 1018 p.

HEPTOMINO. **Virtua Game Engine**. Disponível em <http://www.heptomino.com/vge/>. Acesso em 07 nov. 2015.

ID SOFTWARE. **Doom 3 BFG Source Code**. Disponível em: <https://github.com/id-Software/DOOM-3-BFG>. Acesso em: 05 nov. 2015.

JORDAN, J. **Engines of Creation: An Overview of Game Engines**. Disponível em: <http://www.gamasutra.com/view/feature/132226/engines_of_creation_an_overview_.php>. Acesso em: 07 nov. 2015.

KUSHNER, D. Masters of Doom: how two guys created an empire and transformed pop culture. New York: Random House, 2003. 352p.

LAMBER, S. **An Introduction to Spritesheet Animation**. Disponível em: <gamedevelopment.tutsplus.com/tutorials/an-introduction-to-spritesheet-animation--gamedev-13099>. Acesso em: 05 nov. 2015.

LONG, E. **Enhanced NPC Behavior using Goal Oriented Action Planning**. 2007. 102 p. Dissertação (Mestrado em Tecnologia de Jogos Computacionais). Universidade de Abertay Dundee, Dundee.

LUNA, F. D. **3D Game Programming with DirectX 11**. Dulles: Mercury Learning and Information, 2012. 847 p. (Série Course Technology)

MCSHAFFRY, M; GRAHAM, D. **Game Coding Complete**. 4. ed. Boston: Cengage Learning, 2013. 959 p. (Série Course Technology)

MICROSFT. **Visual Studio**. Disponível em: <www.visualstudio.com>. Acesso em: 05 nov. 2015.

MICROSOFT. **DirectX Graphics and Gaming**. Disponível em <https://msdn.microsoft.com/en-us/library/windows/desktop/ee663274(v=vs.85).aspx>. Acesso em 05 nov. 2015.

MICROSOFT. **Download Microsoft XNA Game Studio 4.0.** Disponível em <www.microsoft.com/en-us/download/details.aspx?id=23714>. Acesso em 05 nov. 2015.

MICROSOFT. **Reference for HLSL.** Disponível em: <https://msdn.microsoft.com/pt-br/library/windows/desktop/bb509638(v=vs.85).aspx>. Acesso em: 05 nov. 2015.

MILLINGTON, I. **Artificial Intelligence for Games.** San Francisco: Elsevier, 2006. 835 p. (Série Morgan Kaufmann).

MILLINGTON, I. **Game Pysics Engine Development.** San Francisco: Elsevier, 2007. 447 p. (Série Morgan Kaufmann).

MONOGAME. **Write once, play everywhere.** Disponível em <www.monogame.net>. Acesso em 05 nov. 2015.

MONOLITH. **FEAR**. Disponível em: <store.steampowered.com/app/21090/>. Acesso em: 05 nov. 2015.

MORDETH. **Source Ports.** Disponível em: <http://www.doomworld.com/classicdoom/ports/>. Acesso em: 05 nov. 2015.

NOEL. **Start pre-allocation and stop worrying.** Disponível em: <http://gamesfromwithin.com/start-pre-allocating-and-stop-worrying>. Acesso em: 05 nov. 2015.

NVIDIA. **GPU Gems 2, Chapter 05.** Disponível em <http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter05.html>. Acesso em 07 nov. 2015.

NYSTROM, B. **Game Programming Patterns: Command.** Disponível em: <gameprogrammingpatterns.com/command.html>. Acesso em: 05 nov. 2015.

NYSTROM, B. **Game Programming Patterns: Event Queue.** Disponível em: gameprogrammingpatterns.com/event-queue.html>. Acesso em: 05 nov. 2015.

POD-Bot Introduction. Disponível em: <http://podbotmm.bots-united.com/doc_v3/index.html>. Acesso em: 05 nov. 2015.

RADICAL FISH GAMES. **Optimizing an HTML5 game engine using composition over inheritance.** Disponível em <http://www.radicalfishgames.com/?p=1725>. Acesso em 07 nov. 2015.

SANGLARD, F. **Doom3 Source Code Review: Renderer.** Disponível em <http://fabiensanglard.net/doom3/renderer.php>. Acesso em 14 nov. 2015.

Source Engine Licensing information sheet. Disponível em: <www.valvesoftware.com/SOURCE_InfoSheet.pdf>. Acesso em: 05 nov. 2015.

SPOLSKY, J. **How Microsoft Lost the API War.** Disponível em: <http://www.joelonsoftware.com/articles/APIWar.html>. Acesso em: 05 nov. 2015.

Sprite. Disponível em: <doom.wikia.com/wiki/Sprite>. Acesso em: 05 nov. 2015.

The Industry's Foundation for High Performance Graphics. Disponível em <https://www.opengl.org>. Acesso em 05 nov. 2015.

UNITY 3D. **Fórum Unity 3D.** Disponível em: <forum.unity3d.com>. Acesso em: 05 nov. 2015.

UNITY 3D. **Obtenha o Unity.** Disponível em: <https://unity3d.com/pt/get-unity>. Acesso em: 05 nov. 2015.

UNITY 3D. **Unity 3D.** Disponível em <https://unity3d.com>. Acesso em: 05 nov. 2015.

UNITY 3D.Scripting Tutorial. Disponível em: <https://unity3d.com/pt/learn/tutorials/topics/scripting>. Acesso em: 05 nov. 2015.

UNIVERSITY OF CALIFORNIA. **Skin**. Disponível em: <http://graphics.ucsd.edu/courses/cse169_w05/3-Skin.htm>. Acesso em: 05 nov. 2015.

VALVE SOFTWARE. **GoldSrc Engine.** Disponível em: <half-life.wikia.com/wiki/GoldSrc>. Acesso em: 05 nov. 2015.

VALVE SOFTWARE. **Source Engine Features.** Disponível em: <https://developer.valvesoftware.com/wiki/Source_Engine_Features>. Acesso em: 05 nov. 2015.

VALVE. **Choreography creation.** Disponível em: <https://developer.valvesoftware.com/wiki/Choreography_creation>. Acesso em: 05 nov. 2015.

VEAZIE, B. J. **Skinned Mesh Animation Using Matrices.** Disponível em: <www.gamedev.net/page/resources/_/technical/graphics-programming-and-theory/skinned-mesh-animation-using-matrices-r3577>. Acesso em: 05 nov. 2015.

WEST, M. **Evolve Your Hierarchy.** Disponível em: <http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/>. Acesso em Acesso em 07 nov. 2015.

What are common rendering optimization techniques […] ?. Disponível em: <gamedev.stackexchange.com/questions/66280/what-are-the-common-rendering-optimization-techniques-for-the-geometry-pass-in-a>. Acesso em: 05 nov. 2015.

YOYOGAMES. **Buy GameMaker Studio.** Disponível em: <www.yoyogames.com/studio/buy>. Acesso em 05 nov. 2015.