URGE: Motor para o Desenvolvimento de Jogos Eletrônicos

Alexandre Augusto Abdalla de Oliveira Cardoso
Vitor Carneiro Maia

25 de abril de 2012

Monografia apresentada para obtenção do Grau de Bacharel em Ciência da Computação pela Universidade Federal do Rio de Janeiro.

Orientador:
Prof. Ph.D. Rodrigo Penteado R. de Toledo

Curso de Bacharelado em Ciência da Computação
Universidade Federal do Rio de Janeiro

Rio de Janeiro, Brasil

25 de abril de 2012

Projeto Final de Curso submetido ao Departamento de Ciência da Computação do Instituto de Matemática da Universidade Federal do Rio de Janeiro como parte dos requisitos necessários para obtenção do grau de Bacharel em Ciência da Computação.

Autores do Trabalho:

Alexandre Augusto Abdalla de O. Cardoso

Vitor Carneiro Maia

Aprovado por:

Prof. Ph.D. Rodrigo Penteado R. de Toledo
Universidade Federal do Rio de Janeiro,
Orientador

Prof. D.Sc. João Carlos Pereira da Silva
Universidade Federal do Rio de Janeiro,
Avaliador

Prof. Ph.D. Adriano Joaquim de Oliveira Cruz
Universidade Federal do Rio de Janeiro,
Avaliador

Prof. D.Sc. Esteban Walter Gonzalez Clua
Universidade Federal Fluminense, Avaliador

Resumo

Este trabalho apresenta um novo motor para o desenvolvimento de jogos eletrônicos tri-dimensionais. A URGE, integralmente idealizada e desenvolvida por quatro alunos do curso de Ciência da Computação da UFRJ, introduz novos paradigmas na programação de um *game*. Apesar de fornecer recursos para qualquer tipo de jogo, seu foco é a produção de jogos 3D em cenários a céu aberto. Para atingir tal objetivo, a ferramenta baseia-se em seus dois elos: o núcleo de visualização e o núcleo de física. Ambos são responsáveis por visualizar esimular, de forma fiel à realidade, uma extensa gama de materiais e toda a interação dinâmica entre corpos baseando-se nas leis da física. A URGE foi preparada para funcionar em computadores de diferentes capacidades de processamento; para isso, ela conta com um sistema de gerenciamento de cena, criado para eliminar o máximo de *overhead*, garantindo seu funcionamento em tempo real. Ao longo da dissertação, é discutido todo o processo de criação, incluindo as técnicas envolvidas no desenvolvimento de cada módulo do motor, e a evolução modelagem do projeto ao longo do tempo. Por fim, apresentamos as experiência da URGE em relação ao público aberto, como mini-cursos, trabalho na disciplina de computação gráfica e palestras sobre desenvolvimento de jogos.

Abstract

This work presents a new engine for the development of three-dimensional computer games. The URGE, which is fully designed and developed by four Computer Science students, UFRJ, introduces new paradigms in game programming. In spite of providing resources for any type of game, your focus is the production of 3D games under the open scenarios. To achieve this goal, the engine is based on its two cores: The Visualization System and the Physics System. Both are responsible for faithfully simulating the reality of an extensive range of materials in nature, and all the dynamic interaction between bodies based on the laws of physics. URGE was prepared to work under different computer types. For that, she has a scene management system, designed to eliminate overheads as much as possible, to ensure real time working. In this work, we discuss the entire build process, including techniques involved in the development of each module, and the development of this project shaping over the time. Finally, we will present the URGE experiences with the open public, such as mini-courses, work in the discipline of computer graphics and lectures on game development.

Sumário

Lista de Figuras

1 Introdução

1.1 Atual Paradigma da Programação de Jogos

Atualmente, com as tecnologias disponíveis, é possível desenvolver Jogos Eletrônicos com uma agilidade que antes não existia [LaM95]. Agora não é mais preciso criar um jogo a partir de recursos de programação em baixo nível, como bibliotecas ou APIs (Interface de Programação de Aplicativos). Uma ferramenta intermediária denominada Motor de Jogos, popularmente conhecida como *engine*, substituiu a necessidade do desenvolvimento de vários recursos no campo da computação gráfica, simulação física, inteligência artificial e tele-processamento em redes, existentes em qualquer aplicação interativa em tempo Real. As *engines* encapsulam uma série de artifícios necessários na programação de qualquer jogo eletrônico tornando muito mais ágil e simples o esforço feito em sua criação.

1.2 Algumas Engines Atuais

É possível encontrar varias engines, proprietárias e gratuitas, pela internet sem a menor dificuldade. Contudo, qual poderia ter artifícios mais avançados? Ou melhor custo? Ou até maior completude de recursos? Nas seguintes subseções, iremos analisar algumas delas.

1.2.1 Unity 3D: Simplicidade e Alto Realismo

Dos diversos motores existentes, a Unity 3D (ou simplesmente Unity) possui uma das interfaces gráficas mais simples e intuitivas, além de suporte a avançados artifícios de computação gráfica e simulação física. Com apenas um pressionar de checkbox o usuário é capaz de habilitar sombras, modificar propriedades gráficas de um objeto específico ou ativar um sistema de detecção de colisão física continua, na qual poucas engines compartilham tal recurso. Contudo o grande problema é o fato dela ser proprietária. Apesar

de possuir uma licença gratuita, caso o desenvolvedor possua uma certa margem de lucro deverá, obrigatoriamente, comprar uma licença de $ 1500 dólares. A Unity, ainda, possui extensões, como suporte para o funcionamento em dispositivos moveis, nas quais são pagas a parte e encarecem o preço do produto.

1.2.2 Cryengine: Realismo de Cinema

É muito difícil e controverso afirmar uma engine como sendo a mais visualmente realista. Mesmo assim, é perfeitamente comum cair na tentação de acreditar não existir gráficos mais bem sofisticados que os da Cryengine [Mit07]. Originalmente criada para um jogo em primeira pessoa (Far Cry), atualmente aplica-se a basicamente qualquer jogo 3D de alto nível de realismo. Seu nível de realismo é tão elevado que a Crytek (desenvolvedora do motor) pretende entrar no mercado de filmes de animações 3D com a ferramenta: Cryengine 3 Sandbox for Cinema, capaz de gerar animações 3D realistas em tempo real. Além disso, possui uma licença gratuita, caso o seu uso seja para projetos sem fins lucrativos. Entretanto, por ser uma engine tão especializada, não é recomendada para sistemas menos sofisticados como celulares ou até computadores menos potentes, além da licença para produção de jogos comerciais possuir um preço relativamente elevado.

1.2.3 OGRE: Gráficos Elaborados de graça

Object-oriented Graphics Rendering engine, ou OGRE, é uma ferramenta gratuita totalmente baseada em padrões de programação orientada a objetos. Programadores experientes são capazes de manipulá-la sem nenhuma dificuldade pois a engine é fundamentada em padrões de projetos de engenharia de software como: Singleton, Abstract Factory e Template Method. Seus gráficos possuem um nível de realismo considerável, e pode ser incorporado por um computador de baixo rendimento. O defeito da OGRE reside no fato dela não ser completa. Como o próprio nome diz, é um motor exclusivamente "gráfico", não possui, por exemplo, nenhum recurso de simulação física. Outro defeito, reside no fato do seu aprendizado poder ser relativamente lento e trabalhoso para usuários menos experientes, como ela é bastante fiel a padrões de projetos, seu aprendizado uso está intimamente ligado ao conhecimento de Engenharia de Software.

1.2.4 Irrlicht engine: União de Recursos sem preço

A Irrlicht é a prova de que um motor open-source pode reunir os mais diversos artifícios [Die09]. A irrlicht é capaz de suportar, sem o uso de plugins, mais de vinte formatos de modelos tri-dimensionais, dez formatos de texturas, diversas formas de cálculos de transparências e mapeamentos. A Irrlicht é totalmente compatível com o Copper Cube, um editor visual gratuito na qual facilita o trabalho no desenvolvimento cenários de um jogo. Mesmo assim, a Irrlicht não é uma ferramenta simples de se manipular. Recursos avançados de programação não são encapsulados, o que obriga o conhecimento dos mesmos por parte dos usuários da engine.

1.3 Por que desenvolver a URGE?

Conforme dito anteriormente, os atuais motores disponibilizam uma série de utilidades que simplificam consideravelmente o desenvolvimento de um jogo. Entretanto, todos os grandes motores como Unity, Source engine, Cry engine, U.D.K, Torque e C4, são estrangeiros e pagos ou presas a licenças comerciais de preço elevado. Poucos motores foram lançados gratuitamente e sem restrições, tais como O.G.R.E., Irrlicht engine, Panda3d e jMonkey, porém requerem um nível técnico mais elevado para sua plena manipulação e, consequentemente, um aprendizado mais complexo e demorado. Além disso, esses não possuem todos os recursos usados na programação de um jogo, sendo necessário o uso de mais de uma engine em paralelo. A proposta da URGE (Unified Resources GDP/Game engine) baseia-se em dois princípios: completude e simplicidade.

A URGE teve como motivação, a reunião das principais técnicas de computação gráfica aplicada em jogos eletrônicos, simulação física em tempo real e tele-processamento em redes em uma só ferramenta, atentado ao fato de trabalhar em computadores de baixo desempenho. Ela possui diversos módulos que fornecem o maximo de suporte para a criação de jogos 3D com cenários abertos, isso significa que seu foco, quanto à complutude, é a simulação, em tempo real e alta qualidade, de ambientes tri-dimensionais a céu aberto.

A simplicidade pode ser observada pela forma intuitiva de se programar na URGE, inclusive a ferramenta já foi utilizada por mais de cinquenta alunos do curso de graduação na qual mais de 75% dos que participaram de uma pesquisa (vide Seção 7.1), confirmaram a facilidade de se programar. Uma outra vantagem da ferramenta é o fato de ter sido criada por programadores brasileiros. Isso facilita muito o aprendizado e *feedback* por parte de todos os usuários do país.

1.4 Resumo dos Recursos da URGE

A URGE possui os seguintes recursos técnicos:

- Carregamento e Animação de modelos Tridimensionais;
- Carregamento de Imagens e Texturas em diversos formatos – bmp, png, jpg, pcx, tga e lbm;
- Geração automática de Terrenos;
- Auto-Geração de primitivas Tridimensionais: Cubo, Esfera, Plano e Cilindro;
- Simulação Física 3D baseada em Impulso;
- SkyDomes: SkyBox e SkySphere;
- Dinâmica de Partículas e auto-geração das próprias imagens;
- Técnicas de *Shading* seletivo de acordo com a configuração do computador em questão – Flat Shading, Gouraud Shading, Phong Shading ou Blinn-Phong Shading;
- Bump Mapping e Parallax Occlusion Mapping;
- Simulação de Reflexão e Refração com dispersão cromática;
- Carregamento de Modelos 2D com transparência (mascara);
- Sáida de Texto;
- Captura de Entrada – teclado ou mouse;
- Gerenciamento de cena - Fustrum Culling, Quadtrees e Octrees;
- Carregamento de Som e Música – wav ou ogg;
- Celshading – Visualização Não Foto-Realística;
- Simulação de Oceanos;
- Conexão com a internet via Sockets – TCP ou UDP e Suporte a MultiCasting .

1.4.1 Estrutura da Dissertação

Na etapa de projeção da URGE, todos os seus recursos foram agrupados em três grandes seções: visualização (*rendering*), simulação física e gerenciamento de cena. As seções de Visualização e Simulação Física representam os dois elos fundamentais da engine, ambos são completamente independentes entre si e integrados exclusivamente pelo gerenciador de cena.

Os Capítulos 2, 3 e 4 compõem a primeira parte do trabalho, nela é apresentado as diversas técnicas envolvidas no processo de criação dos respectivos três elos da engine: visualização, simulação física e gerenciamento de cena.

A segunda parte do Trabalho é dedicada ao desenvolvimento e estrutura do projeto, ela é composta por dois capítulos, 5 e 6.
O Capítulo 5 aborda todos detalhes sobre o processo de desenvolvimento da URGE, tais como os conceitos de engenharia de *software* que foram aplicados, as modelagens do projeto e os testes de uso por parte de público externo.
No Capítulo 6 encontra-se a descrição da estrutura e o funcionamento interno da engine. Em outras palavras, como ela é organizada de forma a possibilitar que um desenvolvedor de jogos consiga, sem esforços, manuseá-la com o intuito de se criar aplicações interativas de alta qualidade.

Por fim, o Capítulo 7 discute sobre a experimentação da URGE ao público, e realiza o desfecho da dissertação fazendo um breve apanhado geral, explicitando o que as qualidades URGE e o que ainda falta melhorar.

Parte I

Técnicas e Conceitos Envolvidos

2 *Visualização: Mapeamentos, Iluminação e Efeitos*

O primeiro elo da URGE é seu núcleo de visualização (sistema de rendering) que envolve técnicas de mapeamentos de texturas, iluminação em tempo real e efeitos visuais. Tal sistema foi criado com dois propósitos: simular a visualização de uma extensa gama de materiais presentes em jogos (superfícies reflexivas, rugosas, refrativas, não foto-realísticas...) e gerar gráficos de alto nível de realismo em tempo real, considerando o limite da capacidade de processamento gráfico do respectivo computador. Neste capítulo entenderemos as técnicas de programação envolvidas no desenvolvimento do sistema de visualização da URGE.

2.1 Programação de Shaders

Conforme dito anteriormente, um dos principais objetivos do sistema de visualização da URGE é gerar gráficos de alta qualidade. Isso já implica em um problema inicial: O *pipeline* padrão do OpenGL, conhecido como pipeline fixo, não dispõe a grande parte dos recursos para a programação de técnicas que possibilitam a geração de gráficos que atendam nosso objetivo.

Para tal, foi necessário que os programadores da URGE migrassem para um pipeline secundário do OpenGL: O pipeline programável. Repare, segundo a Figura 2, que os módulos de processamento de vértices (*Vertex*) e processamento de pixels (*Fragment*) foram convertidos para *Vertex Shader* e *Fragment Shader*, respectivamente, isso significa que tais módulos agora são programáveis. Em outras palavras, segundo o novo pipeline, todo o processamento de vértices e pixels está aberto a ser programado pelo desenvolvedor OpenGL.

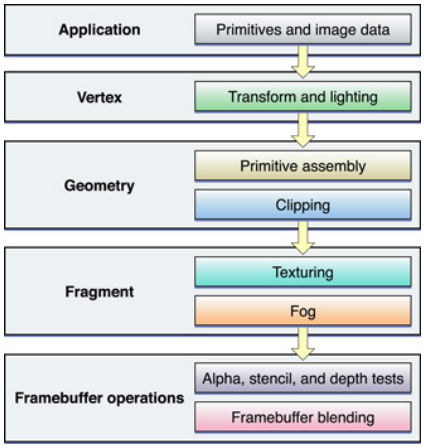

Figura 1: Fluxograma Representativo do pipeline fixo do OpenGL

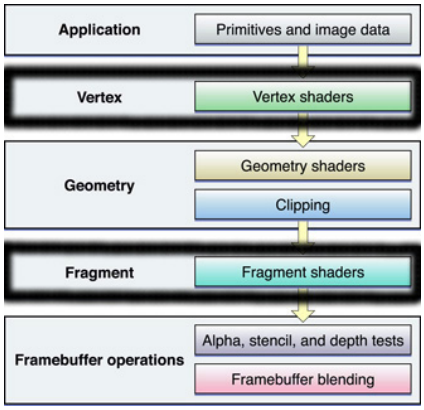

Figura 2: Fluxograma Representativo do pipeline programável do OpenGL

Essa diferença no pipeline é extremamente vantajosa, pois o programador agora pode implementar suas próprias técnicas de visualização, isto é, programar *shaders* (no caso: Vertex Shader e Fragment Shader). Por exemplo, no pipeline fixo, as técnicas de ilumina-ção já são implementadas pelo próprio OpenGL (*Flat Shading* ou *Gouraud Shading*), por isso o nome. Já no pipeline programável, fica sob responsabilidade do desenvolvedor da engine implementar técnica de iluminação desejada, isso implica numa considerável liber-dade que, no caso da URGE, possibilitou a implementação do *Phong Shading* (iluminação realizada pixel a pixel, traduz-se em uma melhor qualidade visual).

Ha pouco tempo, a programação de shaders era suportada apenas por computadores dotados de unidade de processamento gráfico, afinal estimou-se a necessidade de passar a responsabilidade de grande parte do processamento gráfico para tais unidades para assim poder atingir um elevado nível de qualidade gráfica. Contudo, atualmente computadores sem tal unidade já possuem suporte a esse recurso.

A primeira linguagem considerada padrão pelo OpenGL para programação em shaders

foi criada em 2002. O ARB-Assembly (OpenGL Architectural Review Board Assembly) é uma linguagem de baixo nível presente a partir da versão 1.5 da OpenGL (a URGE possui suporte para a compilação e uso de tal linguagem) [Gre04]. Contudo, sua manipulação é complexa e trabalhosa por se tratar de uma linguagem de baixo nível de programação, além de possuir uma série de restrições quanto ao tamanho do shader (inicialmente, os programas shaders não funcionavam além de um certo numero de instruções) e a falta de laços de repetições (não suportava instruções de desvio de código).

```
ATTRIB     vPos  = vertex.position;
ATTRIB     vNorm = vertex.normal;
ATTRIB     vCol  = vertex.color;
OUTPUT     oPos  = result.position;
OUTPUT     oCol  = result.color;
PARAM CTM[4]     = { state.matrix.mvp };
PARAM IMV[4]     = { state.matrix.modelview.invtrans };
PARAM lVec  = program.env[1];
PARAM lHalf = state.light[0].half;
TEMP  eyeNorm, coeff, shade;
PARAM red = { 1, 0, 0, 1 };
DP4   oPos.x, CTM[0], vPos;
DP4   oPos.y, CTM[1], vPos;
DP4   oPos.z, CTM[2], vPos;
DP4   oPos.w, CTM[3], vPos;
DP3   eyeNorm.x, IMV[0], vNorm;
DP3   eyeNorm.y, IMV[1], vNorm;
DP3   eyeNorm.z, IMV[2], vNorm;
DP3   eyeNorm.w, eyeNorm, eyeNorm;
RSQ   eyeNorm.w, eyeNorm.w;
MUL   eyeNorm, eyeNorm, eyeNorm.w;
MUL   shade, state.lightmodel.ambient, vCol;
DP3   coeff.x, lVec, eyeNorm;
DP3   coeff.y, lHalf, eyeNorm;
MOV   coeff.w, state.material.shininess.x;
LIT   coeff, coeff;
MAD   shade, coeff.y, vCol, shade;
MAD   shade, coeff.z, vCol, shade;
MOV   oCol, shade;
END;
```

Figura 3: Exemplo de Simples Vertex Shader em ARB-Assembly que produz uma Iluminação via Gouraud Shading

Posteriormente, em 2004, a OpenGL lançou sua primeira Linguagem Shader de Alto Nível de Programação, a *OpenGL Shading Language* (GLSL), na qual é usada em larga escala até os dias de hoje [RLKG+09]. GLSL assemelha-se bastante com C, e esse era o objetivo, pois facilita seu aprendizado. Atualmente na versão 4.10 possui vários recursos para facilitar ao maximo sua manipulação e permitir implementações de novas técnicas de computação gráfica. Um bom exemplo de sua completude é o tipo de váriavel *SamplerCube*

que representa uma váriavel capaz de armazenar seis texturas correspondentes as faces de um cubo, o que é muito usado em simulação de superfícies reflexivas. Repare na Figura 4 a maior intuitividade do código em relação ao ARB-Assembly.

```
uniform vec3 eye;

void main( void )
{
    vec3 N = normalize( gl_NormalMatrix * gl_Normal );
    vec3 V = ( gl_ModelViewMatrix * gl_Vertex ).xyz;
    vec3  L = normalize( gl_LightSource[0].position.xyz - V );
    vec3  H = normalize( L + eye);
    float c_diffuse = max( 0.0, dot( N,L ) );
    float c_specular = pow( max( 0.0, dot( N,H ) ), gl_FrontMaterial.shininess );

    gl_Position = ftransform();
    gl_FrontColor =
        gl_FrontMaterial.ambient  * gl_LightSource[0].ambient +
        gl_FrontMaterial.diffuse  * gl_LightSource[0].diffuse  * c_diffuse +
        gl_FrontMaterial.specular * gl_LightSource[0].specular * c_specular;
}
```

Figura 4: Exemplo de Simples Vertex Shader em GLSL que produz uma Iluminação via Gouraud Shading

Infelizmente GLSL está disponível apenas em computadores que possuam suporte para versões do OpenGL iguais ou superiores a 2.0. A URGE é capaz de compilar e executar programas de Vertex Shader ou Fragment Shader em GLSL. Inclusive, ela possui programas shaders nativos com a implementação das técnicas de computação gráfica, descritas nas proximas seções, para gerar imagens de alta qualidade.

2.2 Iluminação de Materias

O OpenGL já dispõe de um sistema de Iluminação bem simples de se entender, mas capaz de reproduzir gráficos de alta qualidade. Mesmo assim, o sistema apresenta falhas quando a iluminação é aplicada a superfícies com poucos vértices, como nosso objetivo é gerar aplicações em tempo real, é ineficiente ampliar o número de vértices com o intuito de reparar tal problema. Por isso, os programadores da URGE precisaram tirar proveito do uso de programas shaders para potencializar a qualidade do sistema de iluminação da URGE. Nos próximos tópicos, entenderemos como foi implementado tal sistema.

2.2.1 As Componentes de Iluminação

Dado uma superfície e um foco de luz, o OpenGL é capaz de simular o efeito de iluminação sobre tal superfície separando-o em três componentes que juntas são capazes de simular superfícies foscas como madeira ou borracha e superfícies reflexivas como metais. Tais componentes são: ambiente, difusa e especular.

A componente ambiente simula toda a luz aplicada de forma indireta sobre um objeto, como se os raios de luz que chegassem ao objeto em questão foram refletidos por outras superfícies em sua volta. O resultado visual é o objeto iluminado de forma homogênea sendo difícil distinguir se tal objeto é tridimensional ou não, apenas conseguimos identificar suas bordas.

A componente difusa representa os raios de luz que incidiram diretamente sobre o objeto em questão. Seu efeito visual, é a plena impressão tridimensional do objeto do objeto em questão e também de suas faces iluminadas em relação ao foco de luz.

A componente especular representa apenas os raios de luz que foram refletidos pela superfície e incididos diretamente no "olho" do observador. O efeito é de brilhos em partes específicas do objeto que podem mudar de acordo com a posição do observador, inclusive é a unica componente que depende da posição do observador.

2.2.2 Modelos de Iluminação: Flat, Gouraud, Phong e Blinn-Phong

No pipeline fixo do OpenGL, as componentes de iluminação são combinadas de acordo com uma simples equação. Antes de introduzir essa equação, é importante entender que

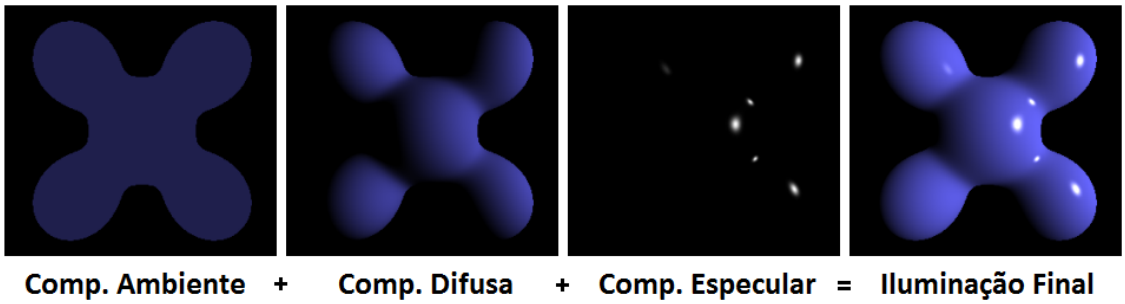

Figura 5: As três componentes de Iluminação e seu efeito final combinado

ela é baseada na seguinte premissa: Dado um ponto sobre superfície, considere $\overrightarrow{N}$ como sendo o vetor normal à superfície no ponto em questão, $\overrightarrow{E}$ como sendo o vetor criado a partir da diferença entre a posição do ponto e a do observador, $\overrightarrow{L}$ como sendo o vetor criado a partir da diferença entre a posição do ponto e a da luz e finalmente $\overrightarrow{H}$ como sendo a média entre os vetores $\overrightarrow{E}$ e $\overrightarrow{L}$ (conhecido como *Half-Vector*). As componentes difusa e especular são calculadas, respectivamente, a partir do ângulo entre $\overrightarrow{L}$ e $\overrightarrow{N}$, e no caso da especular, o ângulo entre $\overrightarrow{E}$ e $\overrightarrow{N}$. Já a componente ambiente é constante em relação a superfície, portanto não depende de nenhuma váriavel citada. A Figura 6 ilustra essas informações.

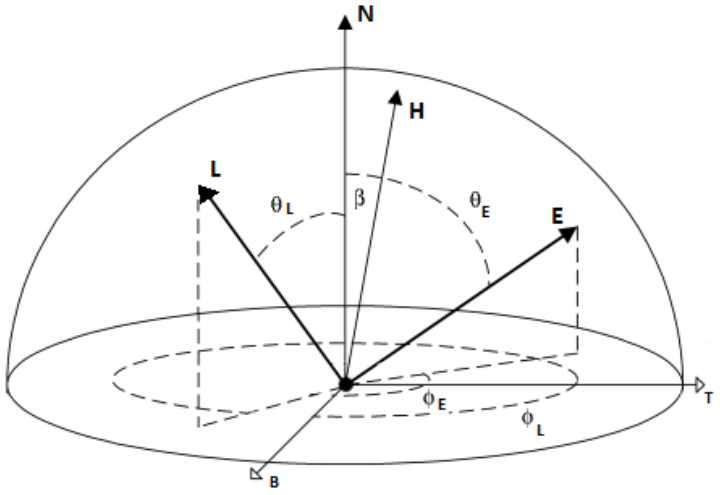

Figura 6: Ilustração geométrica dos vetores envolvidos no cálculo da iluminação, T e B são os vetores tangentes a Normal

Considere os vetores citados acima e os seguinte vetores: *MatAmb*, *MatDif* e *MatEsp* como sendo o fator de reação por parte da superficie à componente ambiente, difusa e especular, respectivamente, da luz emitida pelo foco em questão. Além disso, considere

LuzAmb, *LuzDif* e *LuzEsp* como sendo a intensidade das três componentes emitidas pelo foco de luz (repare que os vetores são tri-dimensionais e suas coordenadas representam o canal RGB). E por último, considere a variável escalar *Brilho* como sendo o quanto acentuado é a centelha da componente especular. O OpenGL, calcula a luz em uma superfície de acordo com a seguinte equação [WNDS99b]:

$$\begin{aligned} CorFinal &= MatAmb \otimes LuzAmb \\ &+max\left(\overrightarrow{N}\cdot\overrightarrow{L},0\right) * MatDif \otimes LuzDif \\ &+max\left(\overrightarrow{N}\cdot\overrightarrow{H},0\right)^{Brilho} * MatEsp \otimes LuzEsp \end{aligned}$$

Lembre que os vetores $\overrightarrow{N},\overrightarrow{L}$ e $\overrightarrow{H}$ foram definidos em relação a um ponto numa superfície. Claramente, é computacionalmente impossível aplicar tais vetores em todos os pontos de uma superfície de forma infinitesimal, contudo devemos aproximar o domínio de pontos a um conjunto finito. Se tal conjunto representar as faces de uma superfície, ou seja, se esses vetores forem calculados em relação a cada face do objeto estaremos aplicando o chamado *Flat Shading*. Contudo, se interpolarmos todos os vetores relacionados às faces para cada vértice da superfície, criaremos o chamado *Gouraud Shading* [Gou71]. E por último, se interpolarmos, pixel a pixel, todos valores relacionados aos vértices do objeto, podemos estender tal equação ao espaço na qual apresenta maior precisão, o espaço dos pixels, atualmente a forma mais realista na simulação de iluminação, isso se chama *Blinn-Phong Shading* [Bli77] e evidentemente deve ser aplicada pelo Fragment Shader.

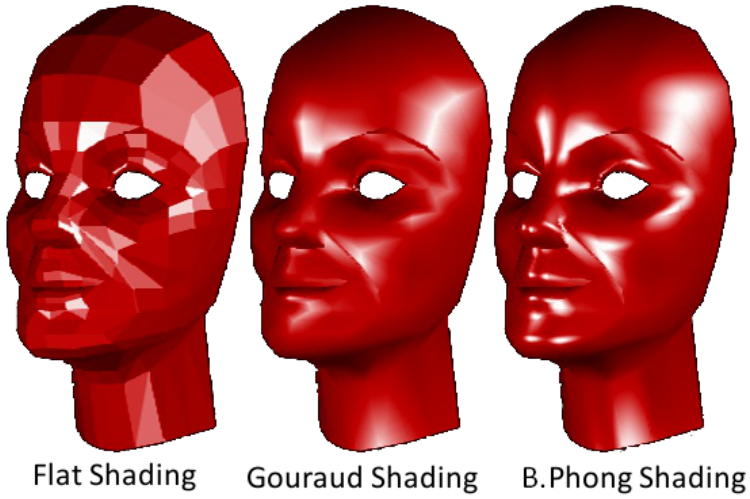

Figura 7: Comparação de Qualidade Visual: Flat, Gouraud e Blinn-Phong

É evidente que o Blinn-Phong Shading possui uma qualidade mais elevada, contudo a um custo de processamento maior. Já o Flat Shading apesar de ser o mais simples pos-

sui uma qualidade bastante inferior. É responsabilidade da engine administrar a escolha do melhor método de iluminação, a URGE, por exemplo, possui uma opção de ajuste de qualidade na qual pode-se migrar entre os três métodos para melhor se adaptar ao computador em questão. O OpenGL suporta apenas o Flat Shading e o Gourarud Shading pelo seu pipeline fixo, a única forma de usar o Blinn-Phong é programando Shaders através do pipeline programável. No caso da URGE, além dos três Modelos, há suporte para mais um modelo chamado *Phong Shading*, ele é quase similar ao Blinn-Phong Shading, a única diferença, é o cálculo da especular, que no caso é dado pelo produto escalar entre a $\overrightarrow{E}$ e um vetor resultante da reflexão de $\overrightarrow{L}$ em torno de $\overrightarrow{N}$, em quanto no modelo Blinn-Phong, a especular é aproximada pelo half-vector. Observe a equação do Modelo Phong [Pho75]:

$$\begin{aligned} CorFinal &= MatAmb \otimes LuzAmb \\ &+max\left(\overrightarrow{N} \cdot \overrightarrow{L}, 0\right) * MatDif \otimes LuzDif \\ &+max\left(\left(\overrightarrow{L} - 2\left(\overrightarrow{L} \cdot \overrightarrow{N}\right) * \overrightarrow{N}\right) \cdot \overrightarrow{E}, 0\right)^{Brilho} * MatEsp \otimes LuzEsp \end{aligned}$$

2.3 Normal Mapping

Em jogos, muitas vezes queremos representar, de forma fiel à realidade, matérias que possuem uma aparência áspera, irregular ou até mesmo esburacada, como é o caso de tijolos, mármore, pedras e outros. Além disso, também é comum encontrar modelos tri-dimensionais com um alto nível de detalhes. Por exemplo, imagine um modelo 3D de um laptop, caso ele se encontre dentro de um cenário, provavelmente não será um objeto muito grande mas possui muitos detalhes, como suas teclas que são menores ainda. Se quisermos retratar tais detalhes com fidelidade, precisamos estender o número de vértices do objeto, o que seria ineficiente pois modelos com muitos vértices devem ser evitados em aplicações de tempo real por questão de performance.

Felizmente, há algoritmos que podem simular fielmente todos esses casos de materiais com um alto nível de detalhe ou de aparência irregular. O algoritmo aqui abordado é o *Bump Mapping* ou *Normal Mapping* [KL96], que significa distorcer as normais de um modelo, pixel a pixel, de acordo com uma função ou textura que chamamos de mapa de normal (*normal map*).

Nesta seção, será apresentado alguns algoritmos que potencializam a qualidade gráfica e aumentam consideravelmente o nível de realismo sem prejudicar a performance de um jogo.

2.3.1 O Espaço Tangente

Quando trabalhamos usando gráficos tri-dimensionais, frequentemente migramos de um espaço vetorial para outro, com intuito de realizar operações que dependem de um específico espaço, como o espaço de objeto, o espaço de imagem ou o espaço de cena. Contudo, o bump mapping baseia-se em um novo espaço vetorial: o Espaço Tangente (também chamado de Espaço de Textura, mas para não causar confusão com o espaço de imagem, optamos pelo primeiro nome). Como qualquer espaço vetorial tri-dimensional, o espaço tangente precisa de três vetores que compõem sua base, e esses são chamados de: normal, tangente e binormal (alguns matemáticos usam o termo bitangente, porém em programação de jogos o outro termo é mais comum). Esse espaço está associado a cada vértice de um Modelo, isto é, cada vértice possui uma normal, binormal e tangente. Observe a representação visual do espaço tangente pela Figura 8.

Quando dizemos que o mapeamento de normal está associado a tal espaço vetorial,

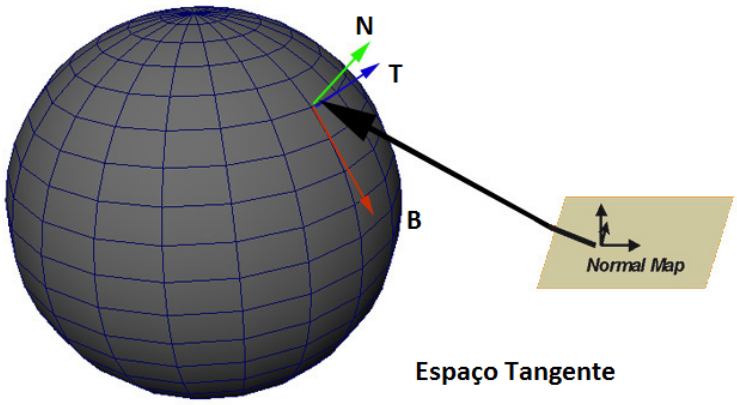

Figura 8: O sistema de Coordenadas do Espaço Tangente de um Esféra

significa dizer que antes de aplicar qualquer algoritmo dessa seção devemos multiplicar-mos certos vetores envolvidos no cálculo do bump mapping pela matriz composta pelos três vetores base, tal matriz é chamada de Matriz TBN.

$$TBN = \begin{bmatrix} T_X & T_Y & T_Z & 0 \\ B_X & B_Y & B_Z & 0 \\ N_X & N_Y & N_Z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Existem diversos algoritmos para descobrir o espaço tangente associado a um vértice de um modelo arbitrário. Na implementação da URGE, optou-se por um algoritmo mais elaborado, porém mais custoso em termos de processamento [Len11]. Mesmo assim, esse algoritmo é efetuado apenas etapa de criação de um objeto visual e armazenado na memória, dessa forma não há nenhum prejuízo no processamento gráfico durante o jogo.

2.3.2 Calculando o Espaço Tangente de um Modelo Arbitrário

Para descobrir o espaço tangente de cada vértice de um modelo 3D arbitrário, precisa-mos descobrir seus vetores base. Esses vetores estão associados a três informações básicas de um modelo 3D: normal, coordenada de textura e, evidentemente, vértice.

A Normal pode ser encontrada facilmente calculando-a para cada uma de suas faces, usando o produto vetorial entre cada dois vetores que compõem um plano tangente à face específica do modelo. Posteriormente, para cada vértice, basta interpolar de forma ponderada (em relação ao ângulo da face) todas as normais das faces que o vértice em

questão pertence.

O problema está em encontrar os outros dois vetores perpendiculares a normal do vértice. Entenda que uma base não ortogonal pode gerar um mapeamento distorcido, e consequentemente, um bump mapping incorreto. Lembre-se que os vetores base estão associados, também, às coordenadas de textura, isso significa dizer que o vetor tangente $\overrightarrow{T}$ deve ser corresponde à coordenada s (também chamada de u) e o vetor binormal $\overrightarrow{B}$ corresponde à coordenada t (também chamada de v) do mapa de normal. Sendo assim, considere $\overrightarrow{P}$ como sendo o vetor posição de um ponto arbitrário de uma face triangular formada pelos vértices $\overrightarrow{V_1}$, $\overrightarrow{V_2}$ e $\overrightarrow{V_3}$ de um modelo 3D. Podemos deduzir a seguinte formula pela interpolação de suas coordenadas de textura:

$$\overrightarrow{P} - \overrightarrow{V_N} = (s - s_N) \times \overrightarrow{T} + (t - t_N) \times \overrightarrow{B} \tag{2.1}$$

Onde N = 1,2 ou 3 e (s_N, t_N) são as coordenadas de textura associada ao vértice N.
Agora, precisamos calcular os dois vetores ($\overrightarrow{p}$ e $\overrightarrow{q}$) que formam o plano tangente a uma específica face de um modelo 3D:

$\overrightarrow{p} = \overrightarrow{V_1} - \overrightarrow{V_0}$
$\overrightarrow{q} = \overrightarrow{V_2} - \overrightarrow{V_0}$

Isso significa afirmar que:

$(s_p, t_p) = (s_1 - s_0, t_1 - t_0)$
$(s_q, t_q) = (s_2 - s_0, t_2 - t_0)$

Portanto, para descobrir os vetores $\overrightarrow{T}$ e $\overrightarrow{B}$ precisamos resolver o seguinte sistema de equações:

$\overrightarrow{p} = s_p \times \overrightarrow{T} + t_p \times \overrightarrow{B}$
$\overrightarrow{q} = s_q \times \overrightarrow{T} + t_q \times \overrightarrow{B}$

Podemos, re-escrever esse sistema para a forma matricial com o intuito de facilitar a visibilidade da resolução:

$$\begin{bmatrix} p_X & p_Y & p_Z \\ q_X & q_Y & q_Z \end{bmatrix} = \begin{bmatrix} s_p & t_p \\ s_q & t_q \end{bmatrix} \times \begin{bmatrix} T_X & T_Y & T_Z \\ B_X & B_Y & B_Z \end{bmatrix}$$

Por último, podemos multiplicar ambos os lados pelo inverso da matriz (s,t) para por a matriz (T,B) em evidência:

$$\begin{bmatrix} T_X & T_Y & T_Z \\ B_X & B_Y & B_Z \end{bmatrix} = \frac{1}{s_p \times t_q - s_q \times t_p} \times \begin{bmatrix} t_q & -t_p \\ -s_q & t_p \end{bmatrix} \times \begin{bmatrix} p_X & p_Y & p_Z \\ q_X & q_Y & q_Z \end{bmatrix}$$

O resultado dessa equação matricial nos fornece $\overrightarrow{T}$ e $\overrightarrow{B}$ não normalizados. A sua vantagem é que não dependemos de $\overrightarrow{N}$, inclusive podemos descobri-lo pelo produto vetorial entre $\overrightarrow{T}$ e $\overrightarrow{B}$. O último importante detalhe é que tal cálculo deve ser aplicado por face, ou seja, cada $\overrightarrow{T}$ e $\overrightarrow{B}$ aplica-se a uma respectiva face, para descobri-los em relação a cada vértice, é necessário interpolar de forma ponderada, em relação ao ângulo da face, todos os $\overrightarrow{T}$ e $\overrightarrow{B}$ das faces que o vértice em questão pertence (semelhante ao cálculo de normal por vértice).

2.3.3 Emboss Bump Mapping

Tendo armazenado o resultado do cálculo do espaço tangente para cada vértice, podemos, agora, efetuar o mapeamento de normais. Inicialmente, a URGE usava a versão primitiva do bump mapping, chamada de *Foward Differencing* [Bli78], mas popularmente apelidada de Emboss Bump Mapping (alguns chamam apenas de Emboss Mapping).

O bump mapping em tempo real (descrito na proxima sub-secção) é mais computacionalmente custoso, pois envolve operações pixel a pixel. A técnica do emboss bump mapping não é tão realista como do bump mapping, pois na verdade ela é um "truque" com o cálculo da iluminação desconsiderando a componente especular do material. Mesmo assim, possui uma considerável qualidade considerando seu custo inferior.

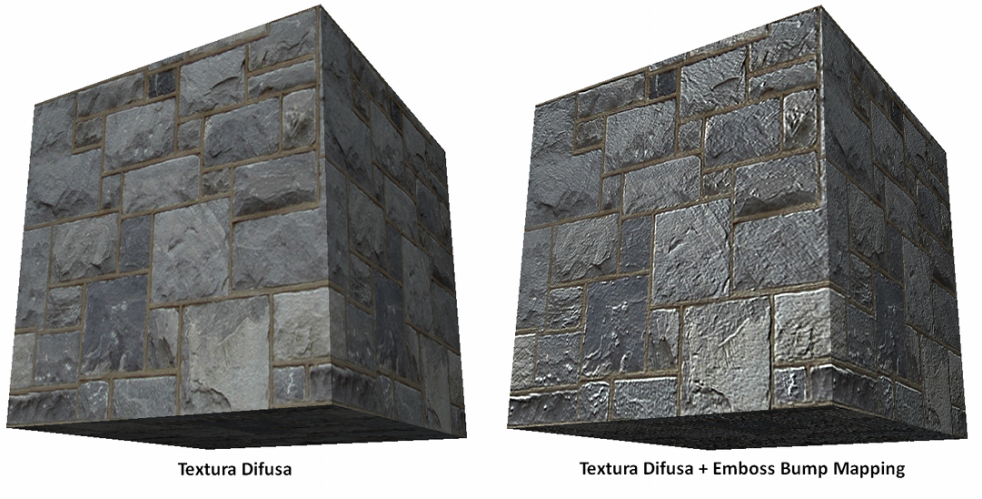

Figura 9: Exemplo de Emboss Bump Mapping aplicado em um cubo.

Ao contrário do bump mapping convencional, nesse mapeamento, ao invés de usarmos um mapa de normais, usamos um mapa de altura, que consiste numa imagem em escala cinza na qual a cor do pixel representa uma pequena distorção da altura do mesmo mapeado numa superfície arbitrária. Tendo a textura de um objeto e seu mapa de altura podemos efetuar o algoritmo descrito nos seguintes passos:

1. Com a luz desabilitada, desenhe a superfície usando o mapa de altura como sua textura difusa.
2. Desloque levemente a coordenada de textura para a posição da luz (a descrição em detalhes pode ser encontrada abaixo).
3. Desenhe novamente a superfície da mesma forma do primeiro passo, porém use agora as coordenadas de texturas calculadas no passo anterior. Faça isso subtraindo os valores desenhados no primeiro passo (para tal, basta inverter o mapa de altura nesse passo).
4. Habilite a luz e desenhe pela ultima vez a superfície, agora com sua textura difusa, misturando seus valores com os obtidos no passo anterior.

Para computar o segundo passo, considere $\overrightarrow{L}$ como sendo o vetor posição da luz, primeiramente $\overrightarrow{L}$ encontra-se no espaço dos objetos e deve ser transformado para o espaço tangente apenas multiplicando $\overrightarrow{L}$ pela matriz TBN, e assim teremos $\overrightarrow{L'}$.

$$\overrightarrow{L'} = TBN \times \overrightarrow{L} \tag{2.2}$$

Repare que a componente $L'_X, L'_Y e L'_Z$ nos fornece a direção da luz, respectivamente, nos eixos de $\overrightarrow{T}$, $\overrightarrow{B}$ e $\overrightarrow{N}$. Como as coordenadas de texturas estão no mesmo plano de $\overrightarrow{T}$, $\overrightarrow{B}$ podemos simplesmente efetuar o nosso segundo passo do algoritmo de acordo com a seguinte equação:

$$(s', t') = (s + L'_X \times k, t + L'_Y \times k) \tag{2.3}$$

Onde k é apenas um fator de escala.
Esse algoritmo apesar de precisar de mais de um passe de renderização, era bastante adotado a pouco tempo atrás. Todavia, a maioria dos computadores, atualmente, possuem uma potente capacidade de processamento gráfico fazendo com o que o emboss bump

mapping perdesse cenário para técnicas mais complexas. Por esse motivo a URGE abandonou tal técnica quando passou a usar programas shaders, o que simplificou a modelagem de seu núcleo de visualização.

2.3.4 Bump Mapping

Repare que na descrição do emboss bump mapping não perturbamos a normal como foi dito que deveria ser feito. Na verdade o algoritmo apenas aproximou a distorção da normal da superfície usando o efeito de deslocamento do seu mapa de altura. Por isso que dissemos que o emboss mapping é um truque, e não simula de forma tão realista quanto o *Normal Mapping* (*Bump Mapping*).

O Algoritmo do bump mapping baseia-se em um Mapa de Normal (*Normal Map*), isto é, uma imagem (textura) na qual cada pixel representa uma normal, no caso, o canal RGB corresponde ao eixo XYZ.

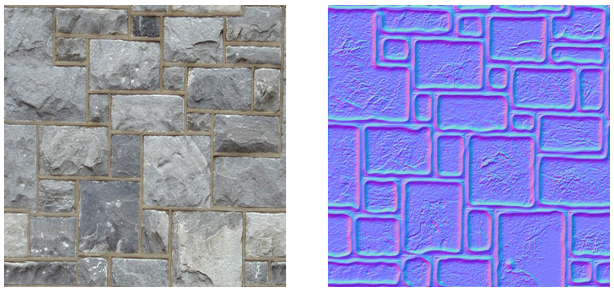

Figura 10: Exemplo de mapa de normal (a direita) associado a respectiva textura difusa(a esquerda)

Partindo-se do normal map, esse algoritmo distorce a normal da superfície em função de tal textura, e por fim usa uma equação de iluminação para computar a luz por pixel na superfície (a URGE usa o modelo Phong, descrito anteriormente). Na Figura 11 podemos comparar a imagem de um cubo com uma textura de ladrilhos de pedra e o mesmo cubo com a mesma textura usando a técnica do bump mapping (a textura e o mapa de altura usados podem ser vistos na Figura 10). Repare como cubo desenhado com esse algoritmo transmite uma idéia de superfície rugosa

Tendo em vista a idéia geral do algoritmo, discutirmos sua prática.
Inicialmente, a normal extraída do Normal Map ($\overrightarrow{N'}$) é um vetor unitário que varia de (-1,-1,-1) até (1,1,1), contudo as cores de qualquer textura, como é o caso do mapa de

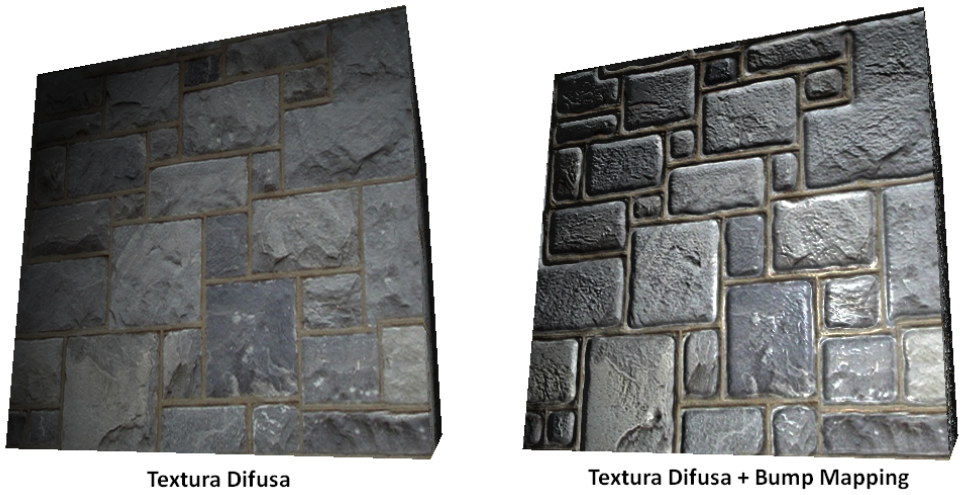

Figura 11: Exemplo criado com a URGE da técnica do bump mapping aplicado em um cubo texturizado.

normal, variam de (0,0,0) a (1,1,1). Portanto precisamos, antes de mais nada, converter tais limites de variação, simplesmente multiplicando a cor do pixel por dois e subtraindo um.

$$N'_x = 2.0 \times N'_r - 1.0$$
$$N'_y = 2.0 \times N'_g - 1.0$$
$$N'_z = 2.0 \times N'_b - 1.0$$

Repare que essa nova normal $\overrightarrow{N'} = (N'_x, N'_y, N'_z)$ substituirá $\overrightarrow{N}$ (normal da superficie) no cálculo da iluminação (descrito na seção 2.2). Entretanto, lembre-se que o normal mapping deve ser efetuado no espaço tangente, isso significa dizer que devemos multiplicar todos os vetores envolvidos no cálculo da iluminação pela matriz TBN, que no caso são:

$$\overrightarrow{L'} = TBN \times \overrightarrow{L}$$
$$\overrightarrow{H'} = TBN \times \overrightarrow{H}$$

E finalmente aplicamos nossos novos vetores à equação de iluminação, como por exemplo o modelo phong:

$$\begin{aligned} CorFinal &= MatAmb * LuzAmb \\ &+max\left(\overrightarrow{N'} \cdot \overrightarrow{L'}, 0\right) \times MatDif \times LuzDif \\ &+max\left(\overrightarrow{N'} \cdot \overrightarrow{H'}, 0\right)^{Brilho} \times MatEsp \times LuzEsp \end{aligned}$$

Esse algoritmo pode ser efetuado de forma simples e elegante usando GLSL como foi o caso da URGE. Também é possível realizar o Bump Mapping sem usar shaders, ou seja, através do pipeline fixo usando funções de extensão do OpenGL ligadas a parametrização

de texturas. Entretanto, essa forma de habilitar tal técnica é muito inflexível tornando difícil alterar certos parâmetros (como a equação de iluminação) ou partir para versões mais sofisticadas como o *Parallax Mapping*.

2.4 Parallax Mapping

Vimos que o bump mapping distorce a normal a nível de pixels adicionando, visualmente, mais detalhes na iluminação. Imagine o seguinte cenário: Uma parede de tijolos sob efeito de um foco de luz posicional, usando bump mapping, podemos apenas adicionar detalhes para a iluminação, afinal essa técnica perturba somente as normais. Porém, há uma técnica conhecida como *Parallax Mapping* na qual muitos consideram a evolução do Bump Mapping convencional, que distorce também as coordenadas de textura com o objetivo de gerar "relevos" a nível de pixels, isso fornece uma impressão tri-dimensional da própria texturização. Sendo assim, no caso da parede de tijolos, teríamos a impressão de que cada tijolo teria uma pequena parte "saltando" para fora da parede, como é o caso da vida real.

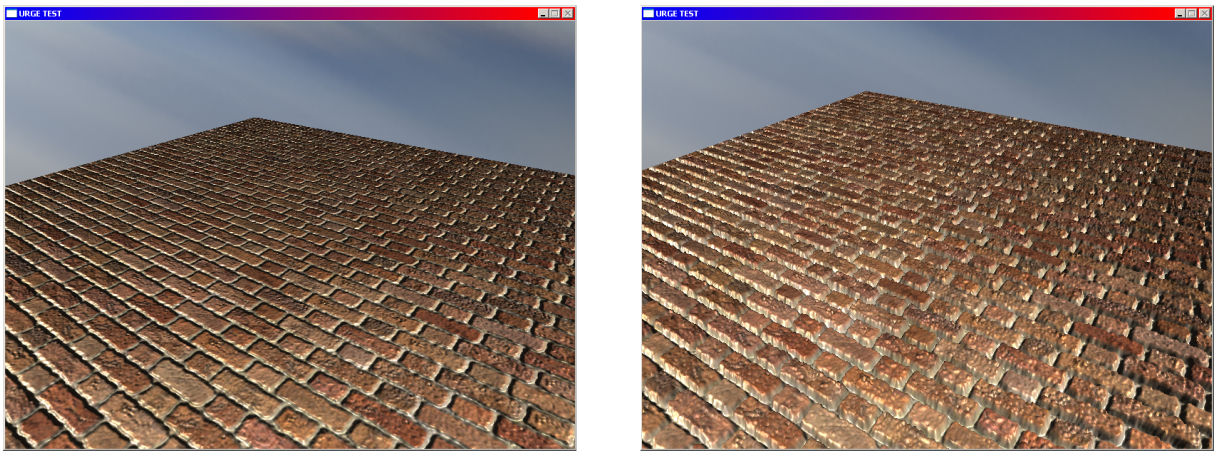

Figura 12: Plano desenhado com a URGE usando Parallax Occlusion Mapping (a direita) em comparação com o mesmo renderizado com Bump Mapping (a esquerda)

Repare na Figura 12 como plano renderizado usando a técnica do Parallax Occlusion Mapping (discutida nas seguintes sub-seções) fornece a impressão de tijolos "saltando" para fora do plano.

2.4.1 Metodologia

Lembre-se que essa técnica não altera a geometria da superficie, ela é feita a nível de pixels, isto significa dizer que ela apenas distorce aparência da mesma com o intuito de simular a noção de profundidade no material da superficie, por isso que ela também é conhecida como *Virtual Displacement Mapping* (Mapeamento por Deslocamento Virtual). Para simular a impressão de profundidade, o algoritmo precisa de uma nova função ou textura para poder efetuar o deslocamento da altura pixel a pixel, esse é chamado de displacement map (height map ou mapa de altura). Essa textura, semelhantemente ao emboss bump mapping, é uma imagem em escala cinza onde cada pixel representa o deslocamento da altura no respectivo ponto (mais "claro" mais alto e vice-versa).

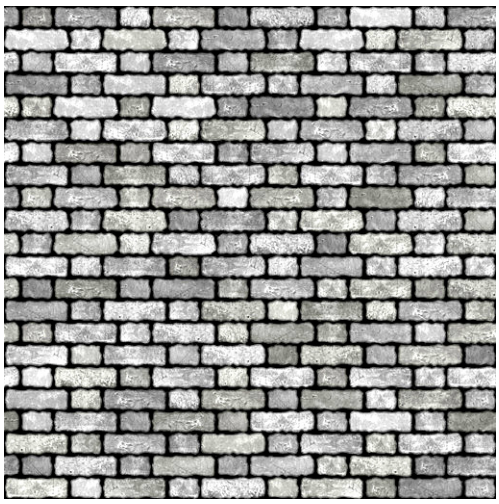

Figura 13: Exemplo de displacement map.

Conforme dito anteriormente, o cálculo do deslocamento da textura tem o objetivo de gerar a idéia de profundidade em função de um mapa de altura. Sendo assim, considere $\overrightarrow{T_D}$ como sendo o vetor de coordenadas de textura gerado pelo parallax mapping usando o vetor de coordenadas de textura original ($\overrightarrow{T}$) e a altura relativa ao respectivo pixel do mapa de altura (h), e por último, considere ($\overrightarrow{E}$) como sendo o vetor direção do observador já no espaço tangente (discutido anteriormente). $\overrightarrow{T_D}$ pode ser descrito conforme a seguinte equação:

$$\overrightarrow{T_D} = \overrightarrow{T} + (h \times k) \times \frac{E_{xy}}{E_z} \tag{2.4}$$

k é um coeficiente de escala para refinar h, valores de k recomendados para um melhor efeito são: $0.01 \leq h \leq 0.06$

Lembre-se que essa equação deve ser aplicada pixel a pixel pelo algoritmo, ito é, todas as variáveis da equação são referentes a um pixel da superfície em questão. Repare que se $\overrightarrow{E}$ encontra-se no espaço tangente, sabemos que $\overrightarrow{E_z}$ foi obtido pelo produto escalar do vetor direção do observador pela normal da superfície, essa coordenada é a responsável pelo efeito de profundidade. A Figura 14 dispõe uma representação visual de como funciona esse algoritmo.

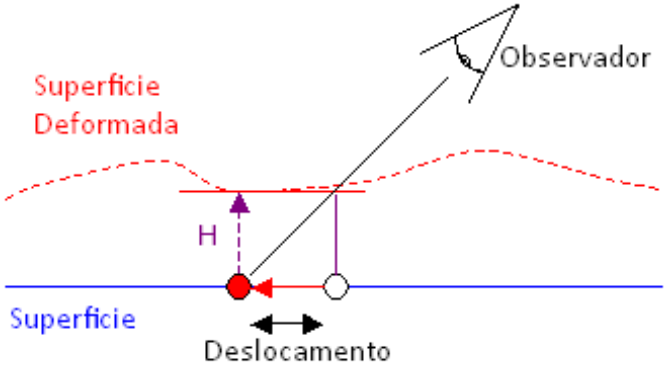

Figura 14: Representação gráfica do funcionamento do parallax mapping.

2.4.2 Parallax Occlusion Mapping

O algoritmo do parallax mapping nos fornece a idéia de profundidade distorcendo a textura e causando a impressão de que foi gerado relevos detalhados na superfície em questão, porém não considera a obstrução do relevo gerado. Felizmente existe a versão aprimorada desse algoritmo: Parallax Occlusion Mapping [MM05]. Essa técnica efetua testes de oclusão partindo do principio do *Ray Casting*, isto é, raios são traçados do ponto de vista do observador em direção à superficie, agora, a nível de pixels testes são efetuados através do displacement map para criar o efeito de obstrução do relevo.

Essa técnica requer um laço de repetição dentro de um programa shader, esse laço de repetição representará o traçar de um raio. A condição de saída do laço é caso a altura do relevo deslocado a partir do displacement map que o raio supostamente colidiu passou da altura encontrada na iteração anterior. No algoritmo, usamos como incremento do laço, a distância que o raio deve percorrer a cada iteração. É claro que há erros de precisão, mas podem ser minimizados diminuindo o incremento.

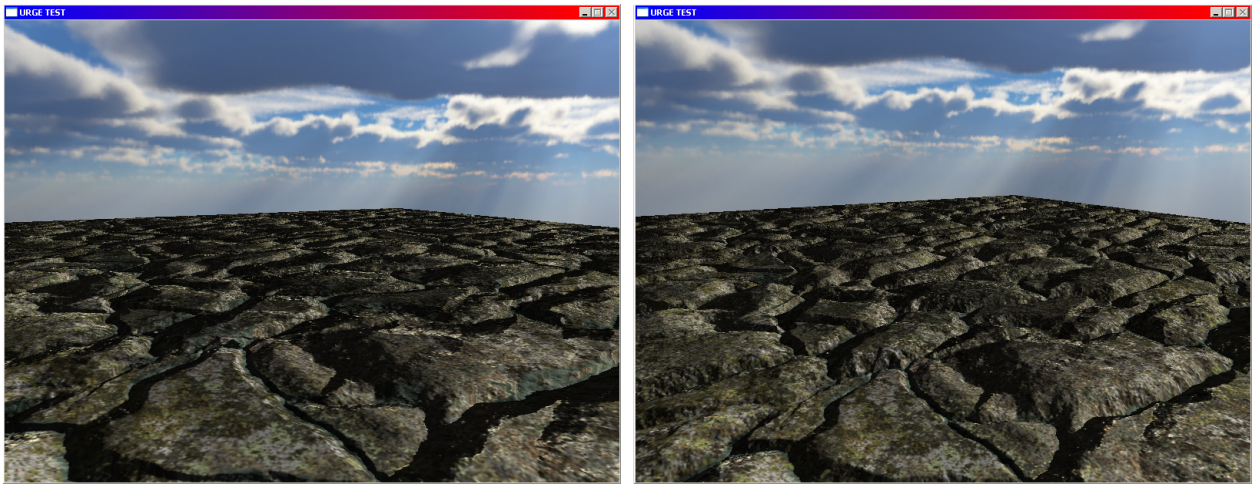

Figura 15: Comparação entre parallax occlusion mapping (a direita) e o parallax mapping convencional (a esquerda)

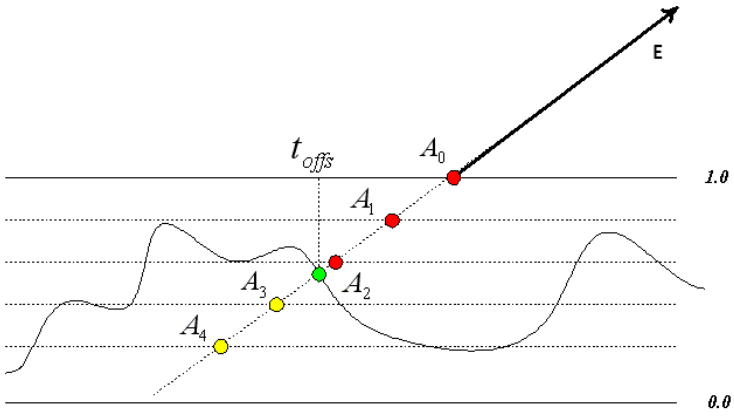

Figura 16: Representação das iterações na sequência de testes do ray casting

A Figura 16 apresenta um erro de precisão. Até o valor encontrado pelo Ray Casting, A_2, não houve colisão, contudo a colisão só foi detectada em A_3 o que nos leva a um erro, pois o verdadeiro ponto de colisão é representado por t_{offs}. A solução mais simples para esse problema é diminuir o incremento do raio, na Figura 16, corresponderia a diminuir a distância entre dois pontos A_n. Todavia, essa solução acarretaria um maior custo de processamento gráfico.

Apesar, do problema de precisão só ocorre em ângulos oblíquos entre o vetor direção do observador e a tangente à superfície em questão, é importante, para a qualidade gráfica, minimizar ao máximo esse efeito indesejado sem acarretar projuízos significantes ao processamento gráfico. Felizmente, há suma solução mais sofisticada do que diminuir o valor do incremento. Ela consiste em alterar o valor do incremento dinamicamente em função de uma busca binária no teste de colisão com o raio [POC05]

A URGE possui a implementação do parallax occlusion mapping usando o ray casting

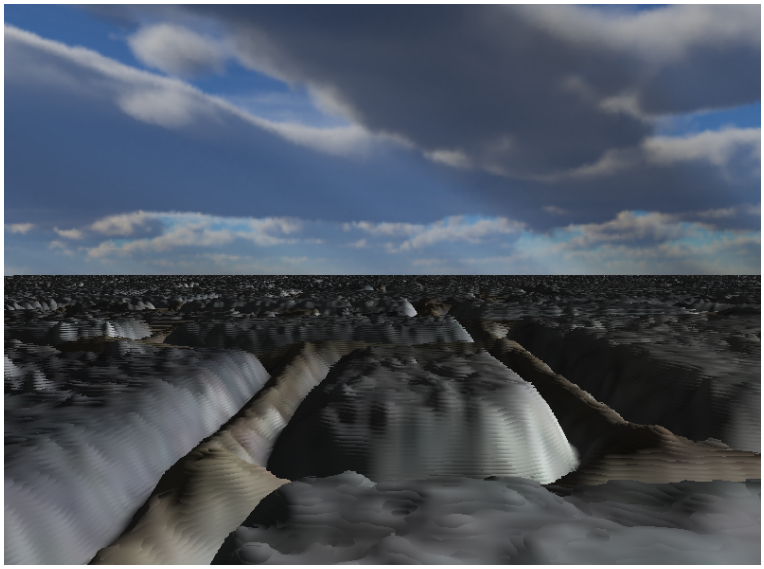

Figura 17: Erro de precisão do ray casting, presente em ângulos oblíquos.

via busca binária em função da opção de qualidade da engine, isto é, o único fator que pode alterar a precisão é o coeficiente de qualidade, que pode ser customizado pelo usuário. Portanto, um maior valor para a qualidade representa maior profundidade na busca binária. A engine ainda implementa o descarte de elementos fora do mapa de altura, isto é, caso o raio disparado passe da área de textura da superfície em questão todo processo de desenho do fragmento relativo será descartado [dTWL07]. Observe esse recurso junto ao parallax occlusion via busca binária na Figura 18.

2.5 Environment Mapping

Uma das técnicas mais clássicas de mapeamento de texturas na programação de jogos é o *Environment Mapping* (Mapeamento Reflexivo). Ela sempre foi bastante usada para aproximar, de forma convincente, a aparência de superfícies reflexivas, como espelhos, materiais metálicos ou plásticos.

Podemos também estender seu uso para simulações de superfícies refrativas como vidro, água ou cristal. Inclusive considerando fatores físicos do cotidiano como dispersão cromática e o efeito fresnel.

Nesta seção apresentaremos uma técnica chave e suas diferentes versões para a simulação, em alto desempenho, de superfícies reflexivas e refrativas em tempo real.

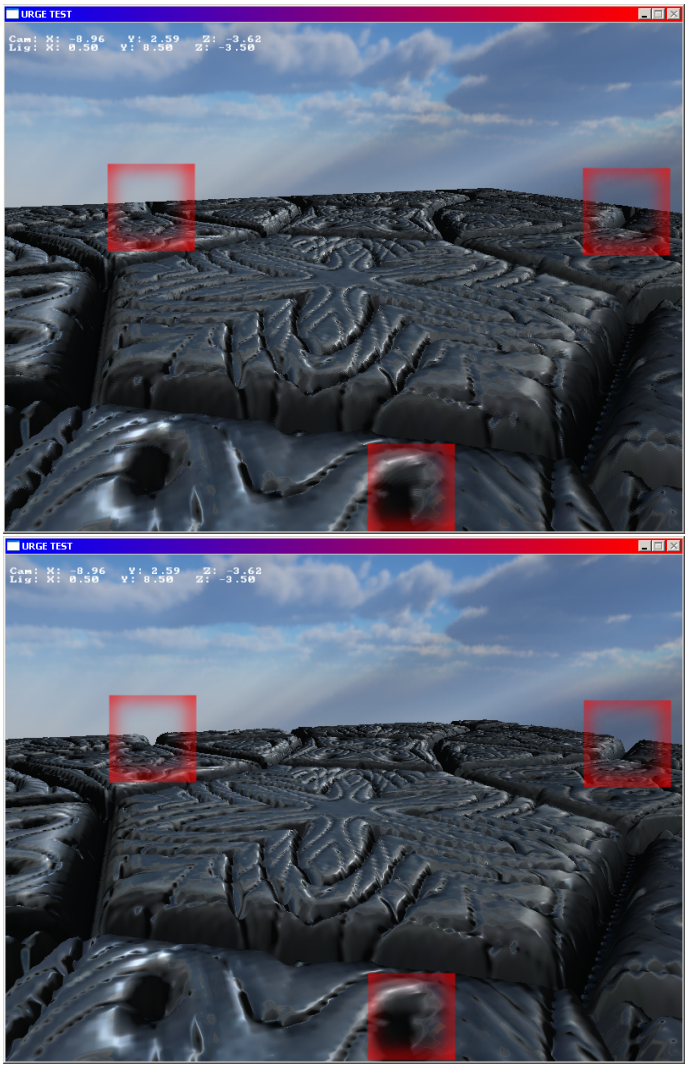

Figura 18: Comparação entre parallax occlusion comum e sua nova forma via busca binária e descarte de fragmentos (ultima imagem)

2.5.1 Iluminação Baseada em Imagens

Partindo-se do princípio do *Ray Tracing*, a simulação de superfícies reflexivas é um conceito bastante intuitivo, porém não é válido para aplicações em tempo real considerando a performance de processamento gráfico da maioria dos computadores pessoais de hoje em dia. Felizmente, já é possível simular tais superfícies com um grupo de técnicas de renderização chamadas de Iluminação Baseada em Imagens, ou IBL (Image-based Lighting) [CON99].

Assim como o ray tracing, o propósito dos algoritmos IBL aqui apresentados são de aproximar a aparência de superfícies reflexivas usando alguma técnica de iluminação, contudo, no caso das IBLs, o objetivo é atingido por meio de uma imagem, ou textura, pré-calculada. É claro que a textura pré-calculada deve ser a cena que a superfície em

Figura 19: Jogo criado com a URGE que usa o artífício de environment mapping.

questão reflete, mas renderizar uma supeficie com esse tipo de textura, por si, não gera nenhuma aparência metálica (veja a Figura 21).

O ponto chave do algoritmo está na forma de texturização da imagem em questão, ela deve ser efetuada de forma interativa de acordo com dois parâmetros chave: a normal da superfície específica e a direção a qual o observador está "olhando". Existem vários tipos de algoritmos de iluminação baseada em imagens cuja única diferença entre si, está na forma da texturização. Nas seguintes sub-seções, apresentaremos os dois algoritmos mais usados para atingir nosso objetivo: *Environment Spherical Mapping* [WNDS99a] e *Cubic Environment Mapping* [MPM02].

2.5.2 Sphere Mapping

Conforme dito anteriormente, o primeiro passo do environment mapping é armazenar a imagem da reflexão de uma supeficíe reflexiva em uma textura, ou seja, essa textura simula o ambiente que está em volta da superfície. O segundo e último passo é a forma de mapeamento. Uma solução é o Environment Spherical Mapping, popularmente conhecido como Sphere Mapping. Essa técnica transmite a idéia de que o ambiente em volta da superfície reflexiva foi projetado sob uma esfera que cobre o objeto em questão.

A Figura 22 descreve como é feito tal algoritmo. A direção para qual o observador

Figura 20: Exemplo feito na URGE de um Modelo de Copo usando efeito de refração com dispersão cromática e fresnel.

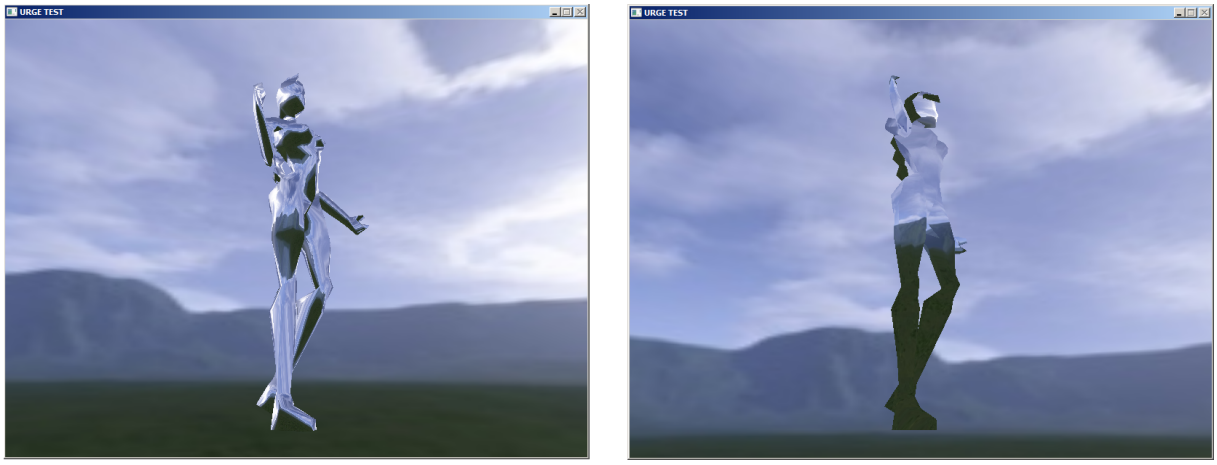

Figura 21: Comparação entre uma texturização reflexiva (a esquerda) e uma texturização difusa comum (a direita).

está "olhando" é representada por $\overrightarrow{e}$, e esse é refletido em torno da normal $\overrightarrow{N}$ da superfície, e por último mapeado na textura que contém a imagem do ambiente (*Sphere Map*), representado por $\overrightarrow{o}$. Para simular o efeito de refração, ao invés de refletir, refratamos o vetor $\overrightarrow{e}$ em relação a $\overrightarrow{N}$, e por último mapear novamente no sphere map. Lembre que $\overrightarrow{o}$ é um vetor bi-dimensional que representa a coordenada de textura do vetor refletido $\overrightarrow{R}$, sendo assim, considere $\overrightarrow{o} = (s, t)$. Podemos descobri-lo matematicamente de acordo uma simples equação.

Considere m como sendo o coeficiente de cálculo da projeção de $\overrightarrow{R}$ na esfera, isto é:

$$m = 2 \times \sqrt{R_x^2 + R_y^2 + (R_z + 1)^2} \tag{2.5}$$

Sendo assim, $\overrightarrow{o} = (s, t)$ pode ser calculado da seguinte forma:

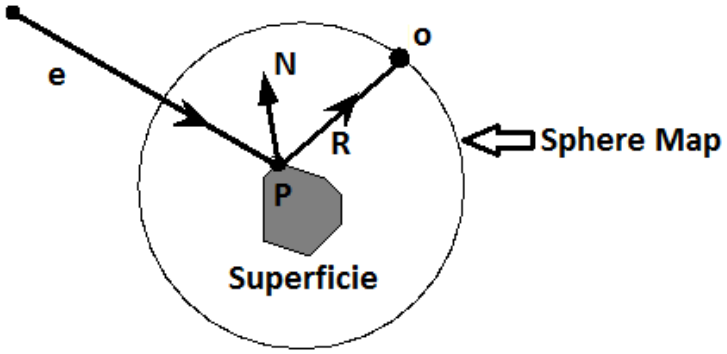

Figura 22: Representação visual do sphere mapping

$$(s,t) = (\frac{R_x}{m}, \frac{R_y}{m}) \tag{2.6}$$

Todavia, essa parametrização não considerou que a superfície deve se encontrar no centro da esfera, caso contrario poderá resultar num efeito incorreto (Figura 23)

Portanto deve-los centraliza-la e, finalmente, teremos a ultima versão da equação:

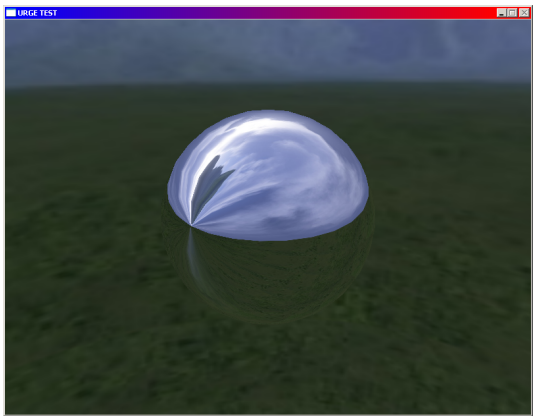

Figura 23: Efeito indesejado da parametrização incorreta.

$$(s,t) = (\frac{R_x}{m} + \frac{1}{2}, \frac{R_y}{m} + \frac{1}{2}) \tag{2.7}$$

Esse algoritmo já é implementado pelo próprio OpenGL (apenas para a aparência de reflexão), contudo o efeito produzido é o de uma reflexão onidirecional, uma vez que ele não tem acesso ao vetor direção do observador (esse é aproximado pela própria posição da superfície). Como a URGE manipula o observador através de uma entidade, ela pode simular uma reflexão referente à sua direção, potencializando a qualidade gráfica.

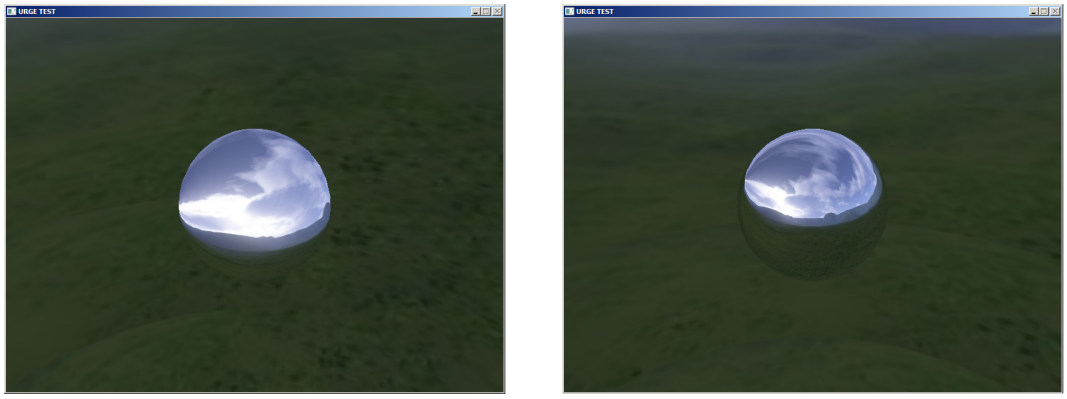

Figura 24: Efeito indesejado da reflexão onidirecional produzido pelo OpenGL (esquerda), correção do mesmo pela URGE (direita)

2.5.3 Cube Mapping

A segunda forma de mapeamento ambiente é o Cubic Environment Mapping, ou simplesmente Cube Mapping. Ela é bastante similar ao sphere mapping, sua única diferença é que, nesse caso, o ambiente deve ser armazenado em seis texturas que compõem as faces de um cubo chamado Cube Map. O cube map, fornece uma idéia de uma superfície que envolve o objeto reflexivo, na prática, todos os vetores refletidos na superfície serão mapeados nesse cubo.

Cada face do cubo é representada por um plano infinito cuja normal passa pela coor-

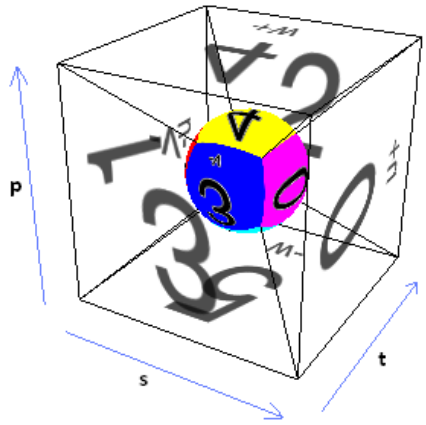

Figura 25: Representação Visual do comportamento do cube mapping

denada de textura (0.5,0.5), assim as coordenada de textura do vetor reflexão projetado no cubo é, claramente, tridimensional $\overrightarrow{\sigma} = (s, t, p)$. O vetor overrightarrowo pode ser simplesmente calculado pela multiplicação de uma constante com o produto escalar do vetor refletido $\overrightarrow{R}$ e a normal de cada face do Cubo.

Graças ao fato do cube map armazenar as seis texturas de forma isométrica (textura quadrada) o efeito de reflexão é mais realista que o sphere mapping

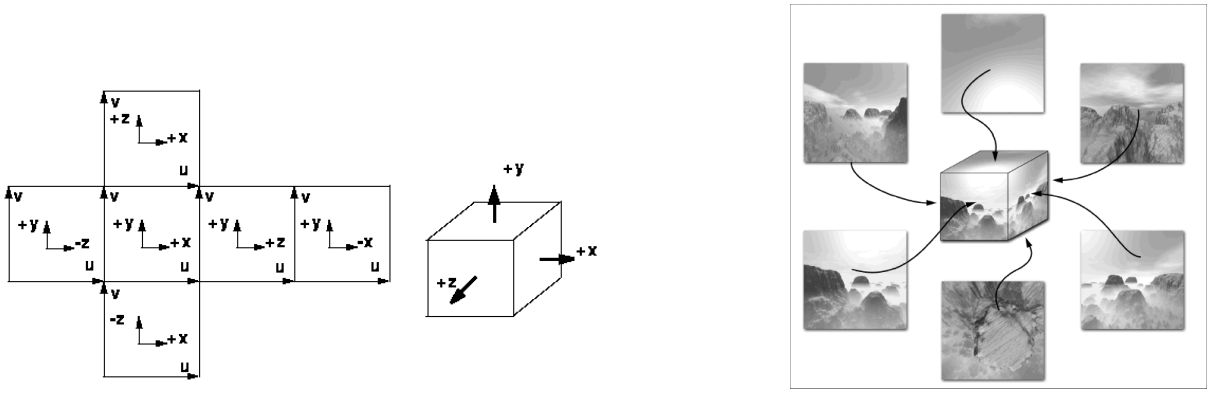

Figura 26: Representação visual dos eixos e normais de um Cube Map (a esquerda) e representação das seis texturas do mesmo (a direita)

Figura 27: Exemplo criado na URGE, um modelo de um carro com reflexão via cube mapping

2.5.4 Environment Bump Mapping

Também é possível adicionar definição pixel a pixel a um efeito de reflexão ou refração, usando a técnica do *Environment Bump Mapping*. Esse algoritmo é parecido com o bump mapping convencional, partindo-se um mapa de normais podemos distorcer em nível de pixels a normais do próprio modelo e em seguida aplicar o environment mapping. Assim podemos adicionar mais detalhes no efeito visual desejado. Isso é extremamente útil para os mesmos casos que usamos o bump mapping.

Figura 28: Modelo 3D renderizado na URGE com Enviroment Bump Mapping (direita), em contraste com o mesmo usando o Environment Mapping comum (esquerda)

Para distorcemos a normal, precisamos basicamente efetuar a soma normalizada do deslocamento do vetor normal extraído a partir do normal map com a normal da superfície, também é interessante multiplicar por uma constante com o intuito de diminuir a rugosidade. Considere $\overrightarrow{N_R}$ como sendo a normal distorcida a partir de $\overrightarrow{N}$ (normal da superfície) e $\overrightarrow{N'}$ (normal do mapa de normais), e por último, k $(0.0 < k \leq 1.0)$ sendo o coeficiente de rugosidade, logo podemos definir $\overrightarrow{N_R}$ como:

$$\overrightarrow{N_R} = \frac{(\overrightarrow{N} + k \times ((0.0, 0.0, 1.0) - \overrightarrow{N'}))}{|\overrightarrow{N_R}|} \tag{2.8}$$

A divisão pelo módulo de $\overrightarrow{N_R}$ significa, simplismente, efetuar uma normalização após o calculo descrito no numerador da equação.

Esse algoritmo é suportável apenas por computadores com suporte ao pipeline programável do OpenGL. Além disso, considere que essa operação é pixel a pixel, ou seja, pode ser muito custoso para computadores de desempenho limitado.

2.6 Renderização de Terrenos

Hoje em dia é possível encontrar facilmente uma extensa gama de métodos de simulação visual de terrenos. Afinal, há anos, essa área é o foco de diversos pesquisadores no mundo inteiro, cada um deles apresenta diferentes e inovadoras técnicas de representação de terrenos em computadores. Muitas dessas técnicas foram feitas para serem processadas pela CPU. Contudo, ultimamente com o avanço da tecnologia dos computadores pessoais, técnicas de visualização de terrenos orientadas a GPU (processadores gráficos)

tem ganhado cada vez mais influência no campo da computação gráfica aplicada a jogos eletrônicos.

A URGE possui suporte a renderização de terrenos baseado em técnicas relativamente simples programadas em CPU [Ast06a]. O objetivo disso é adaptar o algoritmo para funcionar em computadores com dispositivos gráficos menos potentes. Apesar, de ser orientada pela CPU, a visualização de terrenos da engine implementa técnicas de otimização de desenho como LODs e o uso de VBO.
Nesta seção iremos discutir tais técnicas envolvidas na visualização de terrenos implementadas pela URGE.

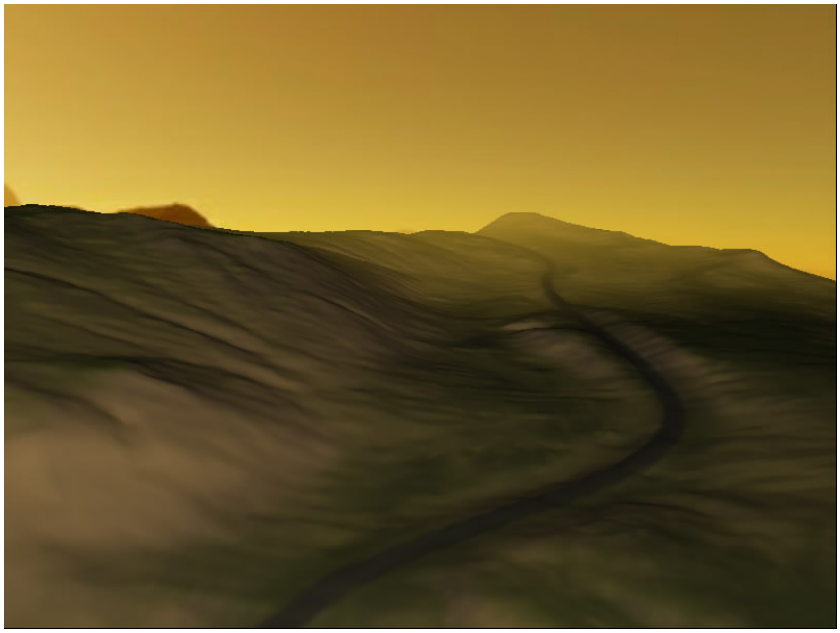

Figura 29: Exemplo de Terreno desenhado a partir da URGE.

2.6.1 Vertex Buffer Objects

Vários elementos visuais da URGE, como os próprios terrenos, se utilizam de um eficiente recurso chamado *Vertex Buffer Object* (VBO) disponibilizado pelo OpenGL [Ope]. VBOs permitem que os todas as informações de um elemento visual (como vértices, normais ou coordenadas de texturas) sejam armazenados em memória de alto desempenho gráfico ao lado do servidor (unidade responsável pelo processamento gráfico) e promove a transferência eficiente de dados. Quando usamos VBO, podemos copiar todos os atributos de um elemento visual específico para um *buffer* especial armazenados em memória de alto desempenho gráfico administrada pelo sistema, isso significa que dependendo do

gerenciamento de memória, os dados poderão ser armazenados na memória de vídeo, na memória da AGP (compartilhada entre CPU e GPU) ou na própria memória da CPU. Na prática, esse recurso potencializa consideravelmente o desempenho da renderização, elevando a eficiência de jogos produzidos pela engine.

VBO encontra-se apenas disponível para versões do OpenGL iguais ou mais recentes que a 1.5, apesar de alguns dispositivos já possuírem suporte através de rotinas de extensão na versão 1.4. Caso o computador não possua disponibilidade de tal recurso, a URGE adapta seus elementos visuais para a manipulação através de *Vertex Arrays*, ou seja, uma alternativa mais simples de armazenamento de dados de objetos visuais, porém de forma menos eficiente.

2.6.2 Carregamento de Terrenos

Na URGE, e também na maioria das engines, entende-se por terreno, como sendo uma superfície tri-dimensional a qual para dois pontos arbitrários $\overrightarrow{P} = (P_x, P_y, P_z)$ e $\overrightarrow{Q} = (Q_x, Q_y, Q_z)$ sobre a superfície, temos obrigatoriamente que: $P_x \neq Q_x, P_z \neq Q_z$. Isso significa afirmar que, no terreno, não pode haver elevações ou depressões curvas como são os casos de túneis ou arcos naturais. Tendo tal premissa em mente, podemos facilmente gerar essa superfície a partir de uma função ou textura que chamamos de mapa de altura (*height map*). No caso de textura, o mapa de altura constitui-se de uma imagem em escala cinza na qual a posição (x,y) do pixel representa a posição (x,z) do respectivo vértice do terreno, e a cor do pixel representa a altura y do vértice em questão (equivalente ao displacement map do parallax mapping, discutido anteriormente).

Uma vez carregado, o terreno pode ser definido a partir do height map de forma bastante simples. Imagine, nosso terreno como sendo, inicialmente uma sequência de triângulos alinhados de forma a descrever um plano (veja a Figura 31).

A posição de um vértice $\overrightarrow{V}$ é dado pelas seguintes informações de um respectivo pixel $\overrightarrow{P} = (P_x, P_y)$ do mapa de altura:

$$\overrightarrow{V} = (P_x, cor(\overrightarrow{P}), P_y) \tag{2.9}$$

As respectivas proporções de largura, comprimento e profundidade ficam a critério do usuário.

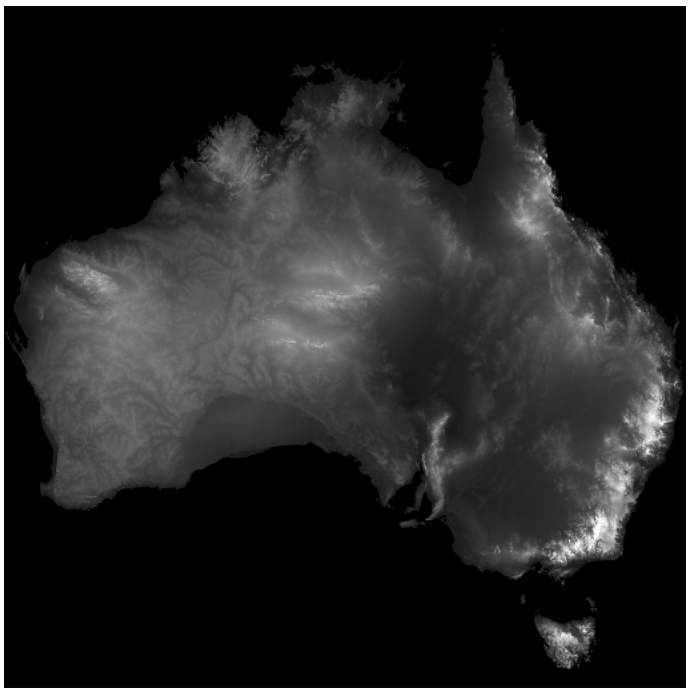

Figura 30: Exemplo de mapa de altura representando a geografia do território da Austrália

O cálculo das normais é semelhante ao cálculo de normais aplicado a qualquer superfície tri-dimensional (descrito na Seção 2.3.1).
A texturização é feita, basicamente, em função de duas constantes: a respectiva dimensão do mapa de altura e o espaçamento entre os vértices do terreno. Considere w e h, como sendo, respectivamente, a largura e altura do height map em questão, e considere k como sendo o espaçamento entre qualquer dois vértices adjacentes do terreno (supondo que o espaçamento entre os vértices é o mesmo para todos). O vetor $\overrightarrow{T} = (s, t)$ que representa a coordenada de textura de relacionada a um respectivo vértice $\overrightarrow{V} = (V_x, V_y, V_z)$ é dado por:

$$\overrightarrow{T} = (s, t) = (\frac{V_x \times k}{w}, \frac{V_z \times k}{h}) \tag{2.10}$$

No caso da URGE, essas três informações mais o cálculo dos vetores tangentes e binormais são armazenados em um VBO para posteriormente serem renderizados pelo OpenGL.

2.6.3 Level Of Detail

Desenhar o terreno inteiro da forma como foi carregado pode ser algo muito custoso, principalmente se ele for grande em relação ao observador ou bastante detalhado (a distância entre cada vértice foi pequena). Por outro lado, um terreno sem detalhes pode ser visualmente desagradável. A solução é otimizar tudo que o observador não enxerga ou enxerga de longe. Há uma série de técnicas que objetivam o desenho de terrenos em

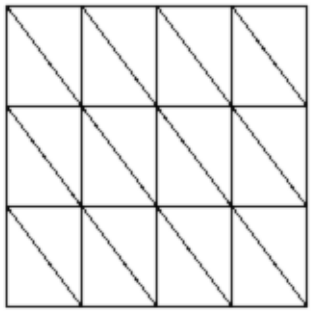

Figura 31: Representação da estrutura interna dos dados de visualização do terreno (planos formados por triangulos)

alta performance e qualidade, todas essas técnicas são baseadas em níveis de detalhes (*Level Of Detail*) ou, simplesmente, LOD. A URGE implementa uma versão adaptada de algoritmos VDR (*View-dependent Refinement*) [Hop97], que diminuem o nível de detalhe (relacionado ao número de vértices) em função da distância do observador. O Primeiro passo do algoritmo é carregar e armazenar cópias de vários LODs do terreno, o padrão da URGE é armazenar 4x4 células de dois triângulos, 8x8, 16x16 até o máximo que é relativo a proporção entre o tamanho do mapa de altura, tamanho do terreno e o nível de qualidade de visualização, não podendo exceder 128x128 (veja a Figura 32).

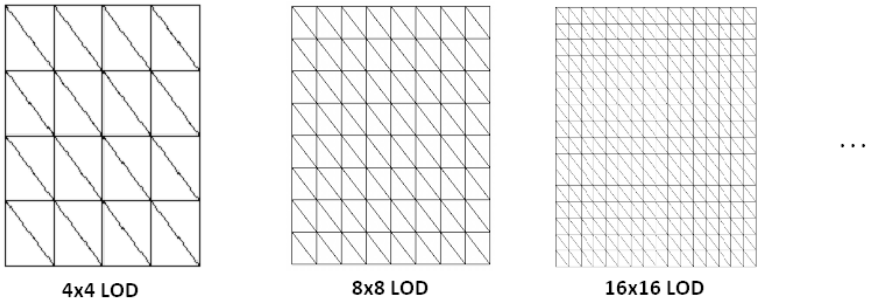

Figura 32: Representação visual dos diversos LODs gerados no carregamento de um terreno.

Então o terreno é seccionado em NxN partições espaciais (nesse caso, N não representa o número de triangulos, mas o espaçamento da dimensão espacial que contém os triangulos). Cada partição é desenhada de forma independente caso passe em um teste de otimização de processamento baseado *Quadtrees* (discutida em detalhes no Capítulo 4).

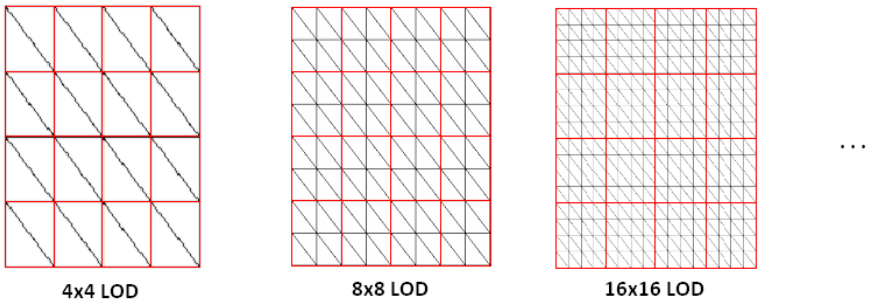

Figura 33: Partição espacial dos respectivos LODs de um terreno

O valor da dimensão N contém exatamente um par de triangulos do menor nível de detalhe do terreno (primeira imagem da Figura 33), e é calculado de forma proporcional à profundidade máxima do algoritmo da Quadtree e o valor da área do terreno.
No caso de uma seção passar no teste de Quadtrees, seu nível de detalhe será selecionado em função da distância do observador, isto é, quando maior a distância, um LOD mais simples é selecionado, e para distâncias pequenas o LOD mais preciso é usado. Existem várias outras soluções para o realizar esse tipo de otimização, inclusive discutiremos algumas delas no Capítulo 4. Porém, essa solução foi escolhida por conta de sua considerável eficiência e relativa simplicidade. É interessante enfatizar que a URGE não foi criada para a renderização de terrenos em alta precisão, como é o caso de estudos na área de visualização científica. Seu objetivo é simular, considerando o desempenho do cotmputador, terrenos atraentes para jogos eletrônicos.

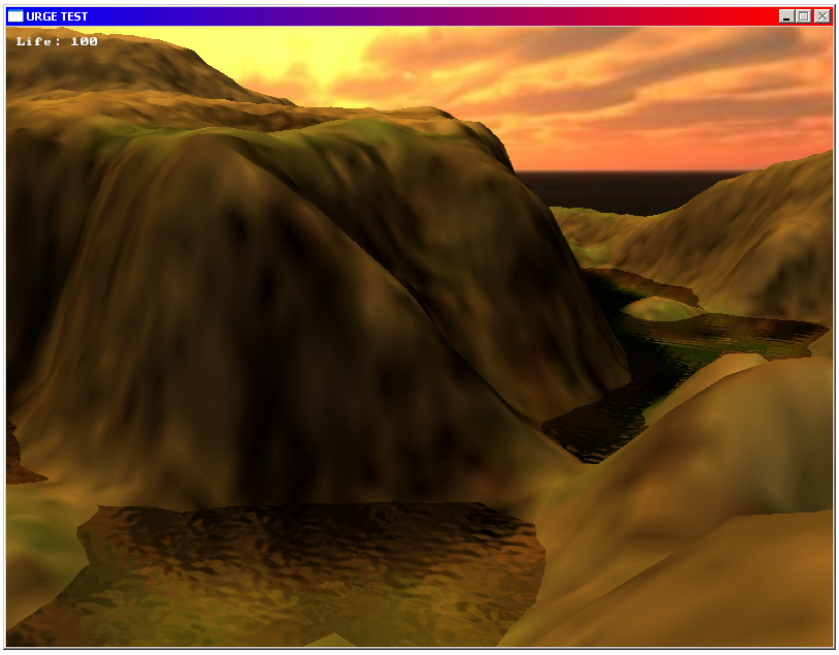

Figura 34: Exemplo de Renderização de terrenos pela URGE.

2.7 Sistemas de Partículas

Sistema de partículas, hoje em dia, é usado nas mais diversas áreas da computação gráfica. Na computação cientifica podemos usá-lo para descrever em detalhes comportamento físico de fluídos, ou focarmos na qualidade visual, como é o caso de animações 3D (filmes), podendo simular líquidos de forma fiel à realidade. No caso de jogos eletrônicos e aplicações interativas, é de suma importância o anteparo com a capacidade de processamento de um computador pessoal, pois devemos nos lembrar que o objetivo é efetuar essa técnica junto a várias outras, e em tempo real, além de possibilitar seu funcionamento em computadores de baixo desempenho.
Sendo assim, nesta seção, explicaremos o funcionamento dos sistemas de partículas implementados na URGE para a simulação de efeitos visuais como fogo, rastros luminosos ou até fumaça.

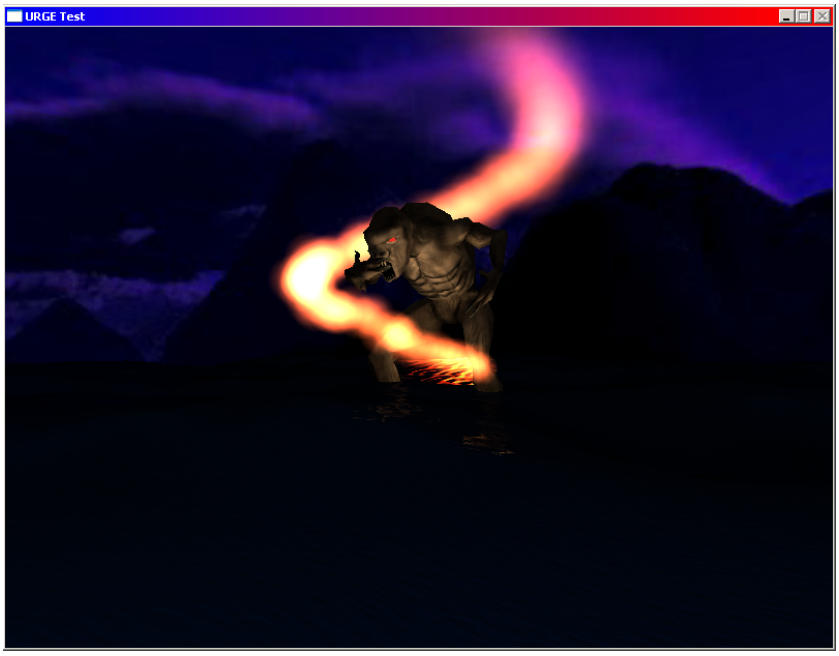

Figura 35: Exemplo de sistema de partículas gerado pela URGE.

2.7.1 Definição

Conforme dito anteriormente, sistemas de partículas servem para simular efeitos como fogo, fumaça, rastros luminosos, água, fogos de artifício, neve, chuva, e outros... Por conta dessa vasta gama de aplicações, não é fácil formular uma definição precisa para esse recurso, devido a abrangência de sua funcionalidade precisamos defini-lo, também,

de forma abrangente [Ree83].

A primeira premissa é que um sistema de partículas deve ser composto de uma ou mais partículas (entidades) individuais. Cada uma destas partículas é dotada de atributos comuns entre as mesmas que, diretamente ou indiretamente, afetam seu comportamento ou, em última análise, como, onde e quando a partícula deve ser renderizada.
A segunda premissa é que, apesar de todas as partículas possuírem os mesmos atributos, é importante entender que seus valores referentes podem e devem ser diferentes entre si, portanto, cada partícula age de forma diferente mas conservando as mesmas leis as quais são submetidas pelo seu sistema em questão.
A ultima premissa para a definição de um sistema de partículas é discutível, porém na grande maioria dos casos é válida e importante: Cada atributo de cada partícula é estocasticamente definido, ou seja, cada unidade do sistema de partículas deve possuir algum tipo de elemento aleatório. Este elemento aleatório pode ser usado para simular de forma simplista o comportamento quântico da natureza, como é o caso de fogo, chuva ou neve. É perfeitamente plausível, e de fato é o que acontece na maioria dos casos, o elemento aleatório ser controlado por algum tipo de lei estocástica pré-definida como limites, variância, ou alguma função de distribuição [Ree85].

2.7.2 Modelagem e Funcionamento

Vimos que sistemas de partículas são entidades que estabelecem e gerenciam leis pelas quais cada partícula deve ser submetida. A forma mais comum de armazenamento é a de listas encadeadas ou até simples vetores, não há a necessidade de nenhuma outra forma mais elaborada, pois em quase todos os casos não distinguimos nenhuma partícula do sistema, portanto qualquer uma está igualmente habilitada a ser selecionada, tornando dispensável o uso de consultas elaboradas. Além disso, não é comum adicionamos ou excluímos uma partícula após a construção de um sistema. Portanto, o acesso a cada partícula pode ser integralmente vetorial, o que torna a forma de armazenamento e acesso bastante simples e com baixo custo de processamento.

Do ponto de vista visual, partículas são representadas por primitivas gráficas como pontos, linhas ou simples polígonos. Em alguns casos, sistemas de partículas, também têm sido utilizados para representar a dinâmica de grupos complexos, como é o caso da bio-róbotica na qual simula comportamentos de animais em grupos como pássaros migratórios.

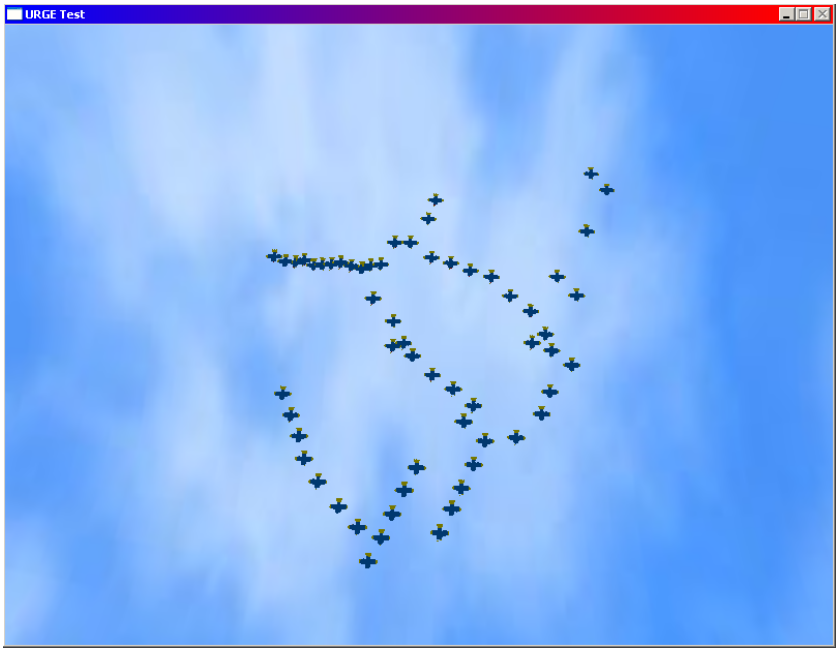

Figura 36: Sistema de Partículas criado na URGE com o intuito de simular o vôo de pássaros migratórios em bando.

2.7.3 Propriedades das Partículas

Vimos anteriormente que a definição de um sistema de partículas não é simples nem específica, da mesma forma são os atributos de cada partícula. Dependendo da funcionalidade do sistema, podemos incluir os mais variados atributos. Mesmo assim, apresentaremos um conjunto de atributos que abrangem a grande maioria das aplicações [Rey87]. Eis as propriedades, adotadas na maioria dos sistemas, que cada partícula dispõe:

- Posição
- Velocidade (vetor)
- Tempo de vida
- Tamanho
- Cor
- Massa Específica
- Transparência
- Idade

Atributos como posição e velocidade, são intuitivos, vale apenas lembrar que a velocidade possui módulo e direção, isto é, ela deve ser vetorial, afinal cada partícula pode ser mover em diferentes velocidades e para diferentes direções.

O Tempo de vida significa o quanto de tempo falta para a partícula desaparecer, isto é, parar de ser renderizada, esse parâmetro pode variar com o tempo, por exemplo, ou com algum fator estocástico. O tamanho remete à dimensão da imagem da partícula em questão, e em geral é opcional, isto é, pode ser proporcional a outros valores, como tempo de vida, por exemplo, conforme menor o tempo de vida, maior é a partícula.
Cor nem sempre é usado, mas quando usamos, não variamos muito seu valor para cada partícula. Por exemplo, o efeito de fumaça da URGE, escurece um pouco cada partícula conforme dois fatores, o primeiro é o tempo e o segundo é um fator de aleatoriedade.
Massa Específica, ou densidade, é um atributo usado em sistemas de forte caráter físico, não é o caso da URGE, mas pode ser usado em simulação de fluídos, onde tal atributo torna-se útil para diferenciar o comportamento de dois ou mais líquidos diferentes.
Repare que qualquer uma dessas propriedades pode adquirir um valor totalmente proporcional a outro atributo pré-existente, tornando-se obsoleto. O caso do efeito de rastro luminoso implementado pela URGE é um exemplo disso, a transparência e a idade de cada partícula é diretamente proporcional ao tempo de vida da mesma.
Os atributos aqui citados, apesar de serem básicos, estão presentes em quase todos os tipos de sistemas e, na maioria dos casos, não há a necessidade de adicionar mais atributos, apenas em situações especificas adicionamos outras propriedades.

2.7.4 Ciclo de Vida de uma Partícula

O ciclo de vida de cada párticula pode ser classificado se forma simples em três etapas:

- Emissão
- Dinâmica
- Extinção

A emissão, também conhecida como geração, em geral é em função do tempo, raramente usamos algum fator estocástico para criar a partícula. Na grande maioria das situações (como é o caso de todos os efeitos produzidos pela URGE) emitimos uma ou mais partículas a cada quadro de animação do jogo.

A principal etapa, que diferencia cada sistema, é a dinâmica de partículas. Nessa etapa variamos os atributos, discutidos anteriormente, em função do tempo, de fatores aleatórios ou até mesmo de funções, ou comportamentos, pré-estabelecidos. Por exemplo, o efeito de fogo implementado pela URGE (imagem 35), varia a transparência e o tamanho

em função do tempo, em quanto sua velocidade, e portanto, sua posição, possuem uma certa variância definida por um fator de aleatoriedade.

A ultima etapa, a extinção, consiste na condição pela qual o sistema se baseia para parar de desenhar uma certa partícula, quando isso acontece, dizemos que a ela foi apagada. A extinção também é uma etapa de características comuns em relação à maioria dos sistemas, ou seja, na maioria dos casos apaga-se uma partícula conforme seu tempo de vida ou sua idade, e esses por si, são em função do tempo ou, raramente, em função de um processo estocástico.

2.8 Conclusão sobre Visualização

Durante esse capítulo, discutimos as técnicas usadas pelo núcleo de visualização da URGE (sistema de rendering). Seu propósito é fundado em três objetivos:

- Simular a visualização de uma extensa gama de materiais presentes em jogos;
- Produzir imagens de alta qualidade e fieis a realidade;
- Funcionar em tempo real considerando o limite da capacidade de processamento gráfico do respectivo computador.

Para atingir tal propósito, a engine deve considerar o uso de artifícios de alta tecnologia disponíveis há não muito tempo atrás, e também estar sempre pronta para uma exceção. Em outras palavras, é vital o uso de programas shaders para atingir um alto nível de qualidade gráfica, contudo a URGE está preparada para o caso de um computador não dispor desse recurso, a qualidade pode ser inferior, mesmo assim a engine ainda garante seu objetivo.

3 Simulação Física em Tempo Real

O segundo elo da URGE é o núcleo de simulação física em tempo real. Seu propósito é bastante simples: detectar colisões entre superfícies geométricas e aplicar a interação entre corpos físicos respeitando as leis da dinâmica física.
É interessante ressaltar que esse sistema não objetiva a precisão da colisão física, mas a eficiência e coesão. Afinal, durante um cenário de um jogo, toda interação entre corpos não precisa ser calculada com exatidão numérica, mas deve ser coerente com a realidade e, principalmente, funcionar em tempo real, ou seja, os resultados são computados e simultaneamente exibidos.
Este Capítulo apresenta as técnicas de detecção e resposta de colisão de diferentes formas geométricas, além da dinâmica física em função de conceitos como aceleração, velocidade e massa.

3.1 Sistema de Detecção de Colisões

A primeira etapa do núcleo de Física é realizar testes de colisão em função de formas geométricas [Eri05]. Colisões ocorrem quando objetos se chocam dentro de uma cena do jogo, apesar de nem todos os objetos precisarem interagir, os que interagem precisam saber em que momento ela acontece, para então simular as ações e reações.
Na URGE, as superfícies tri-dimensionais têm sua geometria aproximada para estruturas mais simples, com o intuito de garantir a eficiência do sistema. Iremos discutir a detecção em função de três diferentes estruturas geométricas da engine: Esferas, AABBs e OBBs.

3.1.1 Detecção de Colisão por Esferas

A Detecção de colisão por esferas é o tipo mais simples de se efetuar. Considere dois objetos esféricos de raios R_1 e R_2 e posições dos centros $\overrightarrow{C_1}$ e $\overrightarrow{C_2}$, respectivamente. Será detectada a colisão entre eles, caso:

$|\overrightarrow{C_1}$ - $\overrightarrow{C_2}| \leq |R_1 + R_2|$

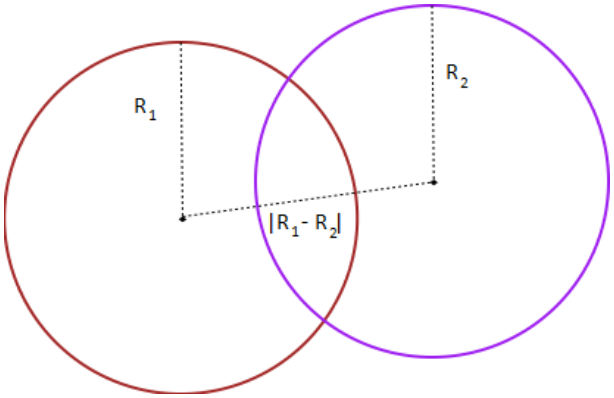

Figura 37: Representação visual da colisão entre duas esferas

3.1.2 Detecção de Colisão por AABB

O termo AABB significa *Axis Aligned Bounding Box*, ou caixa alinhada ao eixo de origem. Essa figura geométrica representa um paralelepípedo de arestas sempre paralelas ou perpendiculares ao eixo cartesiano tri-dimensional.

Considerando $\overrightarrow{P_1}$ e $\overrightarrow{P_2}$ os pontos superiores esquerdos dois AABBs, e L_1, L_2, H_1, H_2, D_1 e D_2 os respectivos valores de largura, altura e comprimento em função dos dois objetos. A forma mais intuitiva de se testar colisões entre duas AABBs é detectar todos os casos em que tais variáveis traduzem um cenário de colisão. Por exemplo, se as seis inequações abaixo forem verdadeiras, teríamos detectado uma colisão.

$P_1x \leq P_2x + L_2$

$P_1y \leq P_2y + H_2$

$P_1z \leq P_2z + D_2$

$P_1x + L_1 \geq P_2x$

$P_1y + H_1 \geq P_2y$

$P_1z + D_1 \geq P_2z$

Entretanto, dessa maneira, teríamos mais de dezesseis casos diferentes, sendo que, para cada caso, seriam necessárias um conjunto de inequações, como o caso mostrado acima. É mais inteligente testarmos as situações em que não há colisão, caso todos eles falhem, a colisão foi detectada. Dessa forma, diminuímos uma série desnecessária de testes, a simplesmente seis casos explícitos abaixo:

$P_1x > P_2x + L_2$
$P_1y > P_2y + H_2$
$P_1z > P_2z + D_2$
$P_1x + L_1 < P_2x$
$P_1y + H_1 < P_2y$
$P_1z + D_1 < P_2z$

Na implementação, repare que basta que uma destas condições seja satisfeita, e é garantido que a colisão não ocorreu.

3.1.3 Detecção de Colisão por OBB

O termo OBB se refere a *Oriented Bounding Box*. Diferentemente das AABB's, as arestas dessas caixas não são paralelas ao eixo, são paralelas a um vetor de orientação que representa a rotação da OBB. A quantidade de variações de posições de uma caixa em relação a outra exige um algoritmo mais complexo que o de caixas alinhadas. Uma solução para este cálculo de colisão é o uso do teorema do eixo de separação [Got00].

3.1.4 Teorema do Eixo de Separação

O SAT (Separating Axis Theorem) diz que se for possível passar uma reta por entre dois polígonos convexos sem que esta se colida com eles, então não há colisão. Ou seja, se não for possível achar esta reta de separação, estes polígonos colidem. A idéia pode ser expandida para o 3D. Temos que se existe um plano que separe totalmente dois poliedros no R^3, eles não se colidem. Veja graficamente nas imagens 38 e 39.

Poliedros podem ser projetados em um eixo via produto escalar. Esta projeção resulta num intervalo, que representa a extensão deste poliedro na direção do eixo. Se ambos os poliedros em questão forem projetados no eixo testado (possível eixo de separação), e os dois intervalos obtidos não se interceptarem, conclui-se que os polígonos não se colidem. O algoritmo parece simples, mas a tarefa de achar candidatos relevantes a plano de separação não é simples.
Devem ser testadas inicialmente seis possibilidades de planos:

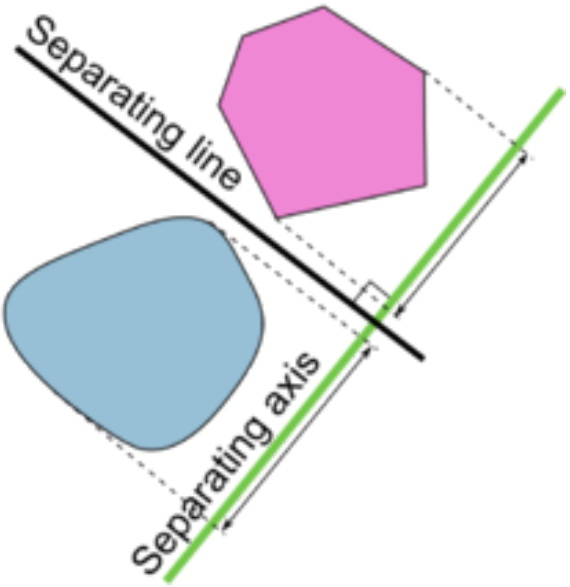

Figura 38: Teorema do Eixo de Separação

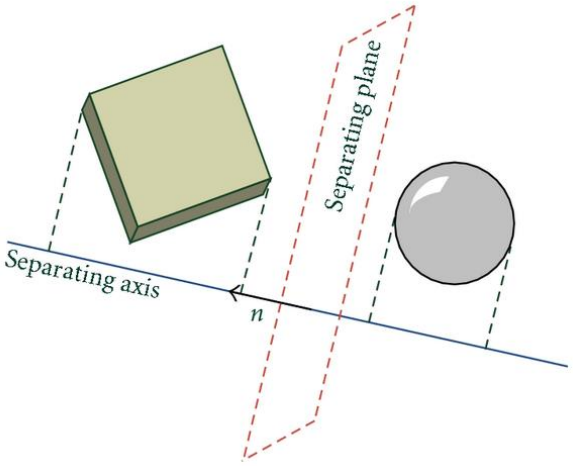

Figura 39: Teorema do Eixo de Separação para o caso 3D

P_A1, plano que possui uma das faces da caixa A, não paralela a P_A2 ou a P_A3
P_A2, plano que possui uma das faces da caixa A, não paralela a P_A1 ou a P_A3
P_A3, plano que possui uma das faces da caixa A, não paralela a P_A1 ou a P_A2
P_B1, plano que possui uma das faces da caixa A, não paralela a P_B2 ou a P_B3
P_B2, plano que possui uma das faces da caixa A, não paralela a P_B1 ou a P_B3
P_B3, plano que possui uma das faces da caixa A, não paralela a P_B1 ou a P_B2

Ou seja, planos referentes às 3 faces não paralelas de cada bloco. Apenas uma de cada par de faces paralelas deve ser considerada no cálculo, pois a outra compartilha da mesma normal em módulo, e o teste SAT não considera o sentido do eixo.

Esses testes não são suficientes para garantir que duas caixas não colidam. O teste falha-

ria caso as O.B.B.'s estivessem próximas por duas arestas ortogonais entre si. Para levar em conta este caso, são adicionados 9 outros testes SAT, que são formados a partir da combinação de pares de normais faciais de duas O.O.B.'s por produto vetorial.
Seja A_n uma normal de face da caixa A e B_n uma normal de face da caixa B, calcular T_i consiste em testar se A_n X B_n é um eixo de separação. Como consideramos 3 normais de cada caixa, temos ao todo 9 cálculos, e i então varia de 1 a 9.
Basta que esses 15 testes sejam executados. Caso algum dos planos resultantes for um eixo de separação, o algoritmo pára e retorna que as caixas não se colidem. Em contrapartida, se nenhum dos 15 planos for um eixo de separação, então as caixas colidem.

3.1.5 Detecção de Colisão entre Boxes e Esferas

Quando dois objetos distintos possuem geometria em forma de esfera e paralelepípedo, temos de tratar de forma diferente dos testes explícitos anteriormente. Mesmo assim, na prática, não há mudança significativa na técnica envolvida.

Considere duas superfícies geométricas, a primeira é uma esfera de raio R_1 e posição em relação ao centro $\overrightarrow{C_1}$. A segunda é um AABB onde $\overrightarrow{P_2}$ representa seu canto superior esquerdo e L_2, H_2, e D_2 são, respectivamente, suas dimensões de largura, altura e comprimento.
Primeiramente, devemos detectar se a posição da esfera em relação ao AABB se encontra dentro dos três seguintes casos:
$C_1x \geq P_2x$ e $C_1x \leq P_2x + L_2$ e $C_1y \geq P_2y$ e $C_1y \leq P_2y + H_2$ ou (1)
$C_1x \geq P_2x$ e $C_1x \leq P_2x + L_2$ e $C_1z \geq P_2z$ e $C_1z \leq P_2z + D_2$ ou (2)
$C_1y \geq P_2y$ e $C_1y \leq P_2y + H_2$ e $C_1z \geq P_2z$ e $C_1z \leq P_2z + D_2$ (3)
Em caso afirmativo da primeira condição (1), a esfera pode ser testada como um AABB, da seguinte forma:
$C_1z + R_1 \leq P_2z + D_2$ e
$C_1z + R_1 \geq P_2z$
Em caso afirmativo da segunda condição (2):
$C_1y + R_1 \leq P_2y + H_2$ e
$C_1y + R_1 \geq P_2y$
Equivalentemente, no caso afirmativo da terceira condição (3):
$C_1x + R_1 \leq P_2x + L_2$ e
$C_1x + R_1 \geq P_2x$
Em caso negativo de todas as condições, devemos descobrir o vértice do AABB mais próximo da esfera e testa-lo de acordo com a simples equação de colisão entre ponto e Esfera:

$|\overrightarrow{P}$ - $\overrightarrow{C_1}| \leq R_1$

Observe que os três testes iniciais equivalem a detectar se a projeção da esfera sobre o AABB se encontra dentro das dimensões do próprio AABB (veja a Figura 40). Caso não se encontre, aplicamos um teste de colisão entre o ponto do box mais próximo da esfera.

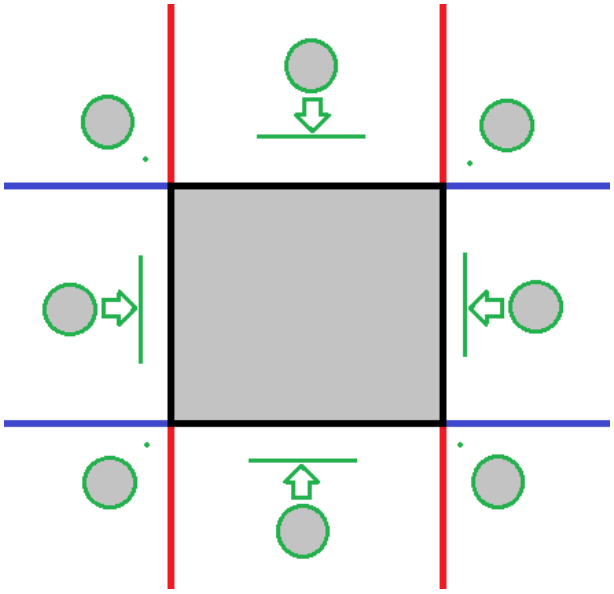

Figura 40: Representação visual bi-dimensional da detecção de colisão entre Esferas e Boxes

O caso de colisão de esferas contra OBBs é equivalente. A única evidente diferença é o eixo que, antes, era estático, agora é dinamicamente alterado em função do seu vetor de orientação.

3.2 Resposta de Colisão

É evidente que detectar a colisão entre dois objetos é a metade do trabalho. A outra metade, considerando o fato de precisar funcionar em tempo real, é minimizar os casos onde dois corpos ocupam o mesmo lugar, isso requer uma translação dos corpos até a área comum entre ambos seja bem próxima a um único ponto. Essa etapa nós chamamos de Resposta de Colisão [VVB08].

Há diversas formas de resposta de colisão, algumas bastante precisas porém custosas em termos de processamento computacional, outras menos precisas porém mais baratas. Veremos algumas dessas formas e qual foi adotada pela URGE.

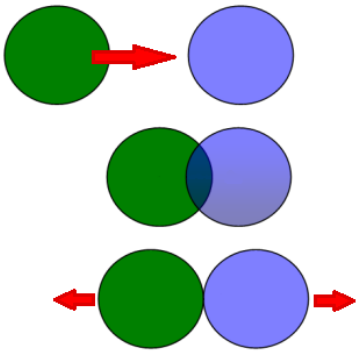

Figura 41: Representação de dos objetos ocupando o mesmo espaço e, posteriormente, realizando a resposta de colisão

3.2.1 Normal de Colisão

Antes de tudo, para efetuarmos a resposta de colisão, precisamos conhecer a normal ao plano de colisão entre os objetos em questão, esse plano segue o mesmo raciocínio do eixo de separação visto anteriormente, isto é, é um plano que separa a colisão entre dois corpos.

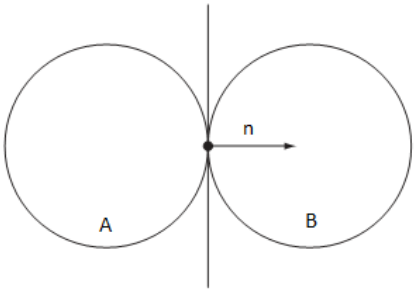

Figura 42: Plano de colisão entre dois objetos e sua normal

No caso de corpos com geometria esférica, a normal de colisão é facilmente encontrada pelo vertor normalizado que liga os centros de ambos os corpos.
Para AABB, as normais são seis constantes referentes às faces do paralelepípedos.
E por último, para OBBs as normais também são seis referentes às faces, entretanto elas são rotacionadas pelo vetor de orientação da OBB.

Entendido como devemos proceder para encontrar a normal de colisão, podemos agora efetuar a resposta usando diferentes técnicas.

3.2.2 Resposta por Translação Direta

A forma mais intuitiva e simples de efetuarmos a resposta de colisão é expulsarmos os objetos diretamente em função da normal de colisão. Em outras palavras, caso a colisão seja detectada, devemos descobrir a normal de colisão (vetor unitário) e transladar os corpos diretamente em função das suas respectivas normais.

Esse método é bastante simples e barato do ponto de vista de processamento. Porém, alguns casos de colisões podem resultar numa penetração de corpos consideravelmente grande, e nesses casos, essa técnica pode demorar alguns instantes para efetuar a expulsão por completo.

3.2.3 Resposta Iterativa por Busca Binária

Uma maneira mais precisa é efetuarmos uma busca binária transladando o objeto até que o coeficiente seja muito pequeno ou até que a área comum entre ambos os corpos seja bem próxima de um ponto.
Considere $\overrightarrow{P_1}$ e $\overrightarrow{P_2}$ como vetores posição de dois corpos abitrários, considere também, $\overrightarrow{N}$ como sendo o vetor normal da colisão entre ambos os corpos. Podemos codificar essa técnica de acordo com o seguinte algoritmo:

$\overrightarrow{P_{12}} = \overrightarrow{P_2}$ - $\overrightarrow{P_1}$

$\overrightarrow{r} = \frac{\frac{\overrightarrow{N}}{|\overrightarrow{N}|} \cdot |\overrightarrow{P_{12}}|}{2}$

enquanto ($|\overrightarrow{r}| > \rho$)faça

{

  se (ambos os corpos estiverem se colidindo por algo próximo de um ponto) então

    fim

  se (ambos os corpos estiverem se colidindo) então

    translade o corpo em relação a $-\overrightarrow{r}$

  se não

    $\overrightarrow{r} = \frac{\overrightarrow{r}}{2}$

    translade o corpo em relação a $\overrightarrow{r}$

}

Essa técnica é bastante precisa, entretanto relativamente custosa para executarmos em um sistema de tempo real, principalmente em máquinas de baixo desempenho de processamento.

3.2.4 Resposta por Distancia de Penetração

A técnica adotada pela URGE não é tão precisa quanto a resposta por busca binária mas é eficiente e produz efeitos bastantes satisfatórios, deixando em piores casos, uma desprezível área comum entre os corpos.
Essa técnica se baseia na detecção da distância de penetração entre os corpos, sendo que ela é calculada em função da geometria dos objetos.
Para duas esferas de raios R_1 e R_2 e centros $\overrightarrow{C_1}$ e $\overrightarrow{C_2}$, a distância de penetração é dada por:

$$dp = (R_1 + R_2) - |\overrightarrow{C_1} - \overrightarrow{C_2}|$$

Para duas AABBs, a distância de penetração é considerada a menor das três possíveis distâncias relacionadas as três dimensões X, Y e Z ais quais os boxes estão se sobrepondo.
Para OBBs, a distância de penetração é dada pelo módulo da diferença entre a soma dos vetores de orientação dos OBBs projetada na reta perpendicular ao eixo de separação, e o vetor que liga os centros de ambos também projetado na mesma reta.

Dessa forma a translação da expulsão dos corpos pode ser feita em uma linha de código para cada corpo físico. Ela deve ser em função do vetor de penetração $\overrightarrow{vp}$, dado por:

$$\overrightarrow{vp} = \frac{\overrightarrow{N}}{|\overrightarrow{N}|} \times dp$$

Sendo $\overrightarrow{N}$ a normal de colisão e dp a distância de penetração.
O vetor de penetração deve ser reduzido pela metade, para então transladar o objeto em questão, pois o outro objeto também deverá fazer o mesmo, portanto para não haver uma translação duas vezes maior que o esperado, deve-se dividir $\overrightarrow{vp}$ pela metade.

3.3 Simulação Dinâmica Baseada em Impulso

Infelizmente, efetuar resposta de colisão não leva em conta a movimentação física dos objetos, isto é, ao colidir precisamos modificar, segundo as leis da física, as velocidades dos corpos em questão. Na URGE, esse cálculo é efetuado em função de um conceito físico comum, o momento linear.
Por esse motivo, a técnica de interação física adotada é a Simulação Baseada em Impulso [Mir96]. Essa técnica considera que, no instante próximo à colisão eminente, os valores das forças e posições dos objetos envolvidos permanecem quase constantes, mas a velocidade, nesse momento, sofre uma descontinuidade. Por exemplo, imagine dois objetos com velocidades apontando uma para a outra, o algoritmo propõe que no momento próximo

à colisão, as velocidades mudam para direções contrarias (observe a Figura 43)

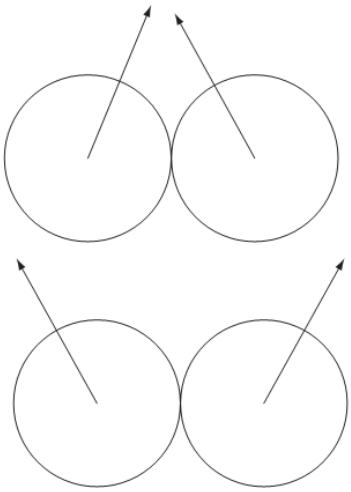

Figura 43: Mudança instantânea nos vetores velocidade no instante da colisão

3.3.1 Movimentação de Objetos

Antes de prosseguir, é interessante apresentarmos como os valores relacionados a movimentação física dos objetos são tratados na URGE.
A cada quadro de animação do jogo, a velocidade deve ser atualizada em função da aceleração, e em caso de colisão, também consideramos outras variáveis relacionadas ao cálculo do impulso linear nas quais iremos discutir.
Já a aceleração é modificada, exclusivamente, pela aceleração da gravidade que pode ser customizada pelo usuário.
Por último, a posição é atualizada quadro a quadro de acordo com a velocidade, e nos casos de colisões, a alteramos em função do algoritmo de resposta de colisão visto anteriormente.
Apesar de não aprofundarmos nesse tópico, existem várias técnicas de movimentação de corpos, mas devido a simplicidade e eficiência do método descrito, não há necessidade de entrarmos a fundo nesse assunto.

3.3.2 O cálculo da velocidade

Considere dois objetos com respectivas velocidades $\overrightarrow{v_1}$ e $\overrightarrow{v_2}$, e normal de colisão entre eles $\overrightarrow{N}$. Queremos modificar a velocidade em função de um impulso linear na direção de $\overrightarrow{N}$, afinal o impulso deve tentar separar os objetos durante a colisão. Entretanto,

a direção da normal depende de qual objeto estamos aplicando tal procedimento. Por exemplo, suponha que os objetos citados anteriormente possuam massa equivalente, então as normais de colisão para cada objeto terão o mesmo módulo mas direções opostas.
Sendo assim, torna-se necessário a definição de uma nova variável j_N que representa o valor escalar que irá multiplicar a normal de colisão para cada objeto envolvido. No exemplo, anterior as velocidades $\overrightarrow{v_1}$ e $\overrightarrow{v_2}$ sofrerão uma alteração direta em relação a $j_N\overrightarrow{N}$ e $-j_N\overrightarrow{N}$.
Sendo assim, agora o problema é encontrar o valor de j_N. Para calcula-lo precisamos da velocidade relativa $\overrightarrow{v_{12}} = \overrightarrow{v_1}$ - $\overrightarrow{v_2}$.

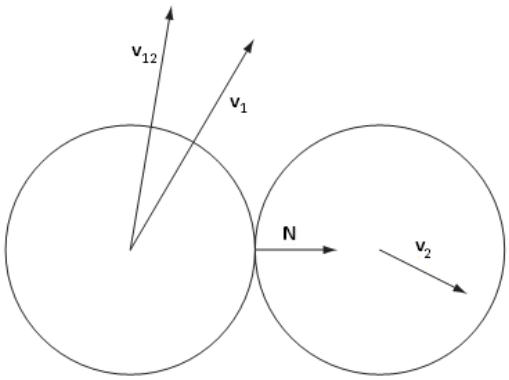

Figura 44: Calculando a velocidade relativa entre dois objetos

Agora precisamos descobrir o quanto da velocidade relativa esta sendo aplicada sobre a normal, ou seja, queremos calcular $\overrightarrow{v_n}$:

$$\overrightarrow{v_n} = (\overrightarrow{v_{12}} \cdot \overrightarrow{N}) \times \overrightarrow{N} \tag{3.1}$$

Repare que $\overrightarrow{v_n}$ é a projeção de $\overrightarrow{v_{12}}$ em $\overrightarrow{N}$ (veja a Figura 45).
Dessa maneira, a velocidade final para o primeiro corpos após a colisão, pode ser considerada simplesmente $-\overrightarrow{v_n}$, pois os corpos devem se "expulsar" após a colisão.
Contudo, ainda faltam dois importantes atributos físicos ainda não considerados, o primeiro é o Coeficiente de Elasticidade de um corpo (K, onde $0.0 < K < 1.0$). Caso K tenda a zero, a colisão se torna completamente inelástica, e a intensidade da velocidade resultante tende a zero, mas se K for um valor bem próximo de 1.0, a colisão é elástica. Portanto vamos introduzir a velocidade final, agora considerando o coeficiente de elasticidade:

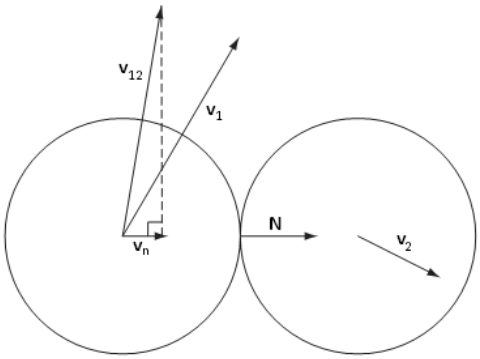

Figura 45: Calculando a influência da velocidade relativa na normal de colisão

$$\overrightarrow{v_{fn}} = -\overrightarrow{v_n} \times K \tag{3.2}$$

Podemos re-escrever a fórmula de acordo com os cálculos efetuados anteriormente:

$$(\overrightarrow{v_{f1}} - \overrightarrow{v_{f2}}) \cdot \overrightarrow{N} = -(\overrightarrow{v_1} - \overrightarrow{v_2}) \cdot \overrightarrow{N} \times K \tag{3.3}$$

Falta mais um atributo físico, a massa. Em nosso cenário, considere m_1 e m_2 as respectivas massas do objeto um e do objeto dois. Para incluirmos tais atributos em nossas equações, devemos nos lembrar que, em um sistema físico conservativo, como é o cenário de toda a interação física da URGE (não estamos interessado em simular dissipação da energia, seja por calor, som ou qualquer outro fator), a energia inicial é igual à energia após a colisão, e o mesmo se aplica ao momento linear, logo:

$$m_1\overrightarrow{v_1} + j_N\overrightarrow{N} = m_1\overrightarrow{v_{f1}} \tag{3.4}$$

Logo podemos calcular $\overrightarrow{v_{f1}}$ como sendo:

$$\overrightarrow{v_{f1}} = \overrightarrow{v_1} + \frac{j_N\overrightarrow{N}}{m_1} \tag{3.5}$$

O mesmo se aplica para o segundo objeto:

$$m_2\overrightarrow{v_2} - j_N\overrightarrow{N} = m_2\overrightarrow{v_{f2}} \tag{3.6}$$

E então:

$$\overrightarrow{v_{f2}} = \overrightarrow{v_2} - \frac{j_N\overrightarrow{N}}{m_2} \tag{3.7}$$

Agora nós temos todas as fórmulas necessárias para calcular j_N, considerando a massa e

o coeficiente de elasticidade. Substituindo as equações 3.5 e 3.7 na equação 3.3, podemos chegar a seguinte fórmula:

$$j_N = \frac{-(1+K) \times (\overrightarrow{v_{12}} \cdot \overrightarrow{N})}{\overrightarrow{N} \cdot \overrightarrow{N} \times (\frac{1}{m_1} + \frac{1}{m_2})} \tag{3.8}$$

Na Figura 46 podemos observar geometricamente as velocidades finais $\overrightarrow{v_{f1}}$ e $\overrightarrow{v_{f2}}$ dos objetos em questão.

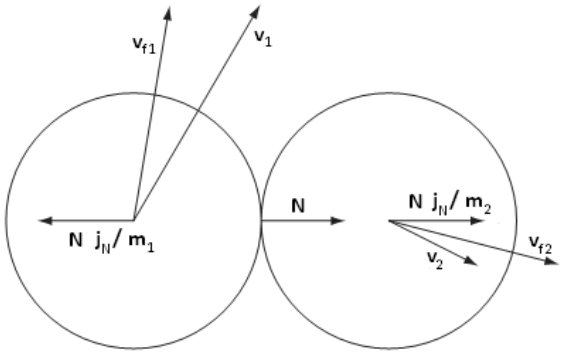

Figura 46: Calculando a influência da velocidade relativa na normal de colisão

3.4 Conclusão sobre Simulação Física

Neste Capítulo, foi apresentado as técnicas envolvidas no processo de criação do Sistema de Simulação Física da URGE. Vimos que seu objetivo não é calcular com exatidão numérica toda a interação entre corpos, mas preservar a total coerência com a realidade e, principalmente, funcionar em tempo real.
Para atingir tal objetivo foi necessário dividir o problema em três etapas:

- Detecção de Colisão entre superfícies geométricas. Aproximando a geometria dos objetos de uma cena à superfícies mais simples, podemos detectar a colisão com eficiência e coerência;
- Resposta de Colisão. Uma vez detectada a colisão, realiza-se a expulsão dos objetos, a fim de impedir que ambos ocupem o mesmo espaço;
- Atualização da velocidade em função do impulso. Após a resposta devemos atualizar a velocidade dos corpos para simularmos a interação física do mundo real.

Essa modelagem para o problema permite uma performance suficiente para funcionar desde computadores mais atuais até máquinas de baixo desempenho de processamento, além de simular de forma fiel à realidade, o comportamento físico dos objetos presentes num cenário de um jogo.

4 Gerenciamento de Cena

Juntando os dois grandes núcleos da engine (visualização e física), é possível criar entidades visuais fies à realidade e que, ao mesmo tempo, interagem fisicamente com outras entidades. Isso significa que com ambos sistemas já estamos habilitados a criar um jogo. Infelizmente, caso partíssemos para a programação de um jogos apenas com tais recursos, iríamos nos deparar, de imediato, com um sério problema de lentidão causado por uma série de *overhaeds* (processamento desnecessário). Por exemplo, imagine um cenário na qual existem dezenas de entidades físicas e visíveis, podemos claramente considerar overhead, o processamento necessário para desenhar um objeto que não está sendo visto pelo observador, ou o processamento necessário para efetuar a colisão entre objetos consideravelmente distantes entre si.

Portanto, com o intuito de solucionar esse tipo de problema, a URGE possui um sistema que integra e gerencia ambos os núcleos, esse é o sistema de gerenciamento de cena (ou Gerenciador de Cena). Tal sistema está encapsulado numa entidade da URGE chamada *Scenario* que é, justamente, a responsável por testes de visibilidade de objetos na cena e otimização física espacial.
No presente capítulo, apresentaremos as técnicas, por trás do gerenciador de cena, as quais permitem que um jogo funcione em tempo real mesmo com uma significativa quantidade de entidades, minimizando todos os processamentos desnecessários.

4.1 O Funcionamento do Gerênciador de Cena

O objetivo de um gerenciador de cena é bastante simples: eliminar o máximo possível de objetos visíveis, que não estão sendo vistos pelo observador, de serem renderizados e o mesmo para objetos físicos que evidentemente não vão se colidir, deixando como *output*, apenas objetos potencialmente visíveis ou, no caso do sistema de física, com colisão eminente a outro objeto.
Lembre-se que os objetos em questão, podem, e de fato representam qualquer entidade

física ou visual de um jogo, desde paredes e terrenos até o próprio jogador.

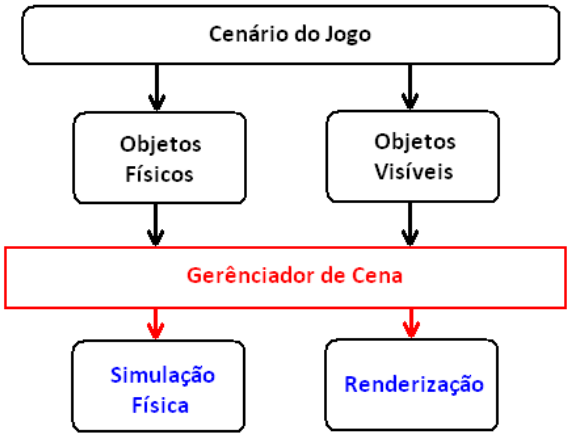

Figura 47: Fluxograma Representativo do papel do Sistema de Gerenciamento de Cena

As técnicas envolvidas na otimização de processamento de objetos numa cena devem ser em função da natureza do cenário de um jogo específico. Isso quer dizer que não existe um algoritmo ou receita para otimizar da melhor forma possível cenas de qualquer natureza.
No caso da URGE, é importante ressaltar que o gerenciador de cena foi preparado para a otimização de cenários abertos, isto não implica na impossibilidade de otimizar cenários fechados, apenas significa que as técnicas aqui apresentadas possuem melhor performance em ambientes abertos, como terrenos, ilhas ou campos.

4.2 View Frustum Culling

A primeira técnica utilizada pela engine é bastante simples e usada desde os primeiros grandes jogos 3D. O algoritmo do *View Frustum Culling* consiste em evitar desenhar tudo que não está dentro do campo de visão do observador [AM00]. O campo de visão é denominado de *View Frustum* e o ato de evitar desenhar elementos fora do mesmo denomina-se *Culling*. A figura 48 representa o processo de otimização dos elementos visuais fora da área de visão do observador.

Existem várias formas de cumprir tal efeito. No caso da URGE, a geometria dos objetos da cena são aproximados para AABBs ou esferas, por questão de eficiência, e

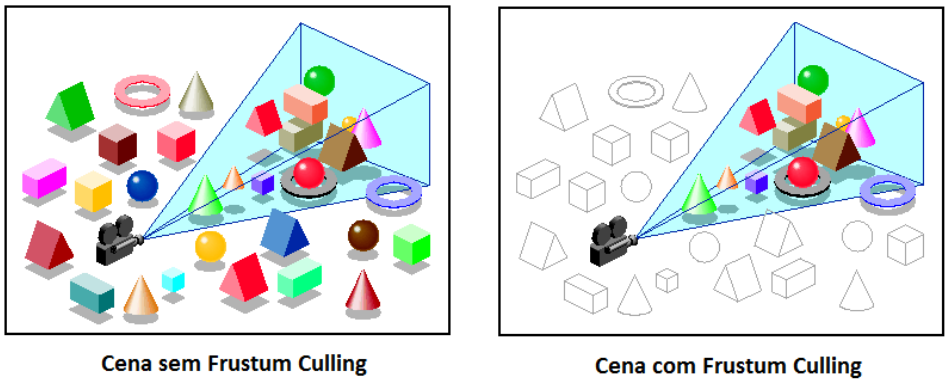

Figura 48: Comparação de duas representações de cena explicitando o uso do view frustum culling.

testados contra os planos gerados pelo frustum. Portanto, esse algoritmo constitui-se em duas etapas: a primeira é extrair os seis planos gerados pelo campo de visão, e a segunda é testar a geometria aproximada dos objetos em cena contra os planos.

4.2.1 Extração dos Planos do Frustum

Para extrairmos os seis planos formados pela geometria do campo de visão, precisamos recorrer a conceitos básicos de álgebra linear. Inicialmente, por questão de eficiência, a informação armazenada, escolhida pela URGE, para definir um plano será a sua normal (lembrando que o vetor normal a um plano pode ser usado para descreve-lo).
Considere, $\overrightarrow{p}$ como sendo a posição do observador, $\overrightarrow{d}$ como a direção do mesmo, $\overrightarrow{up}$ como sendo o vetor ortogonal à $\overrightarrow{d}$ que aponta para o sentido superior ao campo de visão e $\overrightarrow{right}$ como sendo o vetor resultante do produto vetorial, respectivamente, entre $\overrightarrow{d}$ e $\overrightarrow{up}$. Esses foram os vetores que definem o observador, agora considere *Hnear*, *Wnear* e *Dnear* como sendo, respectivamente, a altura, largura e distância entre o observador e o plano mais proximo (*near*). Da mesma forma, considere *Hfar*, *Wfar* e *Dfar* como sendo, respectivamente, a altura, largura e distância entre o observador e o plano mais distante (*far*).

Agora vamos introduzir duas novas variáveis: $\overrightarrow{nc}$ e $\overrightarrow{fc}$, que representam os pontos de intersecção entre a reta formada pelo vetor $\overrightarrow{d}$ e os planos mais próximo e mais distante, respectivamente. Podemos descobri-los, sem dificuldades, de acordo com as seguintes equações:

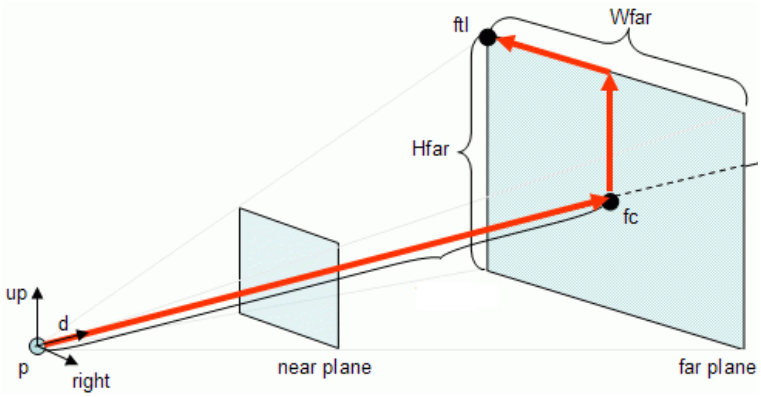

Figura 49: Representação visual dos vetores envolvidos no cálculo do frustum

$$\overrightarrow{nc} = \overrightarrow{p} + \overrightarrow{d} \times Dnear$$
$$\overrightarrow{fc} = \overrightarrow{p} + \overrightarrow{d} \times Dfar$$

Agora precisamos descobrir os seis vetores normais que definem os seis planos. O vetor normal ao plano near é o próprio $\overrightarrow{d}$, consequentemente o vetor normal ao plano *far* é $-\overrightarrow{d}$. Para calcular o vetor normal ao plano direito é preciso descobrir os dois vetores que o originam, podemos concluir geometricamente que um deles é o vetor $\overrightarrow{up}$ do observador, e o outro é um vetor $(\overrightarrow{q_d})$ que tangencia a lateral direita do plano near, esse vetor pode ser calculado da seguinte maneira:

$$\overrightarrow{q_d} = (\overrightarrow{nc} + \overrightarrow{right} \times \frac{Wnear}{2}) - \overrightarrow{p} \tag{4.1}$$

Sendo assim, o vetor normal ao plano direito $(\overrightarrow{n_d})$ é dado pelo produto vetorial entre $\overrightarrow{up}$ e o versor de $\overrightarrow{q_d}$:

$$\overrightarrow{n_d} = \overrightarrow{up} \times (\frac{\overrightarrow{q_d}}{|\overrightarrow{q_d}|}) \tag{4.2}$$

A normal ao plano esquerdo pode ser obtida de forma equivalente:

$$\overrightarrow{q_e} = (\overrightarrow{nc} - \overrightarrow{right} \times \frac{Wnear}{2}) - \overrightarrow{p} \tag{4.3}$$

$$\overrightarrow{n_e} = (\frac{\overrightarrow{q_e}}{|\overrightarrow{q_e}|}) \times \overrightarrow{up} \tag{4.4}$$

A normal ao plano superior também parte do mesmo princípio, isto é, devemos descobrir os dois vetores que dão origem a tal plano, e esses são $\overrightarrow{right}$ e $\overrightarrow{q_s}$, dado por:

$$\overrightarrow{q_s} = (\overrightarrow{nc} + \overrightarrow{up} \times \frac{Hnear}{2}) - \overrightarrow{p} \tag{4.5}$$

Logo o vetor normal à face superior ($\overrightarrow{n_s}$) é:

$$\overrightarrow{n_s} = (\frac{\overrightarrow{q_s}}{|\overrightarrow{q_s}|}) \times \overrightarrow{right} \tag{4.6}$$

E por último, a normal da face inferior, que é semelhante ao cálculo anterior:

$$\overrightarrow{q_i} = (\overrightarrow{nc} - \overrightarrow{up} \times \frac{Hnear}{2}) - \overrightarrow{p} \tag{4.7}$$

$$\overrightarrow{n_i} = \overrightarrow{right} \times (\frac{\overrightarrow{q_i}}{|\overrightarrow{q_i}|}) \tag{4.8}$$

Tendo armazenado os planos pelas suas respectivas normais, agora será possível efetuar os testes de colisão necessários para verificarmos se um objeto está dentro do campo de visão do observador.

4.2.2 Detecção de Colisão: Frustum - Ponto

Para testarmos a colisão de um ponto arbitrário $\overrightarrow{p}$ contra o campo de visão do observador, com o intuito de detectar se $\overrightarrow{p}$ está dentro, basta efetuar seis testes de colisão entre ponto e plano, um para cada plano que compõe pelo frustum. Em outras palavras, se a distância entre $\overrightarrow{p}$ e qualquer plano for negativa, sabemos que o ponto está fora do frustum. Lembrando que a distância entre um ponto $\overrightarrow{p}$ e um plano cuja normal é $\overrightarrow{n}$ pode ser encontrado de acordo com a seguinte formula:

$$\overrightarrow{distancia} = \overrightarrow{p} * \overrightarrow{n} + |\overrightarrow{p}| \tag{4.9}$$

4.2.3 Detecção de Colisão: Frustum - Esfera

O teste de colisão entre uma esfera e o frustum parte do mesmo princípio que o algoritmo anterior, isto é, precisamos comparar a distância entre os seis planos e o raio esfera. O pseudo-código abaixo exemplifica tal procedimento:

```
booleano esferaDentroDeFrustum(Esfera E, Frustum F)
{
    para cada plano P de F {
        distancia = produtoEscalar(E.centro, P.normal) + modulo(E.centro)
        se (distancia < - E.raio)
            retorna falso
```

```
    }
    retorna verdadeiro
}
```

4.2.4 Detecção de Colisão: Frustum - AABB

Testar a colisão entre um AABB e o Frustum é mais simples do que o algoritmo anterior. Precisamos apenas testar os oito vértices do AABB usando o algoritmo de teste de colisão ponto - frustum, se todos os vértices falharem no teste, significa que o box está fora do frustum, por outro lado, se pelo menos um dos pontos estiver dentro do frustum, a engine pode considerar que o AABB está parcialmente dentro do campo de visão.
A URGE não sub-divide um objeto visual com o intuito de desenhar apenas parte dele, ou seja, se o objeto estiver parcialmente visível, então ele será renderizado por completo.

4.3 Grafo de Cena

É perfeitamente comum um cenário de um jogo ser constituído por dezenas de objetos e, possivelmente, encontramos grande parte desses elementos em uma porção espacial específica. Então, afim de evitar overheads, devemos agrupar tais objetos de acordo com sua posição espacial, e esses grupos podem ser agrupados em novos grupos (como se fosse uma redução de escopo), e assim por diante...

Muitas engines, agrupam objetos de forma estática, em função do cenário em questão. Por exemplo, imagine um cenário na qual representa uma casa de dois quartos em um terreno, se o observador encontrar-se dentro da casa mas fora dos quartos, não há a necessidade de desenhar objetos dentro dos mesmos, portanto podemos classifica-los como um grupo da cena em questão, e partindo-se desse raciocínio, devemos fazer o mesmo para a casa. Observe a representação desse exemplo na figura 50.

Repare que a casa será classificada como um super-grupo contendo os dois quartos, já os objetos E e F, por estarem fora da casa e em localizações arbitrárias, não serão agrupados. Essa forma de agrupamento gera um grafo, mais especificamente nesse caso, uma árvore conhecida como Grafo de Cena [Ast06b]. Veja que a figura 51 representa o grafo de cena do nosso exemplo.

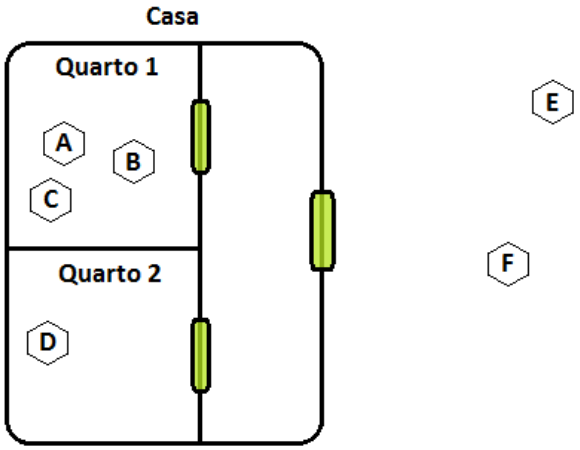

Figura 50: Representação de um cenário arbitrário de um jogo

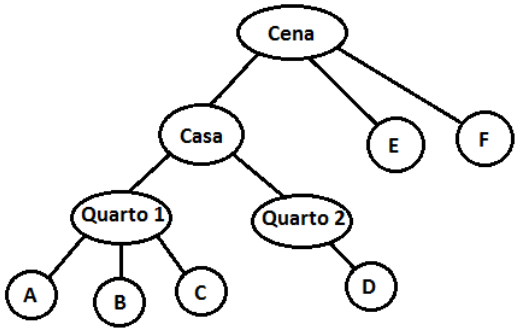

Figura 51: Grafo de cena do exemplo em questão

O exemplo citado, mostra um caso de grafo de cena estático, considerando que os objetos não mudam de grupo durante a execução do jogo. Porém, o grafo de cena da URGE é gerado dinamicamente de acordo com o output do seu algoritmo de otimização espacial via Octrees. Esse método será discutido nas próximas seções.

4.4 Quadtrees

Uma forma eficiente e versátil de otimização espacial de processamento gráfico e físico é o algoritmo baseado em Quadtrees [AMHH08]. Seu funcionamento é simples: dividir o cenário em sub-regiões retangulares com o intuito de eliminar processamento irrelevante para o jogo. Esse algoritmo é feito de forma recursiva e possui como output uma árvore chamada Quadtree.

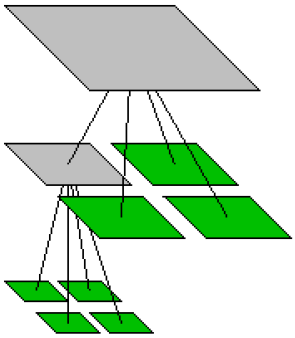

Figura 52: Representação da divisão espacial pelo algoritmo da quadtree

4.4.1 Quadtrees para Otimização de Simulações Físicas

Quadtrees são bastante usadas para evitar processamento desnecessário para realizar testes de colisões físicas, como é o caso de objetos consideravelmente distantes em um mapa. Veja a Figura 53, nela observamos seis objetos independentes em um cenário sendo submetidos a esse procedimento.

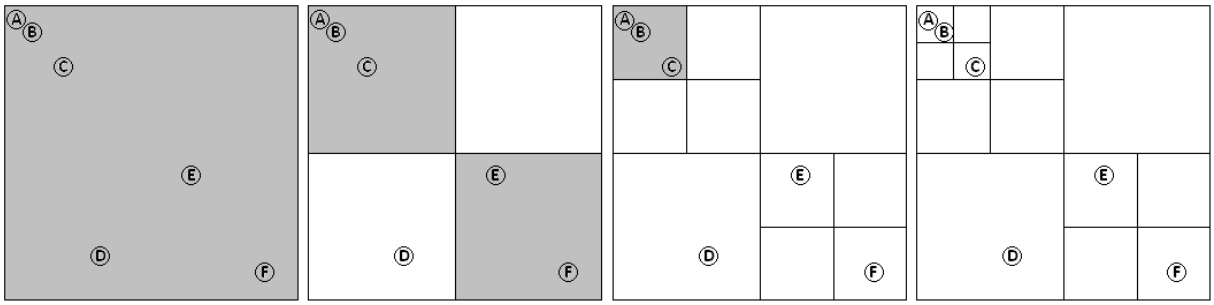

Figura 53: Exemplo do funcionamento do algoritmo da quadtree (profundidade máxima = 3)

Da mesma forma que o view frustum culling, a geometria de cada objeto do cenário

deve ser aproximada para uma forma bastante simples, e nesse caso, tal forma deve ser 2D (em geral usamos retângulos), afinal o algoritmo da quadtree particiona o espaço de forma bi-dimensional.

O primeiro passo do procedimento é dividir o cenário em quatro setores retangulares (não necessariamente do mesmo tamanho) e, então, testa-los contra a geometria aproximada de cada objeto, caso se encontre dentro de um setor, deverá ser armazenado em um nó da árvore. Se um nó estiver armazenando mais de um objeto, ele deverá ser expandido até que haja apenas um objeto para cada nó, ou chegue ao limite máximo de profundidade do algoritmo. A Figura 54 explicita a árvore gerada pelo algoritmo. Veja que usando quadtrees nesse caso, podemos concluir que o núcleo de física precisará aplicar apenas um teste de colisão, entre o objeto A e o objeto B, contra quinze testes de colisões, caso não usemos quadtrees.
Repare também que, com exceção das folhas, todos os nós da árvore possuem quatro filhos, eis também o fato que deu origem ao nome.

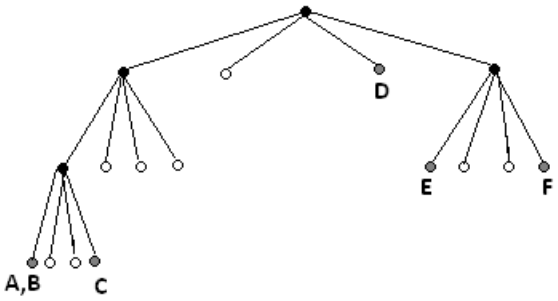

Figura 54: Quadtree resultante do exemplo citado

4.4.2 Quadtrees para Otimização de Renderização de Terrenos

No caso da URGE, Quadtrees são usadas para gerenciar a renderização de terrenos (discutida no capítulo 2). Seu funcionamento, apesar de ter o mesmo comportamento, é um pouco diferente do algoritmo discutido anteriormente [Lue03]. Primeiramente, dessa vez, seu papel não é mais particionar o espaço em função dos objetos contidos no mesmo, agora buscaremos atingir dois novos objetivos: Evitar desenhar seções do terreno que não estão sendo vistas pelo jogador e classificar os diferentes níveis de detalhes em função da distância.
No capítulo 2, ao discutirmos sobre LODs de terrenos, vimos que a engine particiona seus terrenos em quadrados de área NxN, e esse valor é proporcional à profundidade máxima

do algoritmo da quadtree e o valor da área do terreno. Sendo assim, a primeira mudança no algoritmo é o critério de parada, que agora é em função da profundidade máxima ou da falha no teste de colisão. Isso nos leva a segunda mudança, que são os objetos pelas quais vamos testar suas geometrias aproximadas contra as divisões espaciais da quadtree, nesse caso iremos testar apenas um objeto, o view frustum. Em outras palavras, o algoritmo agora testa apenas o campo de visão do observador e enquanto houver colisão, o nó relativo deverá ser expandido até atingir a profundidade maxima ou até não haver mais colisão.

Observe a figura 55, a profundidade maxima nesse caso é três, e a dimensão formada

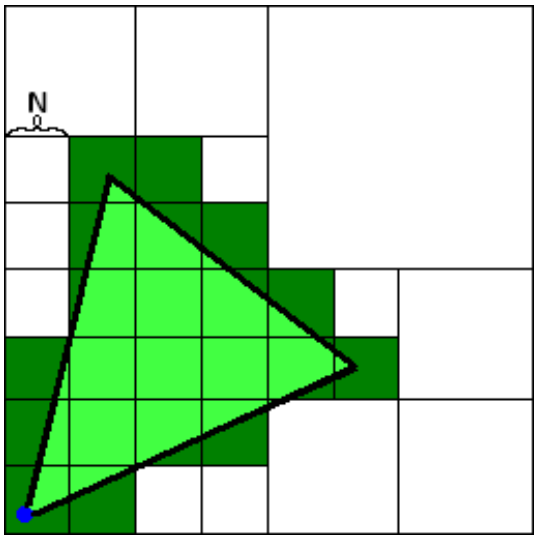

Figura 55: Representação do algoritmo da Quadtree contra o View Frustum, com a finalidade de otimizar a visualização do terreno

pela divisão até tal profundidade define N. Levando em conta que será desenhado apenas a área em destaque, podemos facilmente concluir que esse algoritmo é bastante eficiênte na eliminação de processamento gráfico desnecessário.

4.5 Octrees

Uma vez entendido o funcionamento das quadtrees, podemos introduzir um algoritmo de otimização de cena baseado em uma nova estrutura, chamada *Octrees* [GDD00], ele é basicamente uma extensão do algoritmo baseado em quadtrees para a terceira dimensão. Isto significa que, nesse caso, o procedimento será particionar o espaço em oito regiões relativas ao eixo (x, y, z), ao invés de quatro regiões bi-dimensionais como é o caso da quadtree.

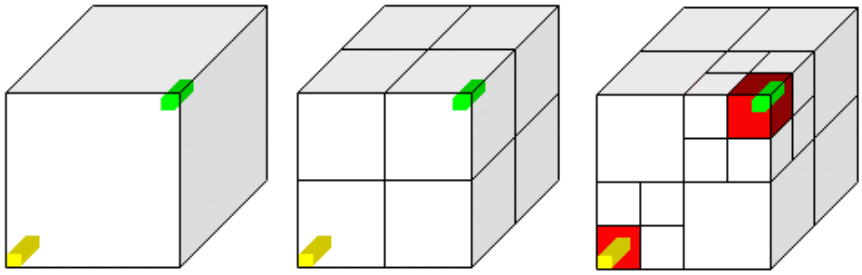

Figura 56: Exemplo do funcionamento do algoritmo da octree (profundidade máxima = 2)

Pelo fato de particionar o cenário de forma tri-dimensional, octrees são usadas pela URGE para eliminar processamento desnecessário por parte do núcleo de física, partindo-se do mesmo funcionamento das otimizações de simulações físicas feitas por quadtrees. É evidente que Octrees são árvores as quais cada nó possui oito filhos (com exceção das folhas), além do que, cada nó representa uma partição do espaço em forma de AABB.

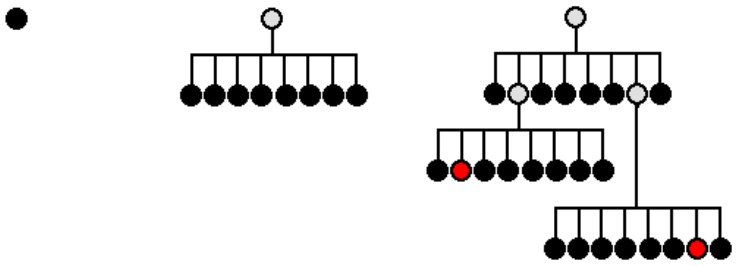

Figura 57: Passo a passo da geração de uma Octree

4.6 Conclusão sobre Gerenciamento de Cena

Sendo assim, vimos no presente capítulo, o funcionamento de técnicas de gerenciamento de cena, objetivando evitar, ao máximo, o processamento gráfico e físico indesejado. Vimos que o view frustum culling, é um algoritmo simples mas eficiente para eliminar desenho de objetos pelos quais não estão sendo vistos pelo jogador. Repare que não é correto aplica-lo para otimizar o sistema de simulação física, pois se isso fosse feito, toda a física de objetos não visíveis pelo jogador seria ignorada.

Para otimizarmos o processamento feito pelo núcleo de física, utilizamos de algoritmos baseados em octrees, que são perfeitos para situações de cenários abertos como campos, terrenos, montanhas...

O único elemento da URGE que possui um tratamento especial é o terreno, que é otimizado por um algoritmo baseados em testes de view frustum contra quadtrees.

A Figura 58 explicíta o funcionamento interno resumido do gerenciador de cena.

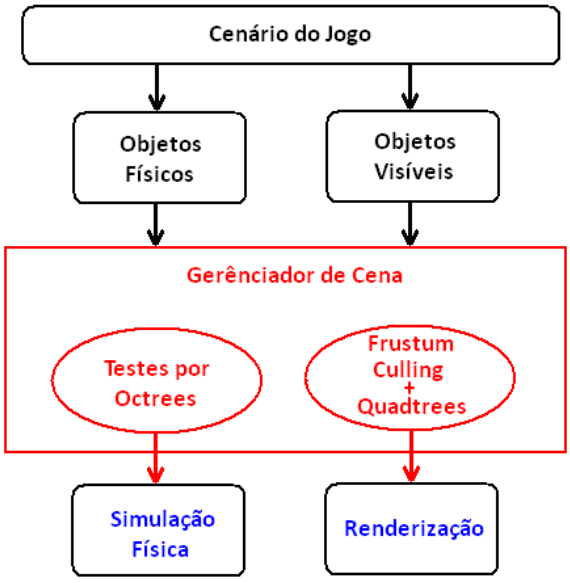

Figura 58: Fluxograma Representativo do papel e funcionamento do gerenciador de cena

Parte II

Desenvolvimento e Origanização da URGE

5 *Processo de criação da URGE*

Uma importante questão é como a equipe desenvolvedora da URGE se organizou para criá-la, isto é, como foram definidas as etapas de projeção, implementação e testes.
Neste Capítulo, veremos a evolução de todo o processo de produção da engine, os modelos de engenharia de software que foram adotados, as diferentes atualizações das modelagens do projeto, o que funcionou e o que foi descartado.

5.1 Evolução da Engenharia de Software

O projeto, inicialmente chamado de "GDP3DE", passou por duas etapas distintas. A primeira etapa durou seis meses (Outubro de 2010 até Março de 2011), contava com oito programadores e teve como método de engenharia de software o Modelo Incremental. Nesse tempo, o projeto não possuiu professor orientador, os alunos se organizaram sozinhos e modelaram o primeiro UML de classes. A equipe organizou e criou as primeiras versões dos núcleos de física e visualização. O projeto, até então, era abrangente, e a intenção do grupo era desenvolver uma engine capaz de criar qualquer jogo em três dimensões.

Já a segunda etapa começou em Março de 2011 e prossegue até hoje. A partir desse ponto, o tamanho da equipe diminuiu para apenas quatro integrantes. Entretanto, nessa época o projeto passou a ser orientado, o que causou boas mudanças, como uma maior integração entre os programadores e, principalmente, a priorização de tarefas.
O modelo de desenvolvimento, adotou características do Modelo Ágil (*Scrum*), isso causou um grande impacto na forma de desenvolvimento dos projetos.

5.1.1 Adaptação do Planning Poker

Etapas de desenvolvimento que antes eram criadas pela sua completude, passaram a ser ponderadas em relação ao custo de desenvolvimento e sua prioridade inspirando-se

na técnica do *Planning Poker*, considerando, também, o risco de futuras mudanças de planejamento.

Planning Poker é um método de estimativa de custo de desenvolvimento que baseia-se no consenso geral dos participantes, buscando não influenciar a decisão na opinião de um membro. Na equipe, essa técnica foi adaptada quando o custo foi estimado segundo discussões a cerca do risco e complexidade do módulo em questão. Ao chegar a um consenso, o valor custo era escolhido de acordo com um elemento da sequência de fibonacci, quanto maior o número, maior o esforço para produzir o elemento em questão.

5.1.2 Adaptação de User Stories

Nessa etapa, a priorização dos recursos da engine, foi adaptada para *User Stories*, uma vez que, a partir dessa mudança, a equipe observou um considerável avanço na agilidade do planejamento e maior coesão na projeção dos módulos da URGE.

Inspirando em User Stories, a equipe passou a planejar o procedimento de priorização no ponto de vista do cliente da URGE, não mais do ponto de vista do desenvolvedor. Como a engine deve ser usada por pessoas menos experientes em programação, todo o processo de priorização não pode ser mais feito em função de requisitos técnicos, ao invés disso, esse processo foi aplicado em função do produto final a qual o cliente tem acesso direto.

5.1.3 Planejamento Contínuo

Com essas mudanças, grandes objetivos foram fragmentados e depois sub-fragmentados em pequenas tarefas com o intuito de simplificar ao maximo o objetivo do passo mais imediato (Planejamento Contínuo), isso possibilitou o desenvolvimento em paralelo de diversos módulos.

Anteriormente era priorizado o desenvolvimento sequencial dos núcleos da engine (um núcleo após o outro) por sua completude, essa forma de desenvolvimento demorava muito para se obter resultados, e quando obtidos, eram bastante específicos a um assunto, por exemplo, no caso do núcleo de visualização o resultado era integralmente sobre tal área. Com as práticas de scrum, a completude de todos os núcleos passou a ser vista como algo “épico”, isto é, um grande feito que só pode ser atingido depois de muito esforço e trabalho, portanto o desenvolvimento do núcleo foi divido em tópicos, e esses eram divididos progressivamente.

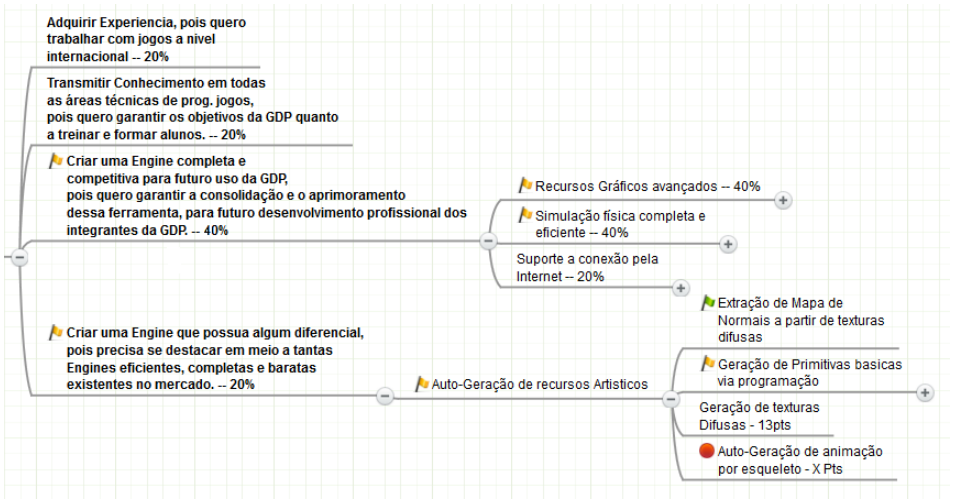

Figura 59: Parte do Modelo Iceberg da URGE

A Figura 59 explicita os feitos épicos nos tópicos mais à esquerda, e histórias de menor complexidade à direita. As porcentagens representam a prioridade em relação aos outros feitos de mesmo nível de profundidade.

Há informações implícitas nesse modelo, os pontos representado por "+" são nós com tópicos cujos objetivos não foram explícitos.

Os tópicos relacionados aos recursos gráficos (visualização) e simulação física foram explorados abaixo na Figura 60.

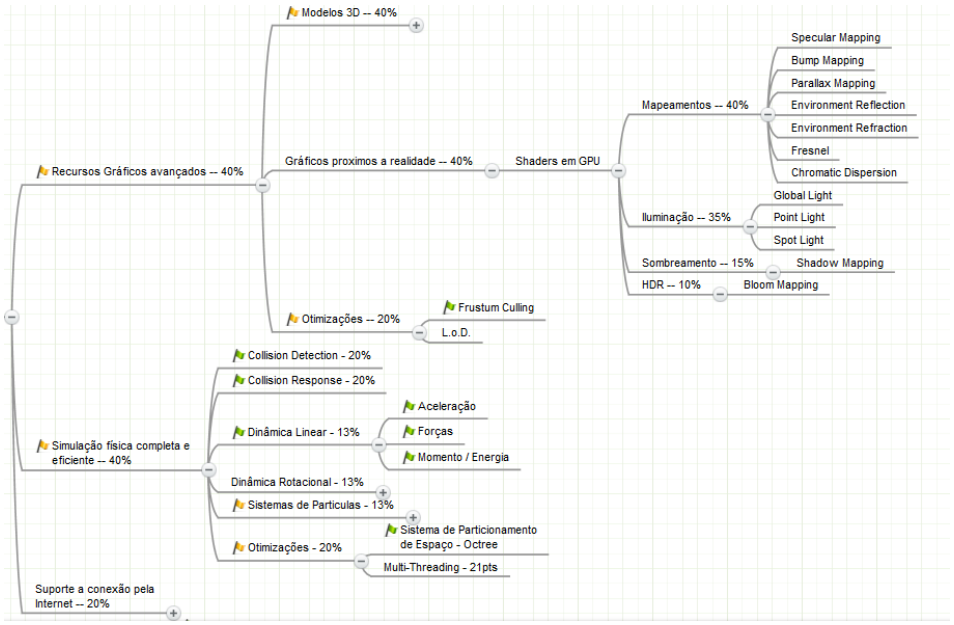

Figura 60: Escopo mais específico do Modelo Iceberg

Repare, por exemplo, que o núcleo de física pode ser dividido em detecção de colisão e resposta de colisão, a parte de detecção pode ser sub-dividida em função das diferentes formas geométricas em que vamos testar a colisão, e assim por diante... Dessa forma foi possível trabalhar em mais de um tópico ao mesmo tempo, o que possibilitou resultados mais rápidos e mais generalizados, ou seja, resultados em diversas áreas da engine.

5.1.4 Inspiração em Scrum

Sendo assim, inspirando-se em elementos de desenvolvimento de software do modelo ágil, o projeto passou a se chamar "URGE". Embora seja possível desenvolver jogos de vários estilos com a URGE, ela passou a ser especializada em apenas um nicho de jogo, os tri-dimensionais de cenário aberto.
A especialização da URGE em apenas um nicho de jogo foi um critério decisivo na para obtermos resultados mais rapidamente. Uma engine capaz de criar qualquer jogo, de qualquer estilo e com qualquer tecnologia existente seria inviável. Afinal, até no mercado não há engines com tal capacidade.

5.2 Evolução da modelagem

A essência da URGE sempre foi a mesma desde a implementação das primeiras classes, mas a modelagem UML foi modificada ao longo do tempo, como consequência da maturação natural do projeto. A primeira versão do diagrama UML de classes foi subdividida em dez módulos, como visto na imagem 61:

Abaixo, podemos ver uma breve descrição de cada módulos:

- Math: *Container* das entidades mais básicas da engine que envolvem matemática, álgebra linear e geometria espacial.
- Physics: Aplicação física das classes de geometria contidas no módulo Math (núcleo de fisica da URGE, descrito no Capítulo 3).
- Objects: Objetos que estariam presentes na cena, por exemplo, primitivas e modelos 3D.
- Environment: Simulação de efeitos presentes em um ambiente, como luz, céu e neblinas.
- Texture: Responsável por carregar texturas em classes do módulo Objects.

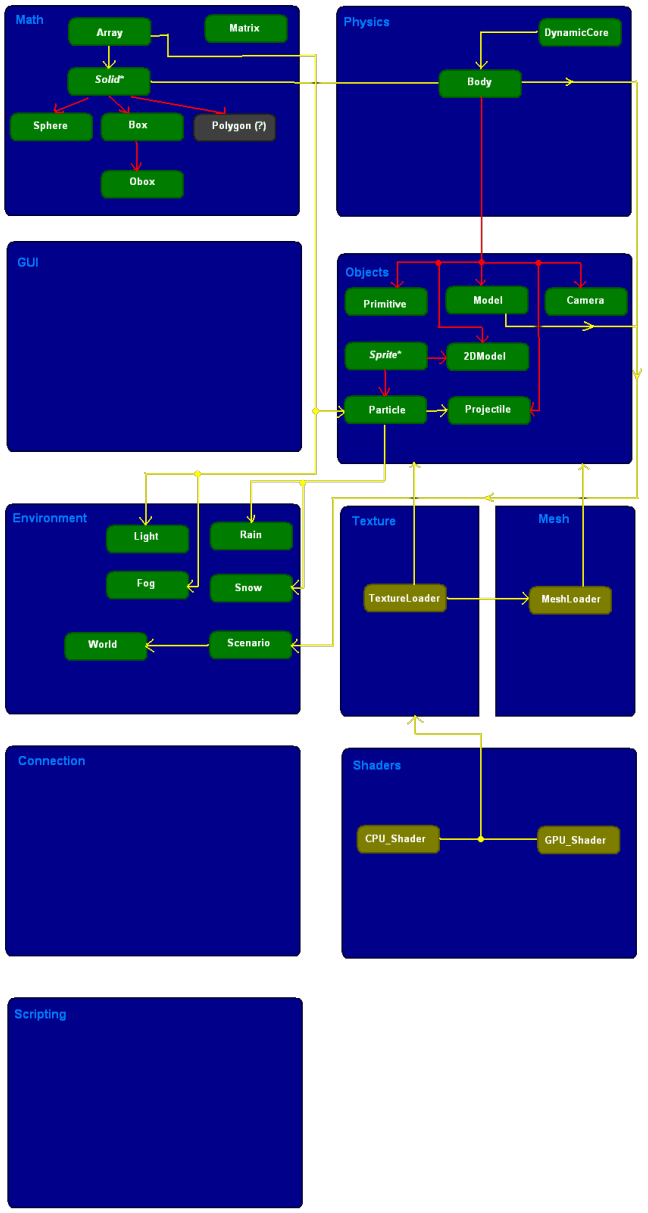

Figura 61: Diagrama UML de classes da URGE (primeira versão).

- Mesh: Responsável por carregar modelos 3D usados pelas classes do módulo Objects.
- Scripting: Classes envolvendo carregamento de scripts e XML.
- Connection: Conexão com a internet.
- GUI: Botões, menus, caixas de texto e outros elementos de interface gráfica.

Os três últimos módulos citados – Scripting, Connection e GUI – tiveram sua necessidade prevista, mas por serem funcionalidades menos prioritárias num contexto de criação de jogos 3D, não tiveram suas classes definidas neste ponto do projeto.

Após a criação primeiras classes, a equipe considerou necessário re-modelar a UML de modo mais organizado (veja a Figura 62). Novos conceitos foram adicionados, e nenhum

módulo foi deixado vazio.

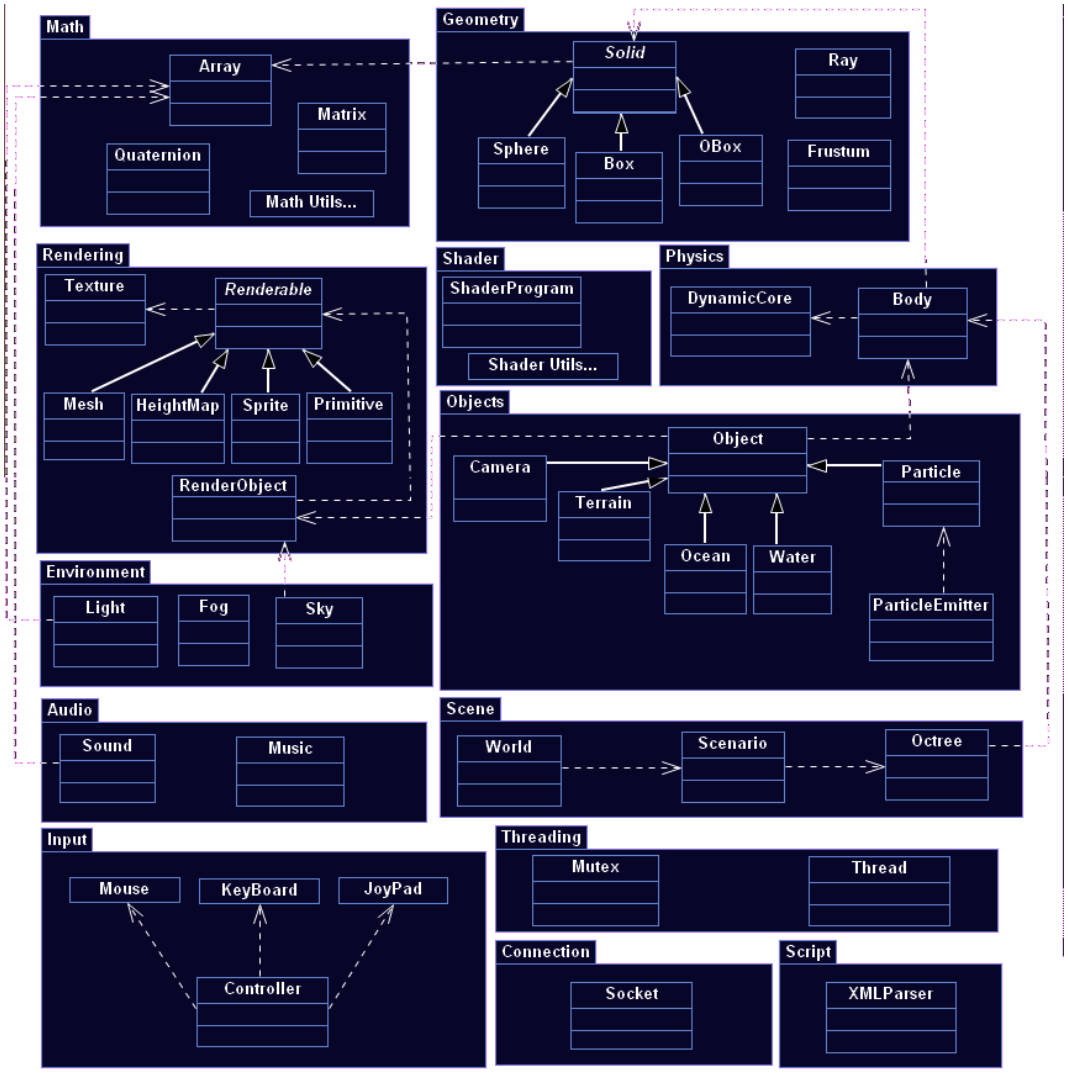

Figura 62: Diagrama UML de classes da URGE (segunda versão).

- Math: Agora contém apenas as entidades e rotinas ligados exclusivamente a matemática e álgebra linear.
- Geometry: Novo módulo, resultante da separação do Math, contém todas as entidades relacionadas a Geometria Espacial.
- Rendering: Entidades ligadas ao sistema de Visualização (Núcleo de Visualização da URGE, descrito no Capítulo 2).
- Shader: Segmento que possibilitou o desenvolvimento de gráficos de alta qualidade, pois sua função é fornecer suporte a programação em placa de vídeo.
- Physics: Aplicação física das classes de geometria contidas no módulo geometry.

- Objects: Classes de básicas da URGE na qual todo o desenvolvimento por parte dos usuários seriam sobre tais entidades.
- Audio: Classes responsáveis por sons.
- Scene: Classes que garantem o gerenciamento da cena do jogo.
- Input: Módulo de controle dos comandos de entrada.
- Threading: Módulo de programação concorrente.
- Connection: Conexão com a internet.
- Script: Carregamento e interpretação de XML.

As duas primeiras versões da UML foram criadas no período anterior à orientação. As classes da segunda versão foram desenvolvidas ao longo de 2011, e apenas no segundo semestre do ano a equipe decidiu fazer uma nova atualização. Poucas modificações ocorreram.

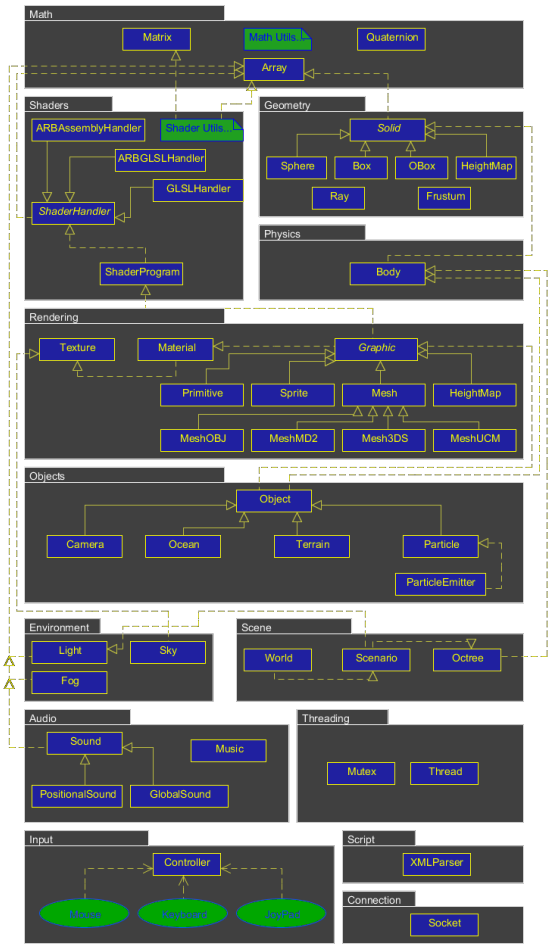

Figura 63: Diagrama UML de classes da URGE (terceira versão).

- O módulo de Shaders foi subdividido para garantir suporte a diferentes linguagens de programação em GPU.
- Uma classe Heightmap também foi criada no núcleo de rendering para representar a parte visual de um terreno.
- Novas classes no núcleo rendering, para carregar tipos diferentes de modelos 3D.
- O material dos objetos passou a controlar todas as propriedades visuais de um objeto.
- O som teria então opção de ser global ou posicional.
- As primitivas passaram a ser feitos por VBO.

Essa versão já estava bastante completa e amadurecida, contudo houve uma última necessidade de mudança na modelagem para aumentarmos a simplicidade de uso e a flexibilidade em futuras mudanças na engine. Isso gerou o último e mais atual diagrama UML da URGE:

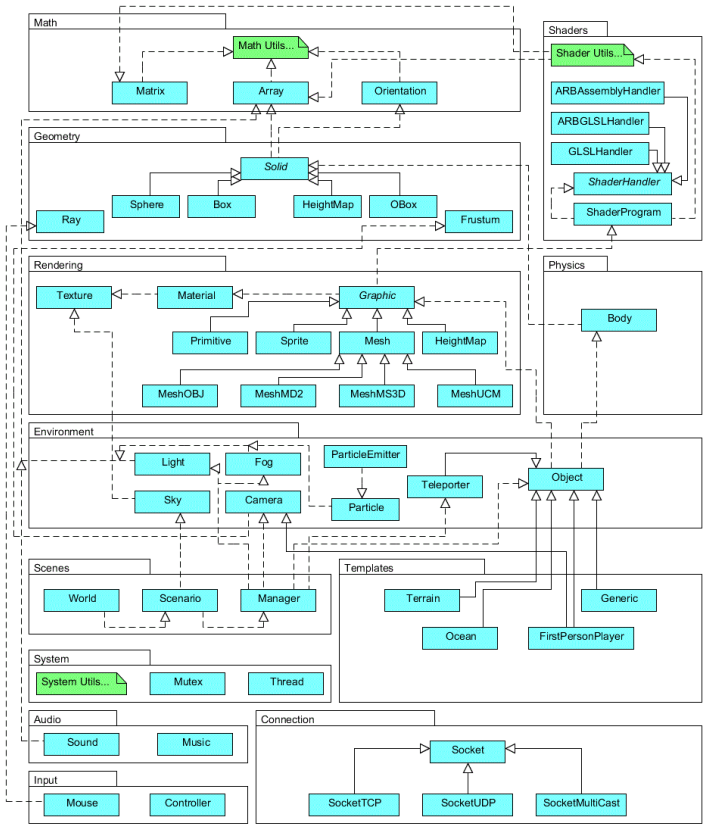

Figura 64: Atual e última versão do diagrama UML de classes da URGE.

- O módulo de Objects deixou de existir, dando origem a um novo módulo: *Templates*.

- A Classe básica, *Object*, passou a ficar dentro do módulo de Environment, que por sua parte, passou a ser reconhecido como container de todos os elementos de uma cena de um jogo.
- *Templates* foi criado, esse módulo engloba uma série de entidades pré-fabricadas a partir da Object com o intuito de simplificar o desenvolvimento de projetos por parte dos usuários.
- Classe *Teleporter* foi adicionada para garantir a inter-conexão entre cenários de um jogo.
- Threading mudou para System, pois agora além de multi-threading, esse módulo possui suporte a rotinas de acesso ao sistema de arquivos.
- Connection foi re-estruturada para poder suportar conexões TCP, UDP e multicasting.

A última versão do diagrama UML de classes é completa e engloba de modo organizado todas as funcionalidades desejadas na versão final da URGE. Porém, não é intuitiva e nem simples, para um usuário comum, por possuir muitas classes que são importantes apenas em etapas avançadas de um projeto. Para facilitar a utilização pelos usuários menos experientes, foi feita uma versão simplificada do último diagrama, que inclui apenas as classes mais comuns.
Programadores iniciantes no uso da URGE não ficariam perdidos, e usuários avançados poderiam acompanhar pela UML original, completa.

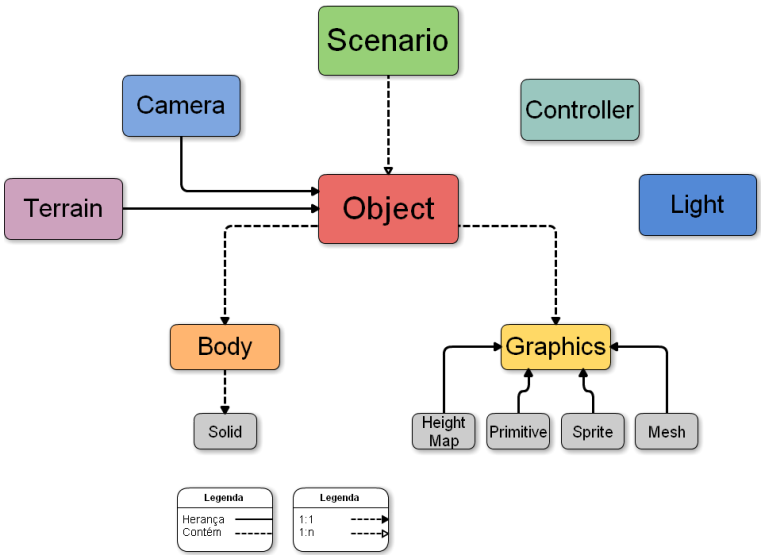

Figura 65: UML simplificado.

5.3 Marcos intermediários do projeto

Ao longo de 2011 vários eventos foram promovidos, tanto para divulgar a URGE quanto para testar sua aceitação pelo público. Jogos exemplo foram criados, um mini-curso ocorreu durante o evento SETI 2011, houve uma apresentação na Jornada de Iniciação Científica, além de outras interações que serão descritas a seguir.

5.3.1 Projetos parciais

Três jogos de demonstração foram feitos com a engine como exemplo do que era possível fazer até então. O primeiro exemplo, chamado "URGE Carnival", foi criado em março de 2011, como visto no semanário (Apêndice A). É um jogo simples, no qual o jogador deve atirar em patos usando bolas. Neste jogo foram testados: colisão de esfera com sprite, renderização de primitivas, texturização, height map, bump mapping e áudio.

Figura 66: URGE Carnival.

O segundo jogo exemplo, um pouco menos simples, tem como objetivo testar a física da URGE. Conta com um modelo 3D de carro, que anda livremente por um plano infinito. Neste jogo foi testada toda a física e o carregamento de modelos 3D.

Figura 67: Jogo exemplo do Carro.

O terceiro exemplo foi inspirado no jogo Angry Birds. O jogador mira livremente pelo cenário, e arremessa modelos 3D de patos. A intenção era testar a colisão entre objetos, que apresentava problemas nesta etapa do projeto.

5.3.2 Kano

Aproximadamente 45% dos recursos de software raramente são usados, enquanto apenas 20% são usadas com frequência [Joh02]. Ou seja, o desenvolvedor teria muito menos trabalho se fizesse apenas o que realmente interessa para o cliente/usuário, e poderia cobrar o mesmo preço. Um modo de descobrir o que mais interessa para o cliente/usuário é perguntar para eles mesmos o quanto seria usado de cada recurso. Uma ferramenta para ajudar nesta pesquisa é o modelo de Kano de satisfação do cliente [Coh06].
Cada recurso é dividido em três categorias: Mandatory, Linear e Excitement.

- Mandatory: Recurso essencial, obrigatório.
- Linear: Quanto mais tiver deste recurso, melhor.
- Excitement: O cliente não vai notar se não existir, mas sua presença chama a atenção.

Foram listados alguns recursos da engine, e sete responderam um questionário sobre o que achavam da presença destes recursos. As sete pessoas foram:

- 4 programadores da engine (Alexandre, Jefferson, Matheus e Felipe)
- Um graduando em Gestão Ambiental
- Um recém graduado em História
- Um graduando em Marketing

Os três entrevistados de fora da equipe são potenciais usuários da engine quando concluída, têm contato com jogos e sabem do que se trata o projeto.
Foram listadas 30 funcionalidades arbitrárias da engine, envolvendo apenas iluminação, primitivas e modelos. Cada uma das sete pessoas deveria responder, para cada uma das 30 funcionalidades, às duas seguintes perguntas:

1. Como você se sentiria se esta funcionalidade existisse?
2. Como você se sentiria se esta funcionalidade não existisse?

E as opções de resposta eram obrigatoriamente uma destas:
A – É esperado que seja assim
B – Iria gostar se fosse assim
C – Tanto faz (Neutro)
D – Consigo conviver com isso
E – Não gosto disso deste modo

Exemplo de resposta para o recurso "Desenhar Planos"
1 – A – "Espero que a engine desenhe planos"
2 – E – "Não iria gostar se a engine não desenhasse planos"

Os sete entrevistados sabiam como responder às perguntas, mas não sabiam como elas seriam interpretas. A interpretação segue a regra da tabela 68:

Por exemplo, para a resposta A e E, o entrevistado sem saber está dizendo que tal recurso é Mandatory.
M – Mandatory
L – Linear
E – Excitement

Customer Requirements →		Dysfunctional Question: Like	Expect	Neutral	Live with	Dislike
Functional Question	Like	Q	E	E	E	L
	Expect	R	I	I	I	M
	Neutral	R	I	I	I	M
	Live with	R	I	I	I	M
	Dislike	R	R	R	R	Q

M Must-have
L Linear
E Exciter
R Reverse
Q Questionable
I Indifferent

Figura 68: Tabela de interpretação do Kano.

R – Reverse (Funcionalidade incorreta e que precisa sair)
I – Indiferente
Q – Resposta sem sentido do entrevistado.

Na tabela 69 estão mapeadas as respostas de todos os recursos e a apuração das respostas dos sete entrevistados.

5.4 O que não deu certo

Ao longo de tudo o processo de desenvolvimento da URGE houve vários artifícios criados parcialmente porém abandonados, por questões de priorização, custo de trabalho ou até custo para atualização de futuras versões.
Nesta Seção, listaremos alguns desses recursos que foram descartados devido a mudanças na projeção da engine.

5.4.1 Câmera com ajuste automático

Durante o desenvolvimento do segundo jogo exemplo, do carro (como visto anteriormente), foi desejado que a URGE tivesse uma câmera inteligente em terceira pessoa. Essa câmera deve evitar "entrar" em superfícies físicas e deve se dirigir à mais provável direção que o jogador deseja ver.

Tema	E	L	M	I	R	Q	Categoria
Renderizar plano	0	14	86	0	0	0	Obrigatório
Renderizar caixa	0	14	72	14	0	0	Obrigatório
Renderizar esfera	0	14	86	0	0	0	Obrigatório
Renderizar cone	0	14	72	14	0	0	Obrigatório
Renderizar tórus	14	29	14	43	0	0	Linear
Renderizar paraboloide hiperbólico	14	14	0	72	0	0	Indiferente
Carregar tipo OBJ	0	0	86	14	0	0	Obrigatório
Carregar outros formatos	14	0	72	14	0	0	Obrigatório
Texturizar primitivas	0	0	86	14	0	0	Obrigatório
Repetição de textura (gl_repeat)	43	0	43	14	0	0	Extra/Obrigatório
Primitivas com cores ou gradientes	14	0	43	43	0	0	Obrigatório/Indiferente
Sem iluminação	14	0	29	57	0	0	Indiferente
Reflexão difusa	0	0	86	14	0	0	Obrigatório
Reflexão especular	14	0	57	29	0	0	Obrigatório
Sem Shadow Mapping	14	0	43	29	14	0	Obrigatório
Sombra em outros objetos	14	58	14	14	0	0	Linear
Sombra em si mesmo	43	14	29	14	0	0	Extra
Bump Mapping	14	0	57	29	0	0	Obrigatório
Parallax Mapping	43	14	14	29	0	0	Extra
Sphere Mapping	72	0	14	14	0	0	Extra
Qualidade mínima	29	0	43	14	14	0	Obrigatório
Qualidades intermediárias	14	0	72	14	0	0	Obrigatório
Qualidade alta	29	14	43	14	0	0	Obrigatório
Sem física	14	0	43	14	29	0	Obrigatório
Com gravidade apenas	14	0	57	0	29	0	Obrigatório
Física real	0	0	86	14	0	0	Obrigatório
Mipmap	0	14	29	57	0	0	Indiferente
Ler primitiva de arquivo	0	0	43	57	0	0	Indiferente
Usar mouse para selecionar objeto	14	29	57	0	0	0	Obrigatório
Controle por teclado	29	0	43	14	14	0	Obrigatório

Figura 69: Resultado da pesquisa com o método Kano.

Entretanto, após pesquisa do Kano, a equipe desistiu da idéia, pois decidiu-se que isso não era prioridade. A implementação é possível, mas requer muito esforço para algo que não é foco da engine.

A idéia constituia-se em calcular a orientação e a posição da câmera usando coordenadas esféricas. A velocidade em relação ao alvo deveria ser calculada usando uma fórmula derivada de uma equação física de extensão da mola.

O modelo de câmera física tem problemas com oclusão. Se um objeto ficar entre o alvo e a câmera, deve-se calcular qual a esfera mais próxima de mesmo tamanho da câmera que não possui objetos no caminho. É preciso implementar colisão com raio. Uma saída para o algoritmo da melhor posição mais próxima é tornar o cenário entre a câmera e o alvo transparente. Mas isso pode tirar a sensação de realidade, em alguns casos.

5.4.2 Editor de cena

O editor de cena será uma ferramenta poderosa, que permitirá a artistas gráficos montar um cenário 3D sem a necessidade de saber programar. Embora esteja nesta sessão, o editor de cena é uma ferramenta ainda presente na organização do projeto, e de grande importância. Mas sua criação no início do desenvolvimento da URGE não foi uma boa experiência. Como os nomes, tipos de implementação e classes sofreram modificações

consideráveis ao longo do desenvolvimento da engine, constantemente o editor de cena também precisava ser modificado. Por fim, a equipe decidiu que o editor deve ser feito apenas quando a URGE sair da fase Beta.

5.5 Conclusão sobre o Processo de Criação

O desenvolvimento da URGE, à parte da temática de programação de jogos, forneceu ao grupo conhecimento sobre metodologia ágil e gerência de projetos. Abordamos conceitos como o Planning Poker para a definição das prioridades, e experimentamos abordagens ao público, como o Kano.

Além disso, abordamos detalhes das diversas versões da modelagem UML, desde o período em que a equipe não era orientada por um professor até o momento atual. Vimos que ao longo do tempo, o projeto tendeu à simplicidade de entendimento e de programação.

Por fim, comentamos sobre algumas idéias abandonadas, como a implementação de uma câmera em terceira pessoa com ajuste automático. A organização do projeto e das tarefas permitiu que testássemos a possibilidade e abandonássemos caso necessário, sem alterar o fluxo do trabalho.

6 A Estrutura da URGE

A organização da engine é o ponto chave para atingirmos uma simplicidade de uso por parte de usuários, e eficiência na implementação de futuros recursos por parte de seus desenvolvedores. Tendo isso como um dos principais objetivos, a URGE foi estruturada de forma a possibilitar o desenvolvimento de jogos com o mínimo de esforço no aprendizado de seu manuseio, e também por parte de usuários com baixa experiência em programação.

Todavia, repare que a simplicidade de manipulação de uma engine tende a obrigar que seus módulos sejam pré-estipulados a atingir um objetivo mais específico. Por exemplo, a URGE possibilita a geração automática de oceanos de alta qualidade gráfica carregando apenas duas texturas, isso é um recurso simples de se usar, porém bastante específico.

Por isso, a ferramenta também deve fornecer módulos de programação que permitam liberdade e assistência na criação de recursos mais avançados ou específicos. Mas, infelizmente, a manipulação de tais recursos vai exigir um nível de experiência em programação um pouco mais elevado. Isso é um paradigma quase impossível de se quebrar.

Nesse capítulo, veremos uma alternativa feita a partir da estrutura e organização da URGE que possibilita desenvolver recursos mais específicos sem, praticamente, nenhum esforço e por outro lado fornece uma grande liberdade na implementação de novos módulos para desenvolvimento de um jogo de grande porte.

6.1 O Modelo Estrutural

Como visto no capítulo anterior, o diagrama UML de classes da URGE explicita e mapeia todas as suas entidades e relações. Porém, com o intuito de simplificar e enfatizar o funcionamento da engine, vamos introduzir o diagrama de componentes, que constitui-se numa representação reduzida das camadas operacionais facilitando o entendimento por parte do usuário da URGE. Eis o diagrama de componentes:

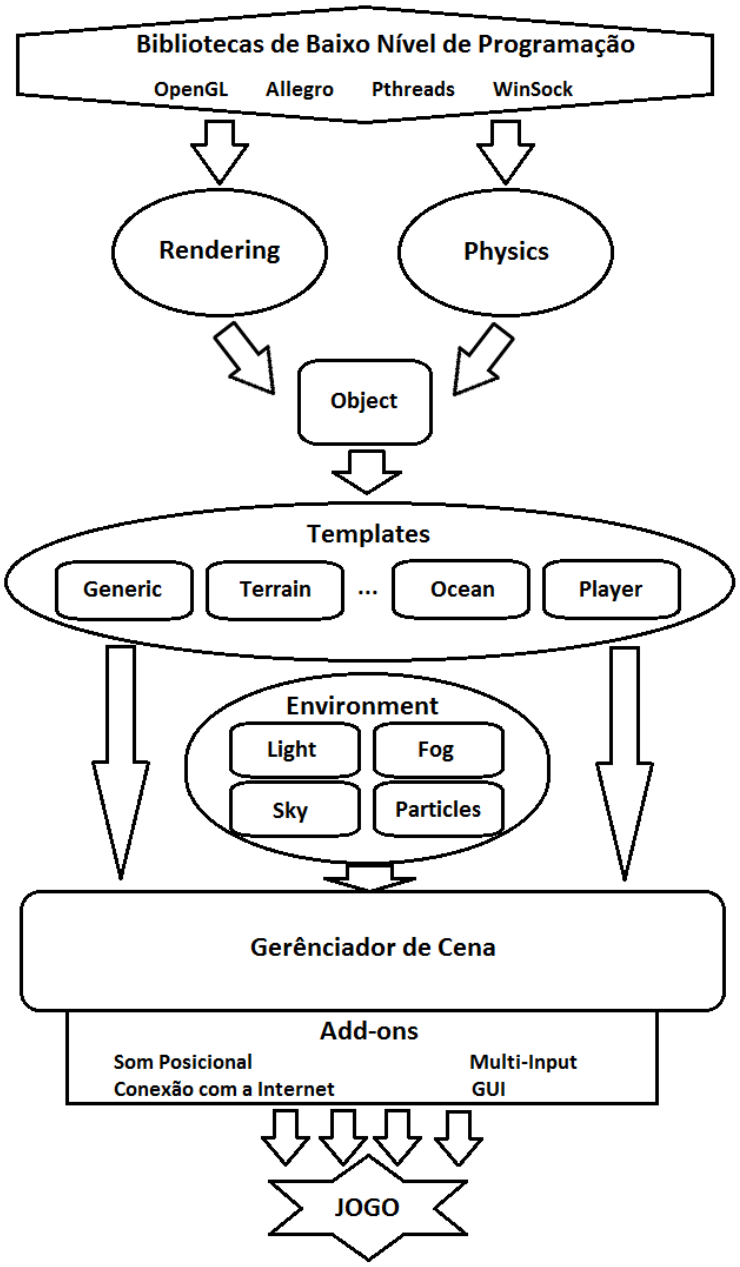

Figura 70: Diagrama de componentes da URGE.

Repare que as todas as transições ocorrem de cima para baixo, pois o objetivo dessa representação é enfatizar todo o processo de geração de um jogo a partir da URGE. Evidentemente, estruturas mais próximas do topo constituem níveis mais baixos de implementação. Por outro lado, estruturas mais próximas ao produto final (o jogo), representam partes de alto nível de programação pelas quais já podem ser manipuláveis por usuários da URGE.

6.1.1 Bibliotecas de Baixo Nível de Programação

Conforme dito anteriormente, o topo do modelo exibe o nível mais baixo de programação, ele é constituído por bibliotecas de código usadas na implementação da engine. A linguagem de programação selecionada para a implementação da URGE é C++, pelo fato de ser extremamente eficiente e possibilitando o funcionamento de recursos computacionalmente caros em máquinas de baixo desempenho de processamento. As bibliotecas acopladas à linguagem são OpenGL, uma versão adaptada da Allegro, PThread e WinSock.

O OpenGL é responsável por todo o sistema visual da engine, isso implica que tudo que está no módulo de *Rendering* é processado para a tela graças a essa biblioteca.

A biblioteca Allegro foi modificada pelos programadores da URGE com o intuito de estender suas funcionalidades para atender certos requisitos da engine. Ela é responsável pela captura de entrada do computador, pelo módulo de som da URGE e pelo carregamento de imagens.

A Pthread foi a biblioteca escolhida para manipular *threads*, possibilitando a programação concorrente.

A WinSock possibilita a conexão com a internet através de *Sockets* TCP ou UDP, e possui um módulo de extensão para conexões *Multicast*.

Todas as bibliotecas envolvidas na implementação formam a base da URGE, e compõem a chave para o funcionamento das partes de nível mais elevado de programação.

6.1.2 Rendering e Physics - Os dois elos da URGE

Apesar da alta complexidade das relações e entidades vistas no diagrama UML de classes, o "coração" da URGE pode ser resumido em apenas dois importantes elos: O Núcleo de Visualização (*Rendering*) e o Sistema de Simulação Física (*Physics*). Ambos os núcleos foram explicitados e analisados em detalhes nos Capítulos 2 e 3.

Vimos que o núcleo de visualização engloba todas as classes e métodos responsáveis por exibir objetos na tela, com o maximo de proveito obtido na relação entre qualidade gráfica e performance, considerando os limites da capacidade de processamento por parte do computador em questão. Na prática, esse sistema é responsável desde tarefas como desenho de primitivas até a renderização de terrenos, mapeamentos e iluminação via shaders e efeitos visuais como simulação de água e fogo.

O núcleo de simulação física engloba toda a movimentação, detecção e resposta de colisão e interação física entre corpos de objetos. Esse sistema sempre deve preservar as leis da física do mundo real, apesar de disponibilizar a customização de qualquer parâmetro como elasticidade, massa e até gravidade.

6.1.3 Object - A Entidade Elementar

Existe uma entidade básica que pode, e deve, ser "pai" de qualquer elemento dentro do espaço de uma cena. Tal entidade é chamada de *Object*, e a sua grande importância, reside no fato de unificar os dois núcleos principais da engine (rendering e physics) através de uma releção de dependência.

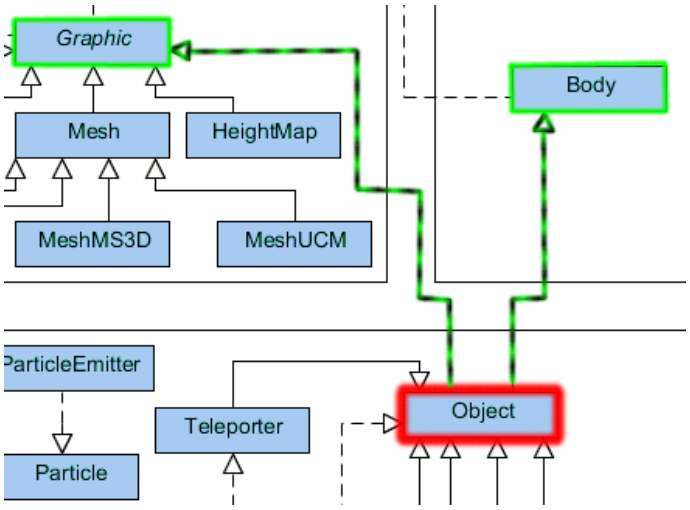

Figura 71: Classe Object representada no Diagrama UML de Classes da URGE

Mesmo assim, repare que tal elemento não é, literalmente, superclasse de todas as classes subsequentes, como o conceito de classe Object sugere, mas apenas de classes que representam objetos físicos e/ou visuais. Um exemplo de classe que não herda de Object é a *Controller*, responsável pela captura e tratamento dos comandos de entrada.
Das classes filhas de Object, três são suficientes para que um usuário iniciante seja capaz de criar um jogo simples. Elas são: Camera, Generic e Terrain.

6.1.4 Templates - Recursos de alta complexidade e fácil acessibilidade

Usando a entidade Object, um desenvolvedor já é capaz de criar uma cena de um jogo sem dificuldades. Todavia, faz parte do objetivo da URGE, simplificar ao maximo

o trabalho necessário para a criação de um jogo. Sendo assim, foi criado um módulo de suporte de elementos padrões em jogos, chamado *Templates*. Tal módulo reúne classes de alto nível e fácil acessibilidade, já pré-fabricadas e prontas a serem instanciadas e usadas diretamente em um jogo. Para exemplificar esse recurso iremos discutir brevemente algumas entidades contidas no módulo Templates.

Há uma classe chamada *Terrain* a qual é capaz de gerar um terreno a partir de um height map (discutido no Capítulo 2), que por sua parte, é definido através de uma imagem carregada de um arquivo. Alguns outros parâmetros editáveis do terreno são as três dimensões e a textura difusa.

Existe, também, uma entidade nomeada *Ocean* a qual baseia-se em apenas dois mapas de normais, carregados do arquivo, para gerar um ambiente semelhante ao oceano, com ondulações, reflexões e refrações como na natureza. Todos estes parâmetros são também editáveis, assim como a região que será ocupada pela água.

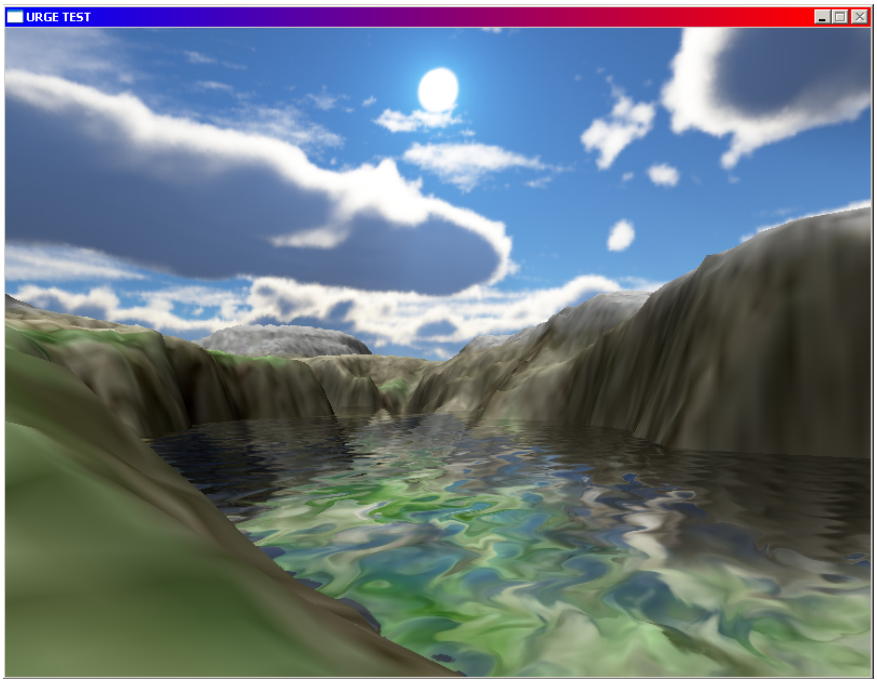

Figura 72: Exemplo das Entidades Terrain e Ocean em ação

Há outra classe chamada *Generic*, como seu próprio nome sugere, ela é capaz de se transformar em diferentes objetos de cena. Dependendo de como o usuário a edite, objetos desta classe podem se tornar desde primitivas geométricas, modelos 3D animados ou estáticos e até imagens 2D em um mundo 3D. Todos com possibilidade serem sujeitos a de simulação física.

Observando tais exemplos, podemos concluir que o módulo Templates é bastante útil para programadores iniciantes, pois fornece recursos de alta qualidade sem exigir algum conhecimento prévio das técnicas de programação que os compõem.

6.1.5 Environment - Criador de Ambientes e Efeitos

Todos os objetos visíveis e/ou físicos de um Jogo já podem ser encontrados ou criados a partir dos Templates ou da classe Object. Entretanto, há um conjunto de elementos que não são visuais nem tão pouco físicos, porém são de suma importância no processo de criação de qualquer jogo.

O módulo de *Environment* reúne todas essas entidades que também são administradas pelo gerênciador de cena. Luz, Neblina, Partículas e Céu são exemplos de seus componentes.

A classe *Light*, na qual faz o papel da luz de um cenário é essencial para garantir a imersão e o realismo em um jogo. A URGE suporta infinitas luzes dentro de uma mesma cena, o responsável por isso é o gerenciador de cena que seleciona as luzes de maior influência sobre o observador para ser aplicada em um respectivo quadro.

A classe *Particle Emitter*, discutida na Seção 2.7, é responsável por vários efeitos visuais, tais como Fogo, fumaça e rastros luminosos. Seu papel é fundamental para o enriquecimento de detalhes do jogo.

6.1.6 Gerênciador de Cena - O agente de Integração da URGE

Segundo o Capítulo 4, é possível criar um jogo apenas juntando os dois grandes núcleos da engine (Rendering e Physics), o que já é feito pela classe Object conforme visto anteriormente. Todavia, se somente juntássemos ambos os núcleos, teríamos um sério problema de ineficiência causado por uma série de overheads devido à falta de um gerenciamento na interação de todas as entidades do jogo.

Com tal objetivo em mente, foi criado uma entidade de gerenciamento de cena a qual realiza uma série de algoritmos (detalhados no Capítulo 4) para evitar desenhar objetos não vistos pelo jogador, e evitar o processamento necessário para efetuar a interação física entre objetos que evidentemente não irão se colidir.

6.1.7 Add-ons - Recursos extras

Apesar do foco da URGE ser fornecer todos os recursos necessários para criar gráficos realistas e uma física concisa a fim de facilitar ao máximo o desenvolvimento de um jogo 3D, preservando a eficiência; A engine possui um sistema de recursos adicionais para potencializar a liberdade do usuário. Chamamos esses de *Add-ons*. Atualmente a URGE conta com os seguintes recursos extras: *multi-input*, som posicional, G.U.I. e conexão com a internet.

O Multi-Input é um sistema de interfaceamento entre o programador usuário da URGE e o acesso a todos os comandos de entrada de um computador.
Seu objetivo é simplificar ao máximo a manipulação do *input* do jogo. Para isso, há uma entidade chamada *Controller* a qual atua como um gerenciador de comandos de entrada. Ela associa cada comando a um rótulo, sendo que o usuário se encabe, apenas, de manipular tal rótulo. Por exemplo, suponha que o usuário queira criar uma ação controlada pelo jogador, algo como pular ou atacar, ele deverá criar um rótulo (um nome para essa ação) e selecionar qual deverá ser o comando de entrada associado, a URGE faz todo o gerenciamento desse procedimento. Dessa forma, não há praticamente nenhum esforço no caso do usuário ou do próprio jogador querer alterar o input que realiza a respectiva ação.

Efeitos sonoros potencializam a realidade de um jogo, da mesma forma que Músicas o tornam muito mais imersivo.
No caso dos efeitos sonoros, a URGE conta com um dispositivo de som 3D (som posicional) a qual ajusta o volume e o balanço do som em função da posição do mesmo.
Além disso, a URGE é capaz de carregar arquivos *.mid, *.wav ou *.ogg como música de um cenário de um jogo.

Embora ainda não finalizada, é desejado que na versão final da URGE seja possível criar interfaces com usuário, envolvendo textos, botões, menus, caixas de opção e de texto, *comboboxes* e similares. Mesmo assim, a engine possui um sistema básico contando com impressão de texto na tela do computador (através de uma rotina bastante semelhante ao *printf(...)* do C) e botões interativos.

Também é possível, com a URGE, criar jogos na qual vários jogadores se interagem por meio de uma conexão com a local ou com a Internet.
Apesar de tais recursos funcionarem através de entidades de relativamente baixo nível de programação, a URGE fornece todo o suporte necessário para conexões TCP ou UDP via

sockets. Além de recursos mais elaborados como *Multi-Casting* que facilitam a troca de mensagens entre vários jogadores na internet.

6.1.8 O Jogo - Produto Final

Todas essas funcionalidades e classes juntas, por fim, dão origem ao jogo. Este fluxo leva em consideração apenas a parte de programação, claramente, também é necessário que se tenha recursos artísticos como modelos 3D, texturas e imagens, além de um roteiro para que um jogo seja realmente interessante e robusto.

6.2 O Comportamento em diversos Computadores

Um importante tópico de discussão é a questão da relação performance X qualidade, ou seja, não é possível maximizar a qualidade gráfica sem prejudicar a performance e vice-versa.

Como solução, a URGE fornece um recurso de ajuste de qualidade em função da relação

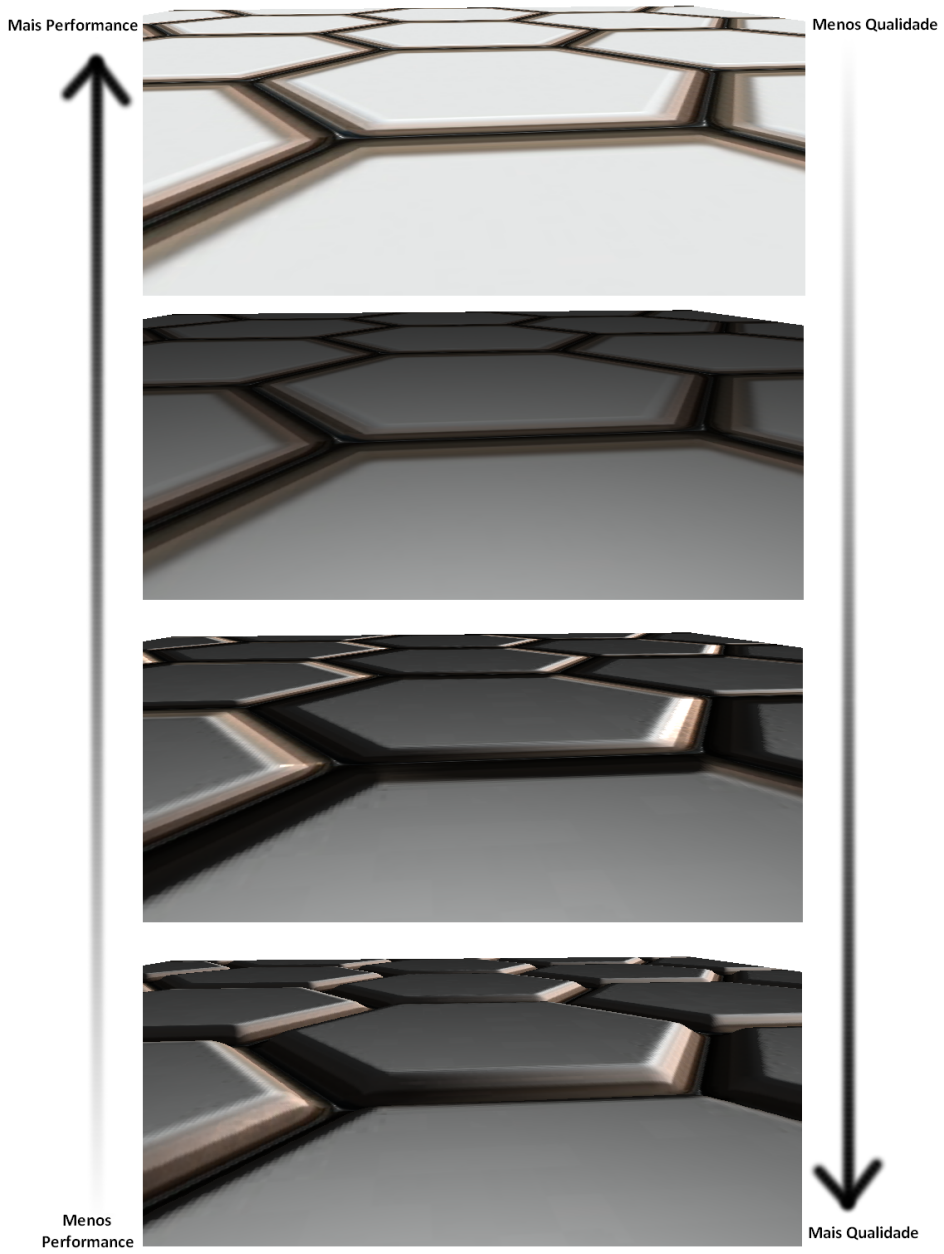

Figura 73: Balança entre qualidade e performance

performance X qualidade, isto é, quanto menor a qualidade maior será a performance, possibilitando seu funcionamento em máquinas de considerável baixo desempenho. A tabela abaixo possibilita a análise de cada opção em função dos recursos da engine:

Opção de Qualidade	Correção de Perspectiva em Texturas	Anti-Aliasing	Alta de Precisão no filtro de Mip-Mapping	Possibilitar Shaders	Alta Precisão no Fragment Shader	Método de Iluminação
Quality Zero	-	-	-	-	-	Flat
Quality Poor	-	-	-	-	-	Gouraud
Quality Low	-	Parcial	-	Sim	-	Gouraud
Quality Average	Sim	Parcial	-	Sim	-	Gouraud
Quality High	Sim	Parcial	Sim	Sim	-	Blinn-Phong
Quality Top	Sim	Total	Sim	Sim	Sim	Phong
Quality Perfect	Sim	Total	Sim	Sim	Sim	Phong

Pela tabela não é possível ver nenhuma diferença entre a opção de maior e a opção de segunda maior qualidade, pois, com o intuito de simplificar, há uma série de outros parâmetros de performance X qualidade omitidos pela tabela. Alguns desses parâmetros são: Quantidade de vértices no cálculo de primitivas, forçar o desligamento de mapeamentos avançados (como bump mapping e parallax mapping), LOD base do terreno, entre outros...

A seguinte tabela coloca o nível de qualidade da engine em função da configuração mínima recomendável de um computador:

Opção de Qualidade	Quantidade de Núcleos da CPU	Necessidade de GPU	Versão do OpenGL	Clock Mínimo de Processamento Gráfico	Quantidade de Núcleos da GPU
Quality Zero	1	-	1.5	-	-
Quality Poor	1	-	1.5	-	-
Quality Low	2	-	2.0	-	-
Quality Average	2	*	2.0	300*	8*
Quality High	2	Sim	2.1	400	16
Quality Top	4	Sim	3.0	450	32
Quality Perfect	4	Sim	3.2	500	64

Na opção de performance X qualidade média (*Quality Average*), não é necessário possuir uma placa gráfica para atingir a configuração mínima recomendável, mas caso o jogo criado possua uma grande quantidade de recursos em uma só cena, é recomendável que o computador seja dotado de um processador gráfico.

6.3 Conclusão sobre a Estrutura da URGE

Portanto, a estrutura da URGE foi projetada para respeitar duas importantes premissas: A facilidade de uso por parte de usuários e eficiência na implementação de futuros recursos, eliminando a necessidade de grandes alterações na engine, por parte de seus desenvolvedores.

Vimos que o modelo estrutural da URGE foi modelado de forma a fornecer liberdade e simplicidade na programação, paradigma que é difícil de atingir. Mesmo assim, a ferramenta apresenta uma conjectura baseada na unificação dos núcleos de visualização e simulação física em uma entidade elementar (a classe Object). Todas as posteriores entidades de um jogo podem se derivar dessa entidade, disponibilizando uma base bastante sólida e permitindo uma grande liberdade para usuários da engine.
Além disso, há um módulo de elementos pré-fabricados que simplificam o desenvolvimento de um jogo a um ponto na qual o usuário não precisa mais criar novas classes, mas apenas instância-las.

Como último ponto, foi destacado o comportamento da URGE em computadores de alto e baixo desempenho. Pode-se observar que a URGE se flexibiliza de forma a respeitar as limitações na capacidade de processamento de máquinas pouco potentes. A chave para essa flexibilidade reside no fato de se encontrar um equilíbrio na relação performance X qualidade, ou seja, agrupar todos os recursos computacionais de uma engine, afim de otimiza-los e simplificar-los em um simples conjunto de opções.

7 Experimentação e Conclusão

Como desfecho do trabalho, será apresentado, neste capítulo, todas as experiências relacionadas ao público que a URGE foi submetida, o feedback por parte de usuários da ferramenta e *workshops* as quais a engine foi o usada como ferramenta para desenvolvimento de jogos 3D. Por último, realizaremos uma revisão geral do trabalho e discutiremos sobre os seus futuros objetivos.

7.1 Experimentação

7.1.1 Trabalho na cadeira computação gráfica

No período 2011-2, a disciplina Computação Gráfica I do curso de Ciência da Computação da UFRJ, então ministrada pelo professor Rodrigo de Toledo, teve como objetivo de um dos trabalhos o desenvolvimento de um jogo simples com a URGE. Foram entregues 27 jogos.
Os objetivos do trabalho foram:

- Criar chão com *height map*
- Criar céu com *sphere mapping*
- Criar inimigos pelo cenário.
- Definir uma condições de vitória e derrota.

Estas foram as únicas especificações, o que permitiu aos alunos criar seus jogos com muita liberdade, tanto de complexidade quanto de tema. O enunciado completo pode ser visto no apêndice.
Como auxílio, foram organizadas aulas-dojo [GB]. Num dojo, alunos devem desenvolver um exemplo simples escolhido pelo professor, sem que estes necessariamente tenham

noção da tecnologia utilizada. Dois alunos de cada vez devem ir ao quadro e trabalhar juntos por 5 minutos, de modo que ao longo da aula, um certo problema seja resolvido. O problema em questão foi algo simples que envolvesse muitos tópicos da URGE: Criar um cenário com chão, céu e com um modelo 3D estático. O dojo serviu para que os alunos se sentissem com experiência suficiente para iniciar seus próprios trabalhos.
Ao longo de duas semanas, a equipe da URGE se disponibilizou para ajudar os alunos. A cada monitoria, os alunos reportavam suas opiniões parciais e os *bugs* encontrados. Nesses poucos dias, graças ao auxílio dos alunos, muitos bugs foram removidos e algumas funcionalidades mudaram de nome ou de lugar dentro da modelagem.
A correção deste trabalho foi bastante leve. A intenção da equipe foi experimentar o uso da URGE com usuários reais, e nisso foi bem-sucedida. Todos os grupos fizeram o mínimo exigido, com um mínimo de bugs, e boa parte dos grupos criaram funcionalidades extras, como a renderização de água com a classe Ocean. O grupo de menor nota foi avaliado com 7.2.

Figura 74: Jogo desenvolvido por alunos da disciplina Computação Gráfica

Através dessa experiência pudemos descobrir pontos positivos e negativos da URGE até então. A equipe criou um formulário, que foi respondido por dez alunos. Nele estavam as seguintes perguntas:

1. O que você achou da simplicidade de Programação na URGE como Engine (Motor) para Criação de Jogos 3D?
    - Muito simples, totalmente intuitiva: 3
    - Simples no Geral, mas com algumas dificuldades: 5

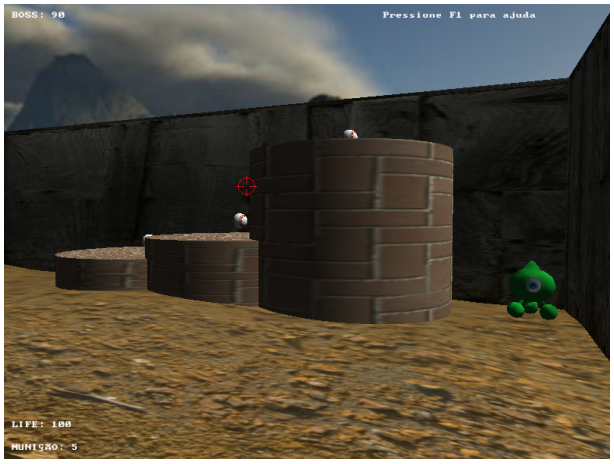

Figura 75: Jogo desenvolvido por alunos da disciplina Computação Gráfica

Figura 76: Jogo desenvolvido por alunos da disciplina Computação Gráfica

- Mais ou Menos, alguns pontos foram fáceis de entender mas outros: 2
- Um pouco complicada, o código está um pouco confuso: 0
- Totalmente complicada, não há nenhuma coerência: 0

2. O que você achou da capacidade da URGE como engine (Motor) para criação do seu projeto?

   - Totalmente Completa, não há nada que falta para criar esse tipo de Jogo: 2
   - Razoavelmente completa, porém faltam alguns detalhes em certos pontos da URGE: 4
   - Mais ou Menos, possui recursos importantes mas faltam outros também de certa relevância: 4
   - Relativamente Incompleta, possui alguns recursos mas faltam muitos outros: 0

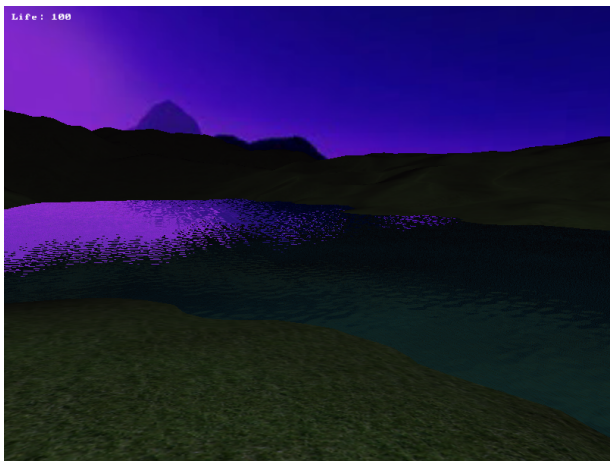

Figura 77: Jogo desenvolvido por alunos da disciplina Computação Gráfica

- Totalmente Incompleta, não tem nada do que eu esperava para esse tipo de Jogo: 0

3. Quanto a integridade, como você Considera a URGE?
   - Nunca deu problemas comigo até agora: 4
   - Algumas raras situações dão bug: 3
   - Bugs acontecem esporadicamente: 3
   - Bugs acontecem normalmente: 0
   - Sempre dá Bug: 0
4. Caso você respondeu que houve bugs, diga o nível de relevância desse(s) bug(s).
   - Os bugs eram detalhes totalmente dispensáveis, nada importante: 1
   - Os bugs eram em geral sem importância, podendo programar sem preocupações: 4
   - Só alguns bugs eram críticos, pois podem prejudicar o desenvolvimento de meu Jogo: 2
   - Muitos bugs que representam problemas para o desenvolvimento de um Jogo: 0

Os gráficos 78, 79, 80 e 81 são referentes a cada uma das quatro perguntas do formulário.

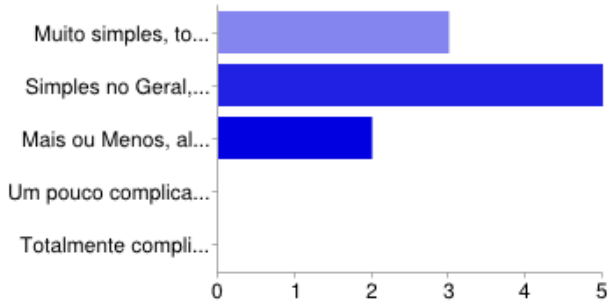

Figura 78: Gráfico referente ao feedback dos alunos de CG.

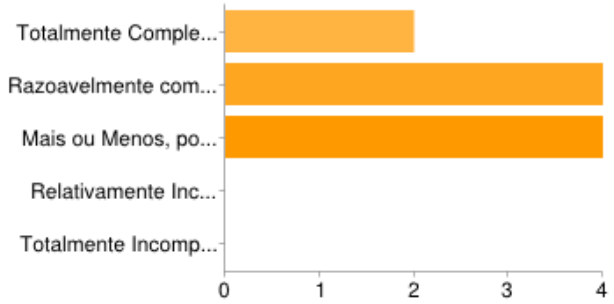

Figura 79: Gráfico referente ao feedback dos alunos de CG.

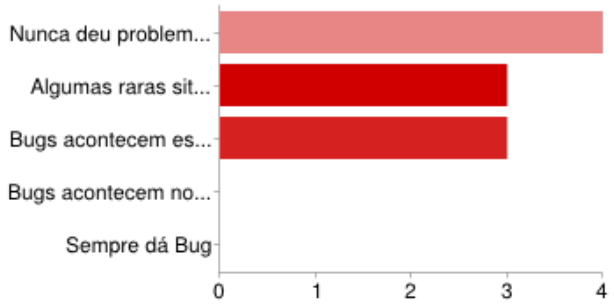

Figura 80: Gráfico referente ao feedback dos alunos de CG.

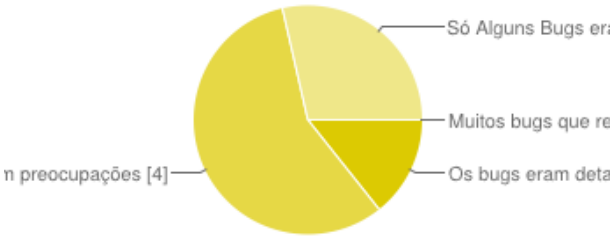

Figura 81: Gráfico referente ao feedback dos alunos de CG.

7.1.2 Apresentações e minicursos

A URGE foi apresentada para o público em dois eventos da UFRJ.

XXXIII Jornada de Iniciação Científica: Visão geral, modelo estrutural, capacidade e

exemplos foram mostrados para professores e alunos.
SETI 2011: Um workshop de quatro horas foi ministrado, no qual foi ensinado como criar um jogo com a URGE.
Outro Workshop está sendo organizado para o segundo semestre de 2012.

7.2 Conclusão Geral

Vimos que engines, conhecidos no Brasil como motores de jogos, hoje em dia, são considerados essenciais para desenvolver jogos de alto nível de qualidade, eliminando tempo e esforço desnecessário. Elas atuam como uma interface entre o desenvolvedor de jogos e recursos de baixo nível de programação e de alta complexidade de implementação.

O trabalho aqui exposto, apresentou uma nova engine integralmente projetada e desenvolvida por alunos do curso de Ciência da Computação da UFRJ, a URGE. Ela busca atingir dois simples objetivos fundamentais: completude e simplicidade.
A completude traduz-se em seus diversos módulos, apresentados ao longo desse trabalho, que visam reduzir ao maximo o esforço para se criar um jogo dentro do foco da engine. Apesar do foco ser jogos 3D com cenários a céu aberto, evidentemente, é possível criar, sem dificuldades, vários outros tipos de jogos 3D.
Pode-se observar a simplicidade da URGE na forma intuitiva de se programar e na simples organização de sua modelagem. Além de ter sido classificada como uma ferramenta de fácil manipulação por usuários que participaram da pesquisa apresentada na Seção 7.1.

No decorrer da primeira parte do trabalho, discutimos as técnicas de programação envolvidas no processo de criação da ferramenta. A URGE, com o intuito de preservar a simplicidade, foi projetada em função de dois grandes núcleos: Rendering e Physics, o núcleo de visualização e o núcleo de simulação física, respectivamente. O foco do núcleo de Visualização é simular de forma fiel à realidade materiais visíveis, preservando a performance, principalmente se a ferramenta precisar trabalhar em computadores de baixo desempenho. O núcleo de Física, busca simular, em tempo real, a interação física entre entidades de um jogo, sempre preservando as leis físicas do mundo real.
Ambos os núcleos são integrados numa entidade básica, chamada Object, a qual representa qualquer elemento no cenário de um jogo. Isso significa que, todas as entidades de um jogo devem ser ou herdar Object.
Dessa forma a URGE pode gerenciar a cena de um jogo para eliminar qualquer processamento gráfico ou físico desnecessário.

Na segunda parte, foi apresentada as metodologias de Engenharia de Software envolvidas no processo de criação da ferramenta, além dos marcos na evolução da projeção e desenvolvimento. Vimos, também em detalhes, a organização da URGE que foi idealizada com o propósito de garantir a simplicidade de uso, e eficiência na implementação de futuros recursos.

Entre diversos workshops e trabalhos de experimentação sobre a URGE, fomos capazes de concluir, segundo a opinião pública, que apesar de ainda precisar amadurecer suas funcionalidades, a ferramenta, como uma das primeiras engines brasileiras, introduz uma nova visão na forma como podemos desenvolver jogos.

7.3 Trabalhos Futuros

Apesar da do tamanho e complexidade da engine, ela ainda se encontra na versão beta 0.2. O desenvolvimento da URGE ainda continua em progresso pela GDP (Game Development Project), que representa o núcleo de pesquisa e desenvolvimento de jogos formado por alunos e professores do Departamento de Ciência da Computação da UFRJ. Isso significa que, novos alunos foram treinados e estão dando sequência a futuras versões da URGE.
Portanto, o objetivo da equipe atual é atingir a versão 1.0, e para isso, precisa-se concluir os seguintes tópicos:

- Concluir um módulo completo de GUI aplicada a jogos, isso envolve botões, *checkboxes* e *sliders*. Atendendo o suporte para criação de telas de Inicio de Jogo, *Game Over*, Menu de opções e status do jogo;
- Efetuar animação por esqueletos e posteriormente carregar modelos do padrão MS3D;
- Criar um sistema de renderização por multi-passes. A Engine, atualmente, possui um suporte primitivo para multi-passes, limitado a apenas dois passes. A consequência de tal limitação impede, por exemplo, o efeito de refração por parte de duas superfícies transparentes ao mesmo tempo;
- *Shadow Mapping + Random Percentage Filters*. Para criar efeitos de sombras de superficies em tempo real;
- Física da Dinâmica Rotacional. Para isso, é preciso criar um sistema de detecção de ponto de colisão para podermos aplicar conceitos como torque e momento angular;
- Possibilitar input via *Joypads*;

- Criar uma estrutura do tipo *container* de Cenários, e *teleporters* para a intercomunicação entre os mesmos;
- Aprimorar o algoritmo de otimização de processamento gráfico de terrenos. Caso duas seções adjacentes de um terreno possuam LODs diferentes, é possível evidenciar uma "rachadura" entre elas, devido a ausência da integração de faces entre seções vizinhas num terreno;
- Editor de Cena. O atual editor de cena foi descontinuado por conta da falta de integração entre ele e a engine, o que levou a uma diferença muito grande nas propriedades de vários elementos comuns entre ambos, conforme a engine foi se aperfeiçoando.
- Criar um efeito de Reflexão e Refração realista e em tempo real. O atuais efeitos são baseados em uma textura (Cube Map ou Sphere Map) do céu. O objetivo seria, re-renderizar a cena e converte-la em seis texturas para compor o cube map na qual o efeito tanto de reflexão quanto de refração deve ser aplicado, tornando assim o efeito muito mais realista;
- Por último, entretanto o mais importante, realizar uma série de testes tanto pelos próprios desenvolvedores quanto por novos e antigos usuários da engine. Para obter melhores retornos, deverá ser organizado novos Workshops, cursos e projetos envolvendo a URGE.

APÊNDICE A – Semanário

Durante os primeiros três meses de orientação, foi escrito um semanário, com o histórico de atividades da semana anterior. Entretanto, o semanário não continuou a ser feito nos meses seguintes.

Dia 15/3/2011

- Primeira reunião. Apenas definimos como seria o projeto final e quem participaria.

Dia 22/3/2011

- Levamos a primeira versão da Product Box, mas decidimos melhorar alguns pontos.

Dia 29/3/2011

- Levamos um jogo simples feito com a engine (Urge Carnival). Atualizamos a Product Box. Foi introduzido o conceito de User Stories.

Dia 5/4/2011

- Apresentamos um protótipo do editor de cenários. As User Stories e Personas levadas para esta reunião estavam incorretas, e por isso definimos um novo esquema. Foi introduzido o conceito homem/dias.
- A Product Box teve sua versão atualizada, mas esta não foi avaliada. Foi pedido um diário de reuniões (este).

Dia 12/4/2011

- Apresentação das Personas e Stories atualizadas, transformadas numa tabela em HTML.
- Das novas Personas e Stories, foram observados que as histórias descritas eram muito técnicas.
- O item "URGE" das Stories, por não ser uma Persona, deveria fazer parte de GDP.
- Deveria ser adicionado como história "Aprender, transmitir conhecimento por aulas."
- Apresentamos o editor de cenário, já lendo primitivas por XML e integrado com a engine.
- Rodrigo pediu para transformar o Diário de Reuniões num Semanário, em Bullets (este).
- Na reunião seguinte deveríamos propor uma nova idéia de jogo.

Dia 19/4/2011

- Foi proposto um jogo de corrida, que começaria a ser feito para a semana seguinte.

Dia 26/4/2011

- Levamos um protótipo incompleto do jogo de corrida, com camera defeituosa e uma esfera no lugar do carro
- O jogo de corrida e a valoração das prioridades foram marcadas com estrela como pendentes

Dia 3/5/2011

- Levamos um exemplo com um carro em terceira pessoa. O exemplo ficou bom, mas possuia alguns bugs.
- Rodrigo viu como foi feito o algoritmo do Bump Mapping, disse não ter gostado e pediu para pesquisar em teses e artigos.

Dia 10/5/2011

- A colisão com plano foi consertada, e o Strip foi remodelado para se repetir por parâmetro e ser feito com 2 triângulos.
- O carro caindo e o toggle estavam prontos, mas acabaram ficando como pendentes.

Dia 17/5/2011

- A física ainda está pendente. Matheus está pesquisando algo antes de programar.
- Em vez do exemplo do carro, ficou pendente fazer outro, com uma bolinha caindo e quicando.
- Os artigos sobre câmera foram lidos.

Dia 24/5/2011

- Foi mostrado o toggle entre os shaders.
- Decidimos usar o Scribtex.
- Mostramos a bolinha caindo e quicando, como foi pedido
- O artigo sobre câmera em terceira pessoa do Game Gems 4 foi explicado.

Dia 31/5/2011

- Mostrado scribtex e 1ª versão estruturada do projeto final.

Dia 7/6/2011

- Pouco andamento, física sendo melhorada.
- Introduzido o conceito de Kano.
- Pedido para estudar sobre Kano e escrever um ou dois parágrafos.

Dia 14/5/2011

- Feito um questionário de Kano, pesquisa feita com 7 pessoas.
- Pedido para montar mais um questionário.
- Física consertada. Foi mostrado um exemplo com 3 bolas se colidindo simultaneamente.
- O exemplo do carro não foi mostrado por causa de um defeito na octree.

APÊNDICE B – Instalando a URGE

O conteúdo abaixo foi retirado do *website* da URGE (http://gdp.dcc.ufrj.br/urge). Ele consiste em um simples tutorial de instalação da Engine em um computador dotado de Windows XP, Vista ou 7.

B.1 Componentes do DevPack

Faça download da ultima versão da URGE DevPack no site execute o self-extractor. Ao extrair o DevPack da ultima versão da URGE você deverá encontrar esses arquivos dentro de sua pasta principal: CodeBlocks = A IDE (Ambiente integrado para desenvolvimento de software) que iremos usar, essa é a principal pasta do pacote.

- Media: Uma pasta contendo imagens e recursos gráficos de exemplos.
- MinGW: O Compilador contendo a URGE. Ferramenta necessária para a geração do Jogo em si.
- MinGWpack: Pack para um MinGW já existente, caso prefira usar seu MinGW.
- Utilities: Vários programas para ajudar a criar recursos para tornar o seu jogo o melhor possível.

B.2 Primeiro Passo: Instalação

- Entre na Pasta CodeBlocks, e execute o programa “codeblocks.exe”
- Vá em “Settings”, “Compiler and Debugger...”
- Clique na aba “Toolchain executables”
- Modifique o diretório do compilador em “Compiler’s installation directory” para a pasta MinGW, contida no DevPack (Caso já tenha um MinGW com a URGE instalada, pode-se usa-lo também).

Pronto! Parabéns, você está pronto para usar a URGE!

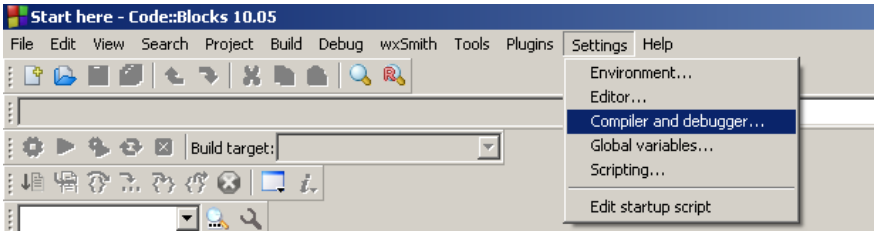

Figura 82: Instalação da URGE

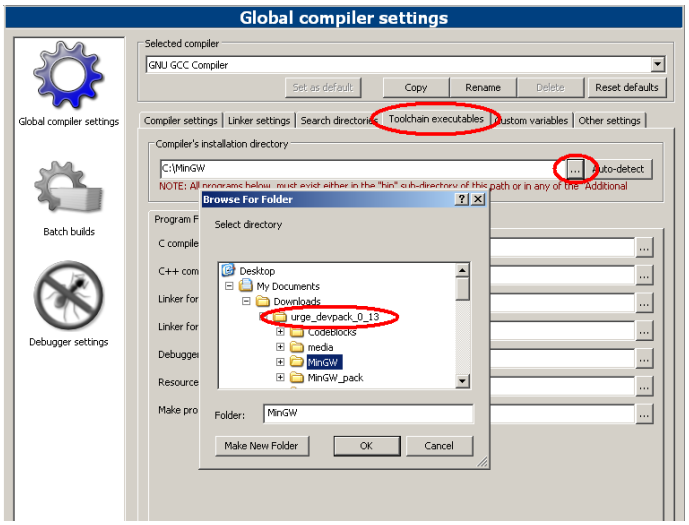

Figura 83: Instalação do compilador com a URGE

B.3 Segundo Passo: Novo Projeto

- Clique em: “File”, “New”, “Project”

- Procure por URGE Project, então aperte “Go”

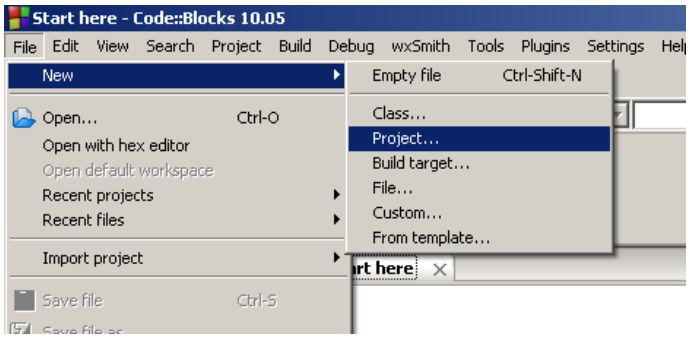

Figura 84: Criando um novo projeto

- Clique em “Next”, até a tela do Titulo do Projeto aparecer.

- No campo “folder to create project in”, clique em “...” e selecione a pasta que deseja colocar seu projeto (Atenção: Esta pasta ter sua localização em mente para podermos colocar imagens e outros importantes recursos no jogo futuramente).

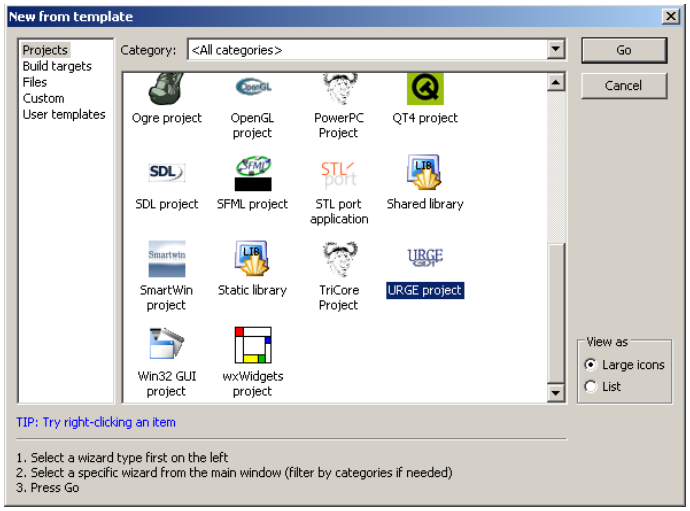

Figura 85: Selecionando um novo projeto

- Preencha o Titulo do Projeto e clique em “Next”

- Clique em “Finish”

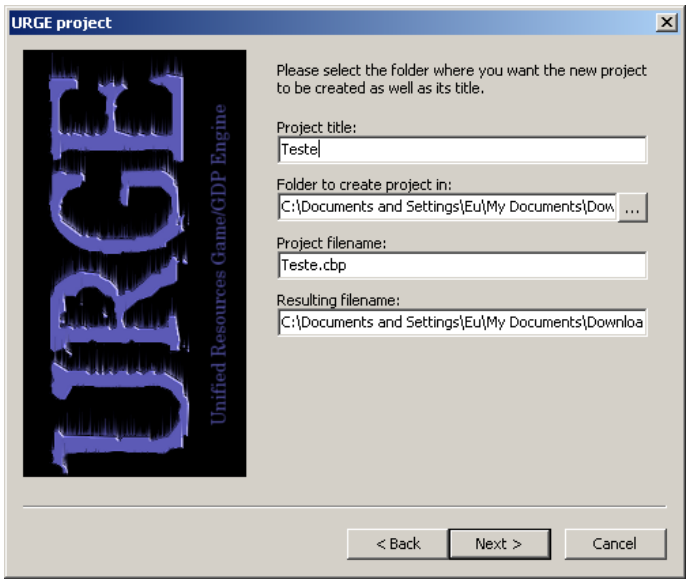

Figura 86: Criando um novo projeto da URGE

Parabéns, um Novo Projeto com um "Hello World"foi criado!

Teste o Programa gerado, para ver se a Engine foi instalada corretamente. Clique em “Build and Run” e Confira se o Hello World! foi executado com sucesso.

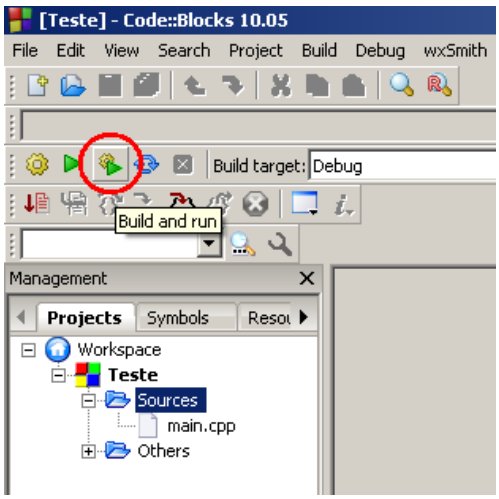

Figura 87: Criando e executando um programa criado com a URGE

APÊNDICE C – Enunciado do trabalho para a disciplina Computação Gráfica I

Regras:

- Trabalhos devem ser feitos em duplas
- Quem fizer individual perderá 1.0 ponto
- As duplas não podem ser repetidas em outros trabalhos
- Apresentação do trabalho:

  Cada dupla terá 10 minutos para apresentar no laboratório

  A demonstração deverá conter:

  Exemplos de execução

  Código compilando

  Entrega no dia 07/12

- Nota:

  Quem realizar corretamente o mínimo terá garantida a nota 6.0

  Quem realizar mínimo + (A xor B xor C) terá garantida a nota 7.0

  Para obter nota maior que 7.0 a implementação deve fazer algo a mais.

Sobre o desenvolvimento:

- Usando a URGE, desenvolva um GameArt nas seguintes especificações:
- O Jogador é também a câmera - First Person Game.

- O Jogo possui apenas um cenário.
- O cenário deve obrigatoriamente possuir um céu, isto é, passe a impressão de que o jogador está andando a céu aberto.
- Neste cenário deve haver um terreno não plano.
- Crie uma primitiva geométrica e insira-a no cenário, está será um pickup do jogo (Item a qual causa algum efeito ao Jogador, caso ele o pegue).
- Crie pelo menos um Inimigo (não se preocupe com animação, pode ser um modelo 3D ou 2D).
- O Objetivo do Jogo é. de alguma forma, derrotar o inimigo com a ajuda do item.
- Crie uma das seguintes opções de Iluminação:

  A - Iluminação "global", uma iluminação que simula a luz do dia ou da noite causada pelo sol. (Atenção: Isso não é uma luz posicional com uma forte componente Ambiente)

  B - Iluminação pontual em mais de um lugar do cenário. Por exemplo: vários postes de luz, ou objetos que "emitem luz".

  C - Iluminação Spot, uma luz que simula uma lanterna na mão do Jogador.

Exemplos de jogos:

- Inicialmente o jogador deve evitar o contato com inimigo, pois se isso acontecer ele será penalizado com o decremento de pontos de vida.
- Após a coleta do item, o jogador pode então colidir com o inimigo, que este será derrotado.
- Inicialmente o Jogo se encontra com a iluminação baixa, no escuro. O item serve como uma lanterna para auxiliar o Jogador a derrotar o inimigo.
- O Jogador pode derrotar o inimigo ao colidir com ele usando algum comando do teclado. O item servirá como um lifeup, incrementando os pontos de vida do Jogador.

Repare que as especificações não são técnicas! Depende de você traduzir o requerimento do Game Designer em conceitos aprendidos em Computação Gráfica I.

Exemplos de Extras:

- Crie um sistema de Dia e Noite, ou seja, o jogo começa de dia e conforme o tempo passa a "luz do sol"vai desaparecendo até ficar de noite.
- Crie uma animação para o Inimigo
- Crie uma Textura com Bump Mapping ou Parallax Mapping no terreno.
- Utilize o efeito de Reflexão da URGE
- Utilize o efeito de Refração da URGE.
- Crie mais de um Tipo de Inimigo
- Crie um Lago Usando o Ocean Rendering da URGE

Referências

[AM00] Ulf Assarsson and Tomas Möller. Optimized view frustum culling algorithms for bounding boxes. *J. Graph. Tools*, 5(1):9–22, January 2000.

[AMHH08] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[Ast06a] D. Astle. *More OpenGL Game Programming*. Thomson/Course Technology, 2006.

[Ast06b] D. Astle. *More OpenGL Game Programming*. Thomson/Course Technology, 2006.

[Bli77] James F. Blinn. Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, 11(2):192–198, July 1977.

[Bli78] James F. Blinn. *Simulation of Wrinkled Surfaces*, volume 12. ACM, 3 edition, 1978.

[Coh06] Mike Cohn. I didn't know i needed that! finding features to satisfy your customers. 2006.

[CON99] Brian Cabral, Marc Olano, and Philip Nemec. Reflection space image based rendering. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '99, pages 165–170, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[Die09] Mike Diehl. 3-d graphics programming with irrlicht. *Linux J.*, 2009(180), April 2009.

[dTWL07] Rodrigo de Toledo, Bin Wang, and Bruno Levy. Geometry textures. In *Proceedings of SIBGRAPI 2007 - XX Brazilian Symposium on Computer Graphics and Image Processing*, pages 79–86, Belo Horizonte, October 2007. SBC - Sociedade Brasileira de Computacao, IEEE Press.

[Eri05] C. Ericson. *Real-Time Collision Detection*. Number v. 1 in Morgan Kaufmann Series in Interactive 3D Technology. Elsevier, 2005.

[GB] Emmanuel Gaillot and Emily Bache. Whatiscodingdojo. http://codingdojo.org/cgi-bin/wiki.pl?WhatIsCodingDojo.

[GDD00] Dan Ginsburg, M. DeLoura, and M.A. DeLoura. *Game programming gems*. Number v. 1 in Graphics series. Charles River Media, 2000.

[Got00] S. Gottschalk. *Collision Queries Using Oriented Bounding Boxes*. University of North Carolina at Chapel Hill, 2000.

[Gou71] Henri Gouraud. *Computer display of curved surfaces.* PhD thesis, 1971. AAI7127878.

[Gre04] Simon Green. History of programmability in opengl. 2004.

[Hop97] Hugues Hoppe. View-dependent refinement of progressive meshes. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, pages 189–198, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[Joh02] Jim Johnson. Standish group chaos report. 2002.

[KL96] Venkat Krishnamurthy and Marc Levoy. Fitting smooth surfaces to dense polygon meshes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '96, pages 313–324, New York, NY, USA, 1996. ACM.

[LaM95] A. LaMothe. *Black art of 3D game programming.* Waite Group Press, 1995.

[Len11] Eric Lengyel. *Mathematics for 3D Game Programming and Computer Graphics.* ISBN, 3 edition, 2011.

[Lue03] D. Luebke. *Level of Detail for 3D Graphics.* The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling. Morgan Kaufmann, 2003.

[Mir96] B.V. Mirtich. *Impulse-based Dynamic Simulation of Rigid Body Systems.* University of California, Berkeley, 1996.

[Mit07] Martin Mittring. Finding next gen: Cryengine 2. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, pages 97–121, New York, NY, USA, 2007. ACM.

[MM05] Morgan McGuire and Max McGuire. Steep parallax mapping. *I3D 2005 Poster*, 2005.

[MPM02] Rafal Mantiuk, Sumanta Pattanaik, and Karol Myszkowski. Cube-map data structure for interactive global illumination computation . . . In *In Proceedings of International Conference on Computer Vision and Graphics*, pages 530–538, 2002.

[Ope] OpenGL. Opengl 2.1 reference. http://www.opengl.org/sdk/docs/man/xhtml.

[Pho75] Bui-Tuong Phong. Illumination for Computer Generated Pictures. 18(6):311–317, 1975.

[POC05] Fabio Policarpo, Manuel M. Oliveira, and João Comba. *Real-Time Relief Mapping on Arbitrary Polygonal Surfaces.* ACM, 1 edition, 2005.

[Ree83] W. T. Reeves. *Particle Systems - A Technique for Modeling a Class of Fuzzy Objects*, volume 17. ACM, 3 edition, 1983.

[Ree85] W. T. Reeves. *Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems*, volume 19. ACM, 3 edition, 1985.

[Rey87] C. W. Reynolds. *Flocks, Herds, and Schools: A Distributed Behavioral Model*, volume 21. ACM, 1987.

[RLKG+09] Randi J. Rost, Bill Licea-Kane, Dan Ginsburg, John M. Kessenich, Barthold Lichtenbelt, Hugh Malan, and Mike Weiblen. *OpenGL Shading Language*. Addison-Wesley Professional, 3rd edition, 2009.

[VVB08] J.M. Van Verth and L.M. Bishop. *Essential Mathematics for Games and Interactive Applications: A Programmer's Guide*. Elsevier Science, 2008.

[WNDS99a] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.1*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2rd edition, 1999.

[WNDS99b] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1999.